Al al-Bayt University

Prince Hussein bin Abdullah College of Information Technology

Computer Science Department

New and Efficient Algorithms for Single Pattern Matching

By

Rami Hasan Omar Mansi

2009

New and Efficient Algorithms for Single Pattern Matching

By

Rami Hasan Omar Mansi

Supervisor: Dr. Jehad Q. Alnihoud

A Thesis Submitted to the
Scientific Research and Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of Master of Science
in Computer Science

<u>Members of the Committee</u>	<u>Approved</u>
Dr. Jehad Q. Alnihoud	
Prof. Adnan M. Al-Smadi	
Dr. Saad Bani Mohammad	
Dr. Ruzayn Quaddoura	

Al al-Bayt University
Mafraq, Jordan
2009

Dedication

I dedicate this thesis to my Family. Without their patience, understanding, support, and most of all, their love, this work would not have been completed.

Acknowledgments

First of all, I would like to thank ALLAH for his graces and guidance. Also, I would like to convey my sincere thanks and deepest appreciation to my supervisor, Dr. Jehad Q. Alnihoud, for his help, guidance, patience and encouragement rendered throughout the different phases of this research.

Also, I would like to thank the members of the thesis examination committee for their advice and comments that have contributed to the improvement of this study.

Table of Contents

List of Tables

List of Figures

List of Appendices

List of Abbreviations

Abbreviation	**Meaning**
ASCII	American Standard Code for Information Interchange
BM	Boyer and Moore algorithm
DDR	Double Data Rate
DNA	Deoxyribo Nucleic Acid
FC-RJ	First Character-Rami and Jehad
FLC-RJ	First and Last Characters-Rami and Jehad
FMLC-RJ	First, Middle and Last Characters-Rami and Jehad
IIDM	Iterative and Incremental Development Method
KMP	Knuth, Morris and Pratt algorithm
KR	Karp and Rabin algorithm
MP	Morris and Pratt algorithm
P	Pattern string
RAM	Random Access Memory
SMT-RJ	String Matching Tool-Rami and Jehad
T	Text string

Abstract

The string matching problem is defined as finding the occurrences of a pattern *P* of length *m* in a text *T* of length *n*. String matching algorithms are considered as one of the most important components used in implementations of practical software under most operating systems. In many information retrieval and text-editing applications, it is necessary to be able to locate quickly some or all occurrences of a user-specified pattern in a text. This study reviewed the most important and mostly used exact single pattern matching algorithms in order to enhance the existing algorithms and propose new single pattern matching algorithms.

In this study, we propose four exact single pattern matching algorithms, First Character-Rami and Jehad (FC-RJ), First and Last Characters-Rami and Jehad (FLC-RJ), First, Middle and Last Characters-Rami and Jehad (FMLC-RJ) and ASCII-Based-Rami and Jehad (ASCII-Based-RJ) algorithms. Furthermore, a string matching tool, called String Matching Tool- Rami and Jehad (SMT-RJ), has been developed, and the proposed algorithms, in addition to the Brute Force and Boyer-Moore algorithms, have been implemented, tested and compared using the developed tool.

The experimental results have shown that the FC-RJ, FLC-RJ and FMLC-RJ algorithms outperformed the Brute Force algorithm, while the ASCII-Based-RJ algorithm outperformed both Brute Force and Boyer-Moore algorithms.

Chapter One: Introduction

The string matching problem, also called pattern matching, may be defined as finding one or more of the occurrences of a pattern *P* of length *m* in a text *T* of length *n* (Gongshe, 2006). It has been extensively studied, and many techniques and algorithms have been designed to solve this problem. These algorithms are mostly used in information retrieval, bibliographic search, molecular biology, and question answering applications (Lecroq, 2007; Wu *et al*., 2007).

String matching is a very important subject in the wider domain of text processing and its algorithms are the basic components used in implementations of practical software under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (Watson and Watson, 2003).

In many information retrieval and text-editing applications, it is necessary to be able to locate quickly some or all occurrences of a user-specified pattern of words and phrases in a text (Amir *et al*., 2002; Alqadi *et al*., 2007). Furthermore, string matching has many applications including database query, DNA and protein sequence analysis. Therefore, the efficiency of string matching has a great impact on the performance of these applications (Crochemore *et al*., 2003). Although data are memorized in various ways, text remains the main and most efficient form to exchange information (Kim and Kim, 1999; Sheu *et al*., 2008).

Basically, a string matching algorithm uses a window to scan the text. The size of this window is equal to the length of the pattern. It first aligns the left ends of the window and the text. Then it checks if the pattern occurs in the window (this specific work is called an *attempt*) and *shifts* the window to the right. It repeats the same procedure again until the right end of the window goes beyond the right end of the text (Amintoosi *et al*., 2006; Rytter, 2007).

Exact string matching means finding one or all exact occurrences of a pattern in a text (Idury and Schaffer, 1995; Watson, 2002). Brute force algorithm, as mentioned in (Charras and Lecroq, 2004), Boyer-Moore (Boyer and Moore, 1977), Morris-Pratt

(Morris and Pratt, 1970), and Knuth-Morris-Pratt (Knuth *et al*., 1977), are exact string matching algorithms.

Approximate string matching is the technique of finding approximate (may not be exact) matches to a pattern in a string (Lipsky and Porat, 2007). The closeness of a match is measured in terms of the number of primitive operations necessary to convert the string into an exact match. The usual primitive operations are insertion, deletion and substitution (Navarro and Fredriksson, 2004). So the input of an approximate string matching algorithm is a text string *T*, a pattern string *P*, and an edit cost bound *k*, and the task of the algorithm is to answer the question: can we transform a part of *T* to *P* using at most *k* additions, deletions, and substitutions? (Cegielski *et al*., 2006; Michailidis and Margaritis, 2007).

Some of the exact string matching algorithms have been presented to solve the problem of searching for a single pattern in a text, such as Boyer-Moore (Boyer and Moore, 1977), Morris-Pratt (Morris and Pratt, 1970), Knuth-Morris-Pratt (Knuth *et al*., 1977) and Karp-Rabin (Karp and Rabin, 1987) algorithms. In the other hand, some have been presented to solve the problem of searching for multiple patterns in a text, such as Aho-Corasick (Aho and Corasick, 1975), Wu-Manber (Wu and Manber, 1994), Kim-Kim (Kim and Kim, 1999), and Cantone-Faro (Cantone and Faro, 2006) algorithms. Although the Knuth-Morris-Pratt algorithm has better worst-case running time than the Boyer-Moore algorithm, the latter is known to be extremely efficient in practice (Crochemore *et al*., 1994; Amintoosi *et al*., 2006).

Since 1977, with the publication of the Boyer–Moore algorithm, there have been many papers published that deal with exact pattern matching, and in particular discuss and/or introduce variants of Boyer-Moore algorithm.

The string matching literature has had two main categories (Danvy and Rohde, 2006; Franek *et al*., 2006):

1. Reducing the number of character comparisons required in the worst and average cases.
2. Reducing the time requirement in the worst and average cases.

This study is an attempt to enhance the time complexity of some common string matching algorithms in their best, average and worst cases.

In order to reduce the processing time of some common string matching algorithms, we propose four exact single pattern matching algorithms. The proposed algorithms improve the length of the shifts of some common string matching algorithms.

The extensive testing of the proposed algorithms yields to speed up some of existing string matching algorithms.

1.1. Scope of the Study

This study focuses on the exact single pattern matching algorithms and enhancing some of the existing algorithms that fall in this scope, such as Boyer-Moore algorithm (Boyer and Moore, 1977) and the Naïve (brute force) algorithm.

1.2. Aims and Objectives

The study aims to propose new and efficient exact single pattern matching algorithms that can be applied in the real-world applications, such as text editors, question-answering and database applications. Moreover, the proposed algorithms enhance the time and space complexity of the existing algorithms.

The study seeks to achieve the following objectives (sub-goals) with respect to the currently applied exact single pattern matching algorithms:

- Enhancing the preprocessing phase.
- Enhancing the searching phase.
- Enhancing (decreasing) the time and space complexities.
- Facilitating and simplifying the implementation.
- Proposing clearer and simpler algorithms to solve the problem.

1.3. Significance of the Study

This study serves the software developers, software users, and researchers who are interested in string matching in the following points:

- The developer will be able to use the new algorithms in his/her system, which will lead to the evolution of the system.

- The developer and the user will be able to decide which algorithm is most suitable to be used to serve his/her needs based on the nature of the available data in the system.
- This study helps software developers, software users, and researchers to collect information about the previously developed algorithms in the field of single pattern matching.

1.4. Contributions

The research contributions may be recorded as follows:

- Proposing new algorithms which decrease the time and space needed as compared to some of the currently used algorithms for solving the problem of single pattern matching.
- The new algorithms simplify the implementation and increase the clarity compared with other string matching algorithms.
- Developing an efficient simulation tool which specialized in implementing and testing string matching algorithms.

1.5. Thesis Organization

This thesis is organized as follows:

- Chapter two presents previous related works and gives a comparison between the previously created algorithms that belong to the scope of this study.
- Chapter three presents the methodology that has been followed during the work of this study.
- Chapter four introduces the proposed algorithms and presents their detailed analysis.
- Chapter five describes the simulation tool that has been developed in order to test the performance of the proposed algorithms.
- Chapter six presents a performance comparison between the proposed algorithms and other related algorithms.
- Chapter seven draws the conclusions of this study.

Chapter Two: Related Works

2.1. Brute Force Algorithm

The brute force algorithm, as mentioned in (Charras and Lecroq, 2004), consists of checking, at all positions in the text between (0) and ($n - m$), where n is the length of the text and m is the length of the pattern, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase, the text character comparisons can be done in any order. The time complexity of the searching phase is O(mn), and the expected number of text character comparisons is ($2n$). Example 2.1 illustrates the work of the brute force algorithm.

Example 2.1: A Brute Force Example:

Assume that we have the following text:

M	A	A	C	T	R	A	A	C	O

And we want to find all occurrences of the following pattern in the text:

A	A	C

Then, the searching will be as follows:

In the first attempt, the algorithm compares the characters of the pattern with the first m characters of the text. If a mismatch occurred (or a complete match of the whole pattern), the searching will be shifted by one character to be started at the next character of the text.

First Attempt:

M	A	A	C	T	R	A	A	C	O
A	A	C							

Second Attempt:

M	A	A	C	T	R	A	A	C	O
	A	A	C						

Third Attempt:

M	A	A	C	T	R	A	A	C	O
		A	A	C					

Fourth Attempt:

M	A	A	C	T	R	A	A	C	O
			A	A	C				

Fifth Attempt:

M	A	A	C	T	R	A	A	C	O
				A	A	C			

Sixth Attempt:

M	A	A	C	T	R	A	A	C	O
					A	A	C		

Seventh Attempt:

M	A	A	C	T	R	A	A	C	O
						A	A	C	

Eighth Attempt:

M	A	A	C	T	R	A	A	C	O
							A	A	C

2.2. Morris-Pratt Algorithm

A left to right string matching algorithm, called Morris-Pratt algorithm (Morris and Pratt, 1970), is presented. The design of the Morris-Pratt algorithm follows a tight analysis of the brute force algorithm, and especially on the way the brute force algorithm wastes the information gathered during the scan of the text.

Let us look more closely at the brute force algorithm. It is possible to improve the length of the shifts and simultaneously remember some portions of the text that match the pattern. This saves comparisons between characters of the pattern and characters of the text and consequently increases the speed of the search (Charras and Lecroq, 2004).

Let y is a text of length n, and x is a pattern of length m, then consider an attempt at a left position j on y, that is when the window is positioned on the text factor $y[j \mathinner{..} j + m - 1]$. Assume that the first mismatch occurs between $x[i]$ and $y[i + j]$ with $0 \leq i < m$. Then, $x[0 \mathinner{..} i - 1] = y[j \mathinner{..} i + j - 1] = u$ and $a = x[i] \neq y[i + j] = b$. When shifting, it is reasonable to expect that a prefix v of the pattern matches some suffix of the portion u of the text. The longest such prefix v is called the border of u (it occurs at both ends of u). This introduces the notation: *mpNext*$[i]$ to be the length of the longest border of $x[0 \mathinner{..} i - 1]$ for $0 \leq i < m$. Then, after a shift, the comparisons can resume between characters *c*=*x*[*mpNext*[*i*]] and *y*[*i* + *j*] = *b* without missing any occurrence of *x* in *y*, and avoiding a backtrack on the text. See Figure 2.1.

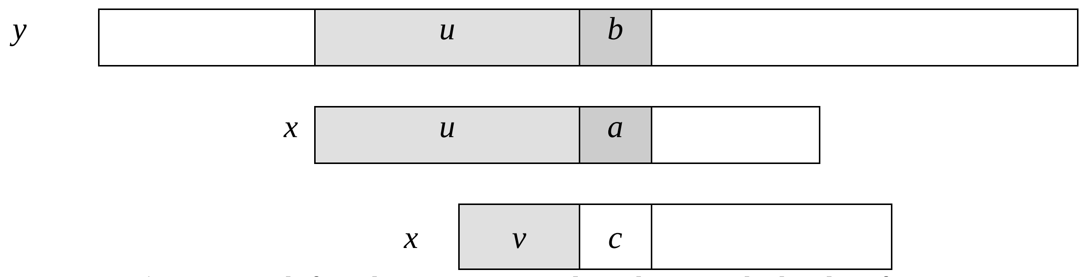


Figure 2.1: Shift in the Morris-Pratt algorithm: *v* is the border of *u*.

The value of *mpNext*[0] is set to (-1) to indicate that if a mismatch occurred at the first character of the pattern, the algorithm cannot backtrack, and it must simply check the next character, since the value of shifting is computed by (*i*-*mpNext*[*i*]), so if (*i*) was (0) then (0 - -1 = 1). The table *mpNext* can be computed in O(*m*) space, and O(*m*) time to scan the pattern's characters, before the searching phase, applying the same searching algorithm to the pattern itself, as is *x* = *y*.

Depending on the detailed analysis of the Morris-Pratt algorithm in (Morris and Pratt, 1970), the searching phase can be done in O(*n* + *m*) time. The algorithm performs at most (2*n* –1) text character comparisons during the searching phase. The delay (maximum number of comparisons for a single text character) is bounded by *m*. See Example 2.2 which illustrates the principle and the work of the Morris-Pratt algorithm.

Example 2.2: A Morris-Pratt Example:

<u>Preprocessing phase</u>:

As illustrated above, the preprocessing phase constructs the *mpNext* table which contains the *mpNext* value of each character in the pattern. The *mpNext* table is constructed by finding the length of the longest border of *u* at each character in the pattern.

The portion *u* contains the sub-string from the first character of the pattern to the (*i*-1)th character of each character at index (*i*) except the first character (when *i* = 0). For example, if we have the pattern: (**ababcda**), then the portion *u* of the fifth character (character **c** at *i* = 4) is the sub-string of the pattern's characters from the first character to the fourth one, which is (**abab**). So, the length of the longest border of *u* is (2), since the characters (**ab**) occur at both ends of *u*. Then the *mpNext* value of the fifth character is (2). This means, if a mismatch occurred at the fifth character of the pattern, then the pattern will be shifted by (2) characters.

This shifting value (2) did not come from the length of the longest border of *u*, it is computed by (*i* – the length of the longest border of *u*), which is, for this example, (4–2 = 2). The *mpNext* value of the first character is always set to (-1) to avoid the backtracking. For the rest of the pattern's characters; the *mpNext* value is computed as illustrated before. The shifting value of the pattern when a mismatch occurs at the i^{th} character is the result of subtracting *mpNext*[*i*] from the value (*i*).

Table 2.1: The *mpNext* table.

i	0	1	2	3	4	5	6	7
x[*i*]	G	C	A	G	A	G	A	G
mpNext[*i*]	-1	0	0	0	1	0	1	0

Searching phase:

First attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
1	2	3	4																				
G	C	A	G	A	G	A	G																

Shift by: 3 (*i*-*mpNext*[*i*]= 3 - 0)

Second attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
			1																				
			G	C	A	G	A	G	A	G													

Shift by: 1 (*i*-*mpNext*[*i*]=0 - -1)

Third attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
				1																			
				G	C	A	G	A	G	A	G												

Shift by: 1 (*i*-*mpNext*[*i*]=0 - -1)

Fourth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
					1	2	3	4	5	6	7	8											
					G	C	A	G	A	G	A	G											

Shift by: 7 (*i*-*mpNext*[*i*]= 7 - 0)

Fifth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
												1	2										
												G	C	A	G	A	G	A	G				

Shift by: 1 (*i*-*mpNext*[*i*]= 1 - 0)

Sixth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
													1										
													G	C	A	G	A	G	A	G			

Shift by: 1 (*i*-*mpNext*[*i*]= 0 - -1)

Seventh attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
														1									
														G	C	A	G	A	G	A	G		

Shift by: 1 (*i*-*mpNext*[*i*]= 0 - -1)

Eighth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
															1								
															G	C	A	G	A	G	A	G	

Shift by: 1 (*i*-*mpNext*[*i*]= 0 - -1)

Ninth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
																1							
																G	C	A	G	A	G	A	G

Shift by: 1 (*i*-*mpNext*[*i*]= 0 - -1)

The Morris-Pratt algorithm performs (20) character comparisons in this example.

In order to simplify the illustration and to make a clear comparison between the algorithms of this literature; the text and the pattern of the next examples are the same as those that have been used in Example 2.2.

2.3. Knuth-Morris-Pratt Algorithm

In (Knuth *et al*., 1977), a left to right string matching algorithm has presented, referred to as Knuth-Morris-Pratt algorithm. The design of the Knuth-Morris-Pratt (KMP) algorithm follows a tight analysis of the Morris-Pratt algorithm. Let us look more closely at the Morris-Pratt algorithm. It is possible to improve the length of the shifts.

Consider an attempt at a left position *j*, that is when the window is positioned on the text factor *y*[*j* .. *j* + *m* – 1]. Assume that the first mismatch occurs between *x*[*i*] and *y*[*i* +

j] with $0 \le i < m$. Then, $x[0 .. i-1] = y[j .. i+j-1] = u$ and $a = x[i] \ne y[i+j] = b$. When shifting, it is reasonable to expect that a prefix v of the pattern matches some suffix of the portion u of the text. Moreover, if we want to avoid another immediate mismatch, the character following the prefix v in the pattern must be different from a. The longest such prefix v is called the tagged border of u (it occurs at both ends of u followed by different characters in x). This introduces the notation: *kmpNext*[i] to be the length of the longest border of $x[0 .. i-1]$ followed by a character c different from $x[i]$, and (-1) if no such tagged border exists, for $0 \le i < m$. Then, after a shift, the comparisons can resume between characters $c = x[kmpNext[i]]$ and $y[i + j]$ without missing any occurrence of x in y, and avoiding a backtrack on the text. See Figure 2.2.

The value of *kmpNext*[0] is set to (-1) to indicate that if a mismatch occurred at the first character of the pattern, we cannot backtrack, and we must check the next character, since the value of shifting is computed by (i-*kmpNext*[i]), so if (i) was (0) then (0--1=1). The table *kmpNext* can be computed in O(m) space and time before the searching phase, applying the same searching algorithm to the pattern itself, as is $x = y$.

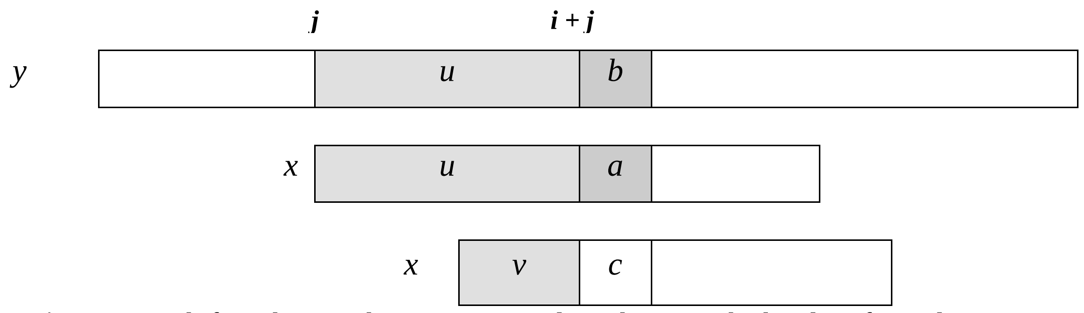


Figure 2.2: Shift in the Knuth-Morris-Pratt algorithm: v is the border of u and $a \ne c$.

As illustrated in the detailed analysis of the KMP algorithm in (Knuth *et al.*, 1977), the searching phase can be performed in O($n + m$) time. The KMP algorithm performs at most ($2n - 1$) text character comparisons during the searching phase. The delay (maximum number of comparisons for a single text character) is bounded by $\log_{\Phi}(m)$, where Φ is the golden ratio ($\Phi = (1 + 5^{0.5}) / 2$). See Example 2.3.

Example 2.3: A Knuth-Morris-Pratt Example:

Preprocessing phase:

The preprocessing phase constructs the *kmpNext* table which contains the *kmpNext* value of each character. The *kmpNext* table is constructed as the *mpNext* table of the Morris-Pratt algorithm with a difference.

In Morris-Pratt algorithm, *mpNext*[*i*] is the length of the longest border of *x*[0.. *i* –1], but in Knuth-Morris-Pratt, *kmpNext*[*i*] is the length of the longest border of *x*[0.. *i*– 1] followed by a character *c* different from *x*[*i*], and (-1) if no such tagged border exists, for $0 \leq i < m$. The value *kmpNext*[0] is set to (-1), to indicate that if a mismatch occurred at the first character of the pattern, we cannot backtrack, and we must simply check the next character. The shifting value of the pattern at the i^{th} character is the result of subtracting *kmpNext*[*i*] from the value (*i*).

Table 2.2: The *kmpNext* table.

i	0	1	2	3	4	5	6	7
x[*i*]	G	C	A	G	A	G	A	G
kmpNext[*i*]	-1	0	0	-1	1	-1	1	-1

Searching phase:

First attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
1	2	3	4																				
G	C	A	G	A	G	A	G																

Shift by: 4 (*i*-*kmpNext*[*i*]=3 - -1)

Second attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
				1																			
				G	C	A	G	A	G	A	G												

Shift by: 1 (*i*-*kmpNext*[*i*]=0 - -1)

Third attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
					1	2	3	4	5	6	7	8											
					G	C	A	G	A	G	A	G											

Shift by: 7 (whole match)

Fourth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
												1	2										
												G	C	A	G	A	G	A	G				

Shift by: 1 (*i*-*kmpNext*[*i*]=1 - 0)

Fifth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
													1										
													G	C	A	G	A	G	A	G			

Shift by: 1 (*i*-*kmpNext*[*i*]=0 - -1)

Sixth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
														1									
														G	C	A	G	A	G	A	G		

Shift by: 1 (*i*-*kmpNext*[*i*]=0 - -1)

Seventh attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
															1								
															G	C	A	G	A	G	A	G	

Shift by: 1 (*i*-*kmpNext*[*i*]=0 - -1)

Eighth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
																1							
																G	C	A	G	A	G	A	G

Shift by: 1 (*i*-*kmpNext*[*i*]=0 - -1).

The Knuth-Morris-Pratt algorithm performs (19) character comparisons in this example.

2.4. Boyer-Moore Algorithm

A string searching algorithm, called Boyer-Moore, is presented in (Boyer and Moore, 1977). The Boyer-Moore (BM) algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of mismatch, or a complete match of the whole pattern, it uses two pre-computed functions to shift the window to the right. These two shift functions are called the *good-suffix shift* and *bad-character shift*.

Assume that a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i+j] = b$ for the text during an attempt at position (j) on the text. Then, $x[i + 1 .. m - 1] = y[i + j + 1 .. j + m - 1] = u$ and $x[i] \neq y[i + j]$. The *good-suffix shift* consists in aligning the segment $y[i + j + 1 .. j + m - 1] = x[i + 1 .. m - 1]$ with its rightmost occurrence in x that is preceded by a character different from $x[i]$, see Figure 2.3. If there exists no such segment, the shift consists in aligning the longest suffix v of $y[i + j + 1 .. j + m - 1]$ with a matching prefix of x, see Figure 2.4.

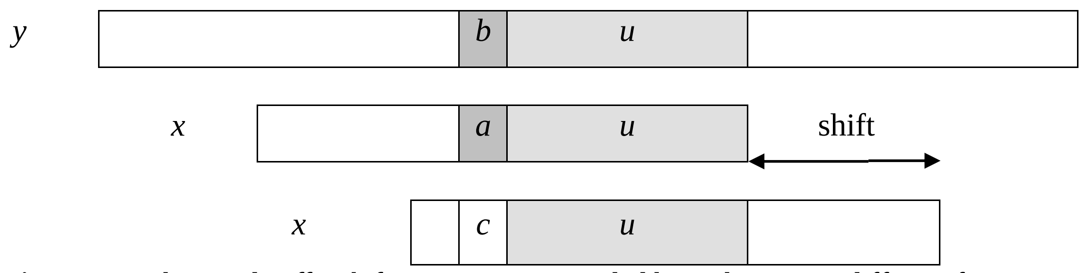


Figure 2.3: The *good-suffix shift*, *u* re-occurs preceded by a character *c* different from *a*.

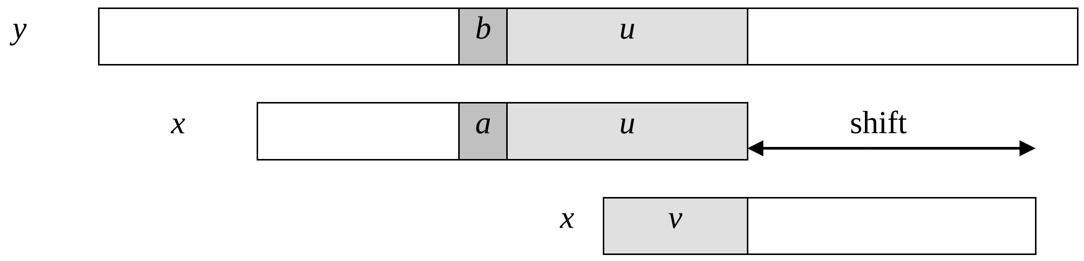


Figure 2.4: The *good-suffix shift*, only a suffix of *u* re-occurs in *x*.

The *bad-character shift* consists in aligning the text character $y[i + j]$ with its rightmost occurrence in $x[0 .. m - 2]$, see Figure 2.5. If $y[i + j]$ does not occur in the pattern x, no occurrence of x in y can include $y[i + j]$, and the left end of the window is aligned with the character immediately after $y[i + j]$, namely $y[i + j + 1]$. See Figure 2.6.

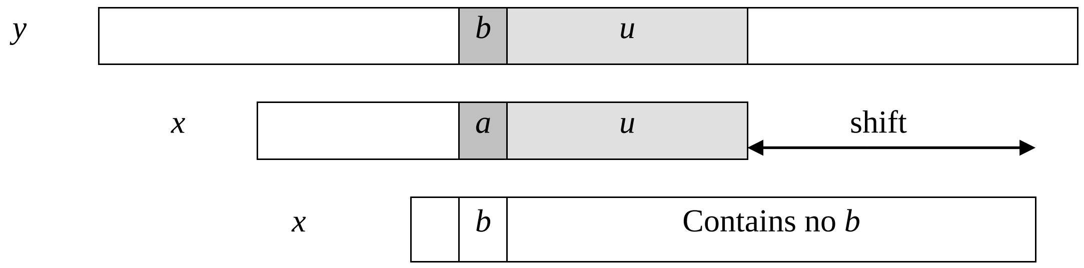


Figure 2.5: The *bad-character shift*, *b* occurs in *x*.

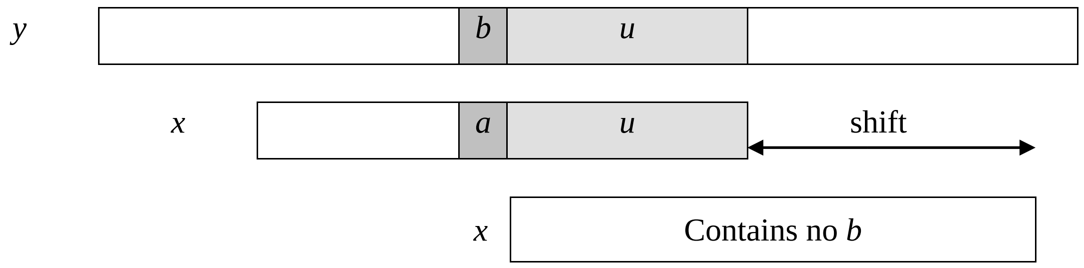


Figure 2.6: The *bad-character shift*, *b* does not occur in *x*.

Note that the *bad-character shift* can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the *good-suffix shift* and the *bad-character shift*.

More formally, the two shift functions are defined in (Boyer and Moore, 1977) as follows:

- The *good-suffix shift* function is stored in a table *bmGs* of size $m+1$.

Two conditions have been defined:

$Cs(i,s)$: for each k such that $i < k < m, s \geq k$ or $x[k - s] = x[k]$, and,

$Co(i,s)$: if $s < i$ then $x[i - s] \neq x[i]$.

Then, for $0 \leq i < m$:

bmGs$[i + 1] = \min \{ s > 0 : Cs(i , s)$ and $Co(i , s)$ hold $\}$.

And *bmGs*[0] is defined as the length of the period of *x*. The computation of the table *bmGs* uses a table *suff* which can be defined as follows:

for $1 \leq i < m$, $suff[i] = \max \{k: x[i - k + 1 .. i] = x[m - k , m - 1] \}$.

- The *bad-character shift* function is stored in a table *bmBc* of size σ.

For *c* in ∑:

$bmBc[c] = \min \{i: 1 \leq i < m - 1$ and $x[m - 1 - i] = c\}$ if *c* occurs in *x*, otherwise, $bmBc[c] = m$.

As discussed in (Boyer and Moore, 1977), tables *bmBc* and *bmGs* can be pre-computed in time $O(m + \sigma)$ before the searching phase, and require an extra-space in $O(m + \sigma)$. The searching phase time complexity is quadratic $O(mn)$, but at most, (3n) text character comparisons are performed when searching for a non periodic pattern. On large alphabets, relatively to the length of the pattern, the algorithm is extremely fast. See Example 2.4, which illustrates the principle of the Boyer-Moore algorithm.

Example 2.4: A Boyer-Moore Example:

<u>Preprocessing phase</u>:

Table 2.3: The *bmBc* table.

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

Table 2.4: The *bmGs* table.

i	0	1	2	3	4	5	6	7
x[*i*]	G	C	A	G	A	G	A	G
suff[*i*]	1	0	0	2	0	4	0	8
bmGs[*i*]	7	7	7	2	7	4	7	1

<u>Searching phase</u>:

First attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
							1																
G	C	A	G	A	G	A	G																

Shift by: 1 (*bmGs*[7]=*bmBc*[A]-8+8)

Second attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
						3	2	1															
	G	C	A	G	A	G	A	G															

Shift by: 4 (*bmGs*[5]=*bmBc*[C]-8+6)

Third attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
					8	7	6	5	4	3	2	1											
					G	C	A	G	A	G	A	G											

Shift by: 7 (*bmGs*[0])

Fourth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
																	3	2	1				
												G	C	A	G	A	G	A	G				

Shift by: 4 (*bmGs*[5]=*bmBc*[C]-8+6)

Fifth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
																						2	1
																G	C	A	G	A	G	A	G

Shift by: 7 (*bmGs*[6])

The Boyer-Moore algorithm performs (17) character comparisons in this example.

2.5. Karp-Rabin Algorithm

In (Karp and Rabin, 1987), an algorithm to find the first occurrence of a pattern *x* in a text *y* using a hashing function has been presented. This algorithm is called Karp-Rabin algorithm (KR algorithm). Hashing provides a simple method to avoid a quadratic number of character comparisons in most practical situations. Instead of checking at each position of the text if the pattern occurs, it seems to be more efficient to check only if the content of the window looks like the pattern. Example 2.5 clarifies the principle of the algorithm.

In order to check the resemblance between these two words, a hashing function is used. This function is called (*hash*), and should have the following properties:

- Efficiently computable.
- Highly discriminating for strings.
- $hash(\ y[\ j + 1 .. j + m])$ must be easily computable from $hash(\ y[\ j .. j + m - 1])$ and $y[\ j + m])$: $hash(\ y[\ j + 1 .. j + m]) = rehash(y[\ j], y[\ j + m], hash(\ y[\ j .. j + m - 1]))$.

For a word *w* of length *m*, let *hash*(*w*) be defined as follows:
$hash(w[0 .. m - 1]) = (\ w[0] \text{ x } 2^{m-1} + w[1] \text{ x } 2^{m-2} + \ldots + w[m - 1] \text{ x } 2^{0} \bmod q$, where *q* is a large number. Then, $rehash(a\ ,b\ ,h) = ((\ h - a \text{ x } 2^{m-1}) \text{ x } 2 + b) \bmod q$.

The preprocessing phase of the KR algorithm consists in computing *hash*(x). It can be done in constant space and O(m) time complexity. During the searching phase, it is enough to compare *hash*(x) with *hash*($y[j\ ..\ j+m-1]$) for $0 \le j \le n - m$. If an equality is found, it is still necessary to check the quality: $x = (y[j\ ..\ j + m - 1])$ character by character.

The time complexity of the searching phase of the Karp-Rabin algorithm is O(mn) and its expected number of text character comparisons is O($m + n$).

Example 2.5: A Karp-Rabin Example:

Preprocessing phase:

Let *hash*[y] = 17597.

Searching phase:

First attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
G	C	A	G	A	G	A	G																

hash(y[0 .. 7]) = 17819

Second attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
	G	C	A	G	A	G	A	G															

hash(y[1 .. 8]) = 17533

Third attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
		G	C	A	G	A	G	A	G														

hash(y[2 .. 9]) = 17979

Fourth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
			G	C	A	G	A	G	A	G													

hash(y[3 .. 10]) = 19389

Fifth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
				G	C	A	G	A	G	A	G												

hash(y[4 .. 11]) = 17339

Sixth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
					G	C	A	G	A	G	A	G											

hash(y[5 .. 12]) = 17597

Seventh attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
						G	C	A	G	A	G	A	G										

hash(*y*[6 .. 13]) = 17102

Eighth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
							G	C	A	G	A	G	A	G									

hash(*y*[7 .. 14]) = 17117

Ninth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
								G	C	A	G	A	G	A	G								

hash(*y*[8 .. 15]) = 17678

Tenth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
									G	C	A	G	A	G	A	G							

hash(*y*[9 .. 16]) = 17245

Eleventh attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
										G	C	A	G	A	G	A	G						

hash(*y*[10 .. 17]) = 17917

Twelfth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
											G	C	A	G	A	G	A	G					

hash(*y*[11 .. 18]) = 17723

Thirteenth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
												G	C	A	G	A	G	A	G				

hash(*y*[12 .. 19]) = 18877

Fourteenth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
													G	C	A	G	A	G	A	G			

hash(*y*[13 .. 20]) = 19662

Fifteenth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
														G	C	A	G	A	G	A	G		

hash(*y*[14 .. 21]) = 17885

Sixteenth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
															G	C	A	G	A	G	A	G	

hash(*y*[15 .. 22]) = 19197

Seventeenth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
																G	C	A	G	A	G	A	G

hash(y[16 .. 23]) = 16961.

The Karp-Rabin algorithm performs (8) character comparisons in this example.

2.6. Comparisons

Table 2.5 shows the main differences between the most important exact single pattern matching algorithms.

Table 2.5: A comparison between string matching algorithms.

Algorithm	Worst Case Complexity	Description
Brute Force	Quadratic: O(mn)	- Left to right - The expected number of text character comparisons is 2n
Morris-Pratt	Preprocessing: O(m) Searching: O(n+m)	- Left to right - Delay is O(m)
Knuth-Morris-Pratt	Preprocessing: O(m) Searching: O(n+m)	- Left to right - Delay is log$_\Phi$(m)
Boyer-Moore	Preprocessing: O(m + σ) Searching: O(mn)	- Right to left - 3n text character comparisons
Karp-Rabin	Preprocessing: O(m) Searching: O(mn)	- Uses a hashing function - O(m+n) text character comparisons

Chapter Three: Methodology

The method that has been followed by this study is the **Iterative and Incremental Development Method** (**IIDM**). IIDM is a cyclic software development process which developed in response to the weaknesses of the waterfall model. It is an essential part of the Rational Unified Process, the Dynamic Systems Development Method, Extreme Programming and generally the Agile Software Development Frameworks (Sommerville, 2001).

IIDM is a rework scheduling strategy in which time is set aside to revise and improve parts of the system. The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system (Sommerville, 2001). Learning comes from both the development and use of the system, where possible. See Figure 3.1 which shows the main procedures of the IIDM.

The main steps of the IIDM that have been followed in this work are as follows:

1. Starting with a very small version of the system with initial requirements.
2. Analysis: deciding which improvement should be made next and making a small change.
3. Design: how to code the change.
4. Coding: coding the modification and compiling to make sure that the coding is correct.
5. Testing: running the program using testing data. If it does not run, the program must be debugged and either the coding or the design should be changed. Because the last modification should have been small, it is usually easy to identify the source of the problem.
6. Continuing around this loop, (from step 2 to 5), until the program is finished, or the specified time for completing the system is up.

The IIDM is considered as the best suitable method for this study due to many reasons:

- Using the iterative approach means that there is a run of the program at the end of each iteration. The run is not for everything; but it is for something the developer can turn in. This means that the user may be able to use the program for some work and get a value out of it before it is finished. It also means that the project can be terminated as soon as the program is good enough.

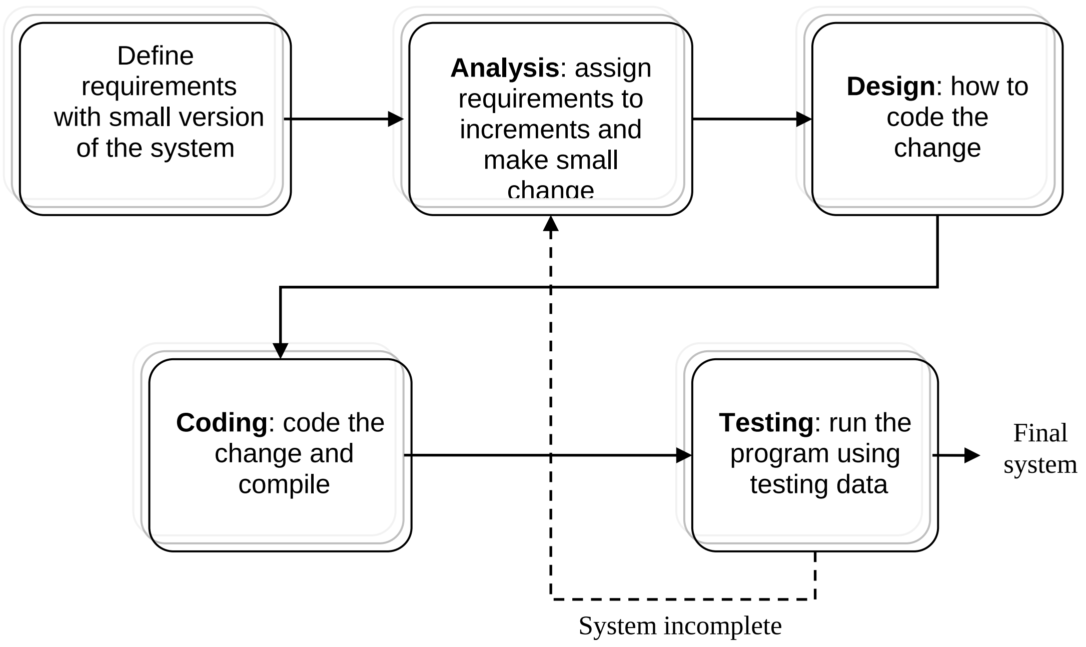

Define requirements with small version of the system
Analysis: assign requirements to increments and make small change
Design: how to code the change
Coding: code the change and compile
Testing: run the program using testing data
Final system
System incomplete

Figure 3.1: Iterative and incremental development method (IIDM).

- By making very small changes; compiling and testing means that the developer is much less likely to be faced with a long list of errors with no idea of how to find the problems. A single missing left brace ({) can produce many error messages. If only a few lines of code are entered before recompiling, the developer knows that the error is in those few lines of code and it will be much easier to track it down.
- The developer always has a running version of the program. So, if the developer (student) runs out of time, he/she can deliver the last iteration, which may not have all functionality, but it does something. This is usually worth more to the thesis than a program which does not compile or run.
- It's psychologically more satisfying to get positive feedback on the work by running the program.
- Corrections early generally take less time than later in the development process.

In the other hand, there are some disadvantages with the IIDM:

- Increments should be relatively small and each increment (modification) should deliver some system functionality. It may therefore be difficult to map the customer's requirements onto increments of the right size.

- Most systems require a set of basic facilities which are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it is difficult to identify common facilities that all increments require.

Figure 3.2 describes how we used the IIDM to develop the proposed algorithms, and how the changes (increments) yield to develop a new algorithm.

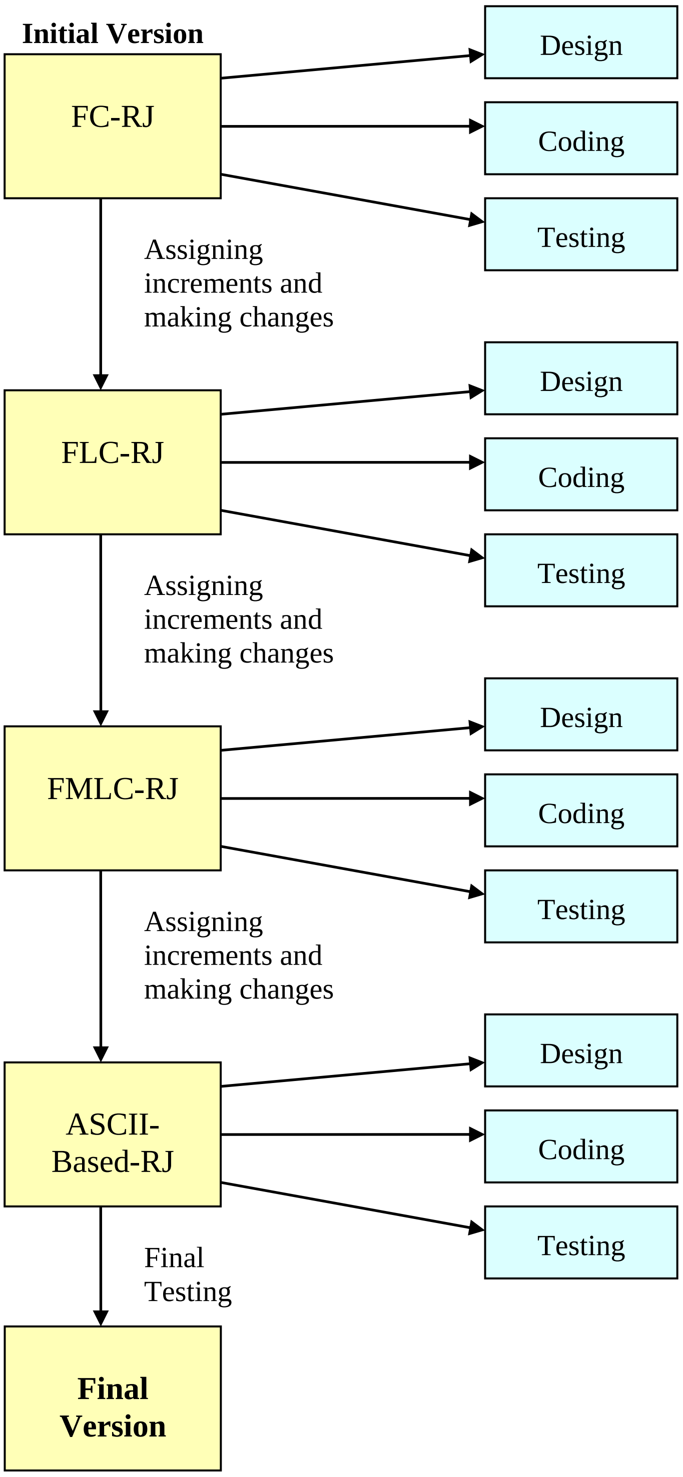


Figure 3.2: Using IIDM in developing the proposed algorithms.

Chapter Four: The Proposed Algorithms

4.1. FC-RJ Algorithm

Most of string matching algorithms search for the pattern in the whole text, and match (compare) most of the text's characters with the pattern's characters (Watson, 2003).

Instead, it is reasonable to assume that it will be more efficient to match the pattern with the sub-strings of the text which start with the pattern's first character, while ignoring the rest of the characters in the text. Depending on the concept and the work of the proposed algorithm, we call it First Character-Rami and Jehad (FC-RJ) algorithm. The FC-RJ algorithm finds the indices of all occurrences of the first character of the pattern in the text prior to the searching phase. These indices should be saved in a list (array) to be accessed during the searching phase, which we name it as (**Occurrence_List**). In the searching phase, the algorithm uses the Occurrence_List to move to the indices of the text that contain the first character of the pattern.

The main procedures of the FC-RJ algorithm are expressed as follows:

a. Preprocessing phase:

1. The algorithm creates a new array called (**Occurrence_List**) of size (n-m+1), where n is the size of the text and m is the size of the pattern. The length of the Occurrence_List is (n-m+1) because it is impossible to the pattern to occur after the position (n-m) in the text. This array will hold the indices of the occurrences of the pattern's first character in the text using an integer variable (i) starting from (0) and incremented by one after each match.
2. The algorithm scans the text in a single pass, using an integer variable (j), and compares its characters with the pattern's first character. If the current character of the text (jth character) is equal to the pattern's first character, the algorithm saves the index of the current character in the text (the value of j) in the ith index of the Occurrence_List array and increments the value of (i) by one.

b. Searching phase:

1. If the value of (i) is greater than (0); this means the pattern's first character occurs in the text. So, go to step (2), otherwise; the pattern cannot be found in the text at all, so go to step (6).

2. Create an integer variable (*c*), which starts from (0) and reaches the value (*i*-1), and incremented by one.
3. If the value of the variable (*c*) is less than (*i*); go to step (4). Otherwise, go to step (6).
4. Scan the sub-string of the text starting from the index (Occurrence_List(*c*)+1) to ((Occurrence_List(*c*) + *m*-1), which represents the size of the pattern, and compare each character of this sub-string with the corresponding character in the pattern. If all characters are matched, then this is an occurrence of the pattern in the text at index (Occurrence_List(*c*)).
5. Increment the value of (*c*) by one and go to step (3).
6. Exit.

In step (1) of the searching phase, if the value of (*i*) is equal to (0), then this means that the first character of the pattern does not occur in the text, and there is no need to search for the pattern. In step (2), the value of the variable (*c*) must not exceed the value (*i*-1), which is the number of the occurrences of the pattern's first character in the text. Example 4.1 illustrates the work of the FC-RJ algorithm.

4.1.1. Pseudocode of FC-RJ Algorithm

The pseudocode of the preprocessing phase of FC-RJ algorithm is expressed as follows:

```
procedure PRE-FC-RJ(array T[n],array P[m])
    var j:=i:=0 as integer
    Create array: Occurrence_List[n-m+1]
    for j from 0 to n-m do
      if T(j) == P(0) then
        Occurrence_List(i):= j
        i:= i + 1
    SEARCH-FC-RJ(T[n], P[m], i, Occurrence_List[n-m+1])
end procedure
```

The pseudocode of the searching phase of the FC-RJ algorithm is as follows:

```
procedure SEARCH-FC-RJ(array T[n],array P[m], i,
                             array Occurrence_List[n-m+1])
   if i > 0 then
     if m==1 then output the content of Occurrence_List()
     else
       var c:=x:=0, count:=1, as integer
       var value as Boolean
       while c < i do
         value:= true
         for x from Occurrence_List(c)+1
                  to  Occurrence_List(c)+ m-1 do
```

```
          if T(x)≠P(count) then
            value:= false
            break the for loop
          count:= count+1
        if value==true then
          output(Occurrence_List(c))
        c:=c+1
        count:=1
  else output("The pattern is not found!")
end procedure
```

Example 4.1: A single pattern matching example using FC-RJ algorithm:

For simplicity, assume that we have the following text and pattern, and we want to find all occurrences of the pattern in the text:

Text

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	M	A	C	C	O	A	M	B	A	C	H	A	M	A	B	C	O	A	M	A	L	C	O

Pattern

0	1	2	3	4	5
A	M	A	B	C	O

Then the algorithm creates the Occurrence_List to save the indices of the text's characters that equal the pattern's first character, which is (A) in this example. The algorithm searches for the first character of the pattern in the range of indices from (0) to (n-m=24-6=18) of the text, because what is left is less than the length of the pattern and it is impossible to the pattern to occur after index (18) in the text. The Occurrence_List will be as follows:

Occurrence_List

0	1	2	3	4	5	6
0	2	6	9	12	14	18

In the searching phase, the algorithm will make 7 matching attempts to search for the pattern in the text using the elements values of the Occurrence_List as indices, as follows:

First attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
	1	2	3																				
A	M	A	B	C	O																		

Mismatch, go to index 2.

Second attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
			1																				
		A	M	A	B	C	O																

Mismatch, go to index 6.

Third attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
							1	2															
						A	M	A	B	C	O												

Mismatch, go to index 9.

Fourth attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
										1	2												
									A	M	A	B	C	O									

Mismatch, go to index 12.

Fifth attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
													1	2	3	4	5						
												A	M	A	B	C	O						

An occurrence of the pattern at index 12. Go to index 14.

Sixth attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
															1								
														A	M	A	B	C	O				

Mismatch, go to index 18.

Seventh attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
																			1	2	3		
																		A	M	A	B	C	O

Mismatch.

The algorithm performed (17) character comparisons in the example.

4.1.2. Analysis of FC-RJ Algorithm

The preprocessing phase of FC-RJ algorithm is concerned with determining and saving the indices of the text segments that represent the expected occurrences of the pattern. These indices are saved in the Occurrence_List array of size (***i***). This variable represents the number of the expected occurrences of the pattern in the text, which is, at most, equals to (*n*-*m*+1), where *n* is the length of the text and *m* is the length of the pattern.

To do so, the preprocessing phase scans the first (*n*-*m*) characters of the text. The worst case of the preprocessing phase arises when the pattern consists of only one character, since it will check the occurrences of the first character of the pattern in the whole text in such case. Thus, it is linear in **O(*n*)** time in the worst and average cases. In the best case, when the input text and pattern are of the same length, the preprocessing phase will compare the first character of the pattern with the first character of the text, and if a mismatch occurred, it will stop searching and the algorithm will finish. Therefore, the preprocessing phase of the FC-RJ algorithm takes **O(1)** time in the best case.

The searching phase uses the Occurrence_List array to move to the indices of the text that represent expected occurrences of the pattern using the variable *x*, which starts with the value (0) and ends with (*i*-1), where *i* is the number of expected occurrences of the pattern in the text.

The best case of the searching phase of FC-RJ algorithm arises when the variable *i* equals to zero. In other words, when there are no occurrences of the pattern in the text, in this case, the time complexity of the searching phase of FC-RJ algorithm is **O(1)**.

The number of places (indices in the text) that the algorithm starts searching at (*i*) represents the number of expected occurrences of the pattern in the text. At each x^{th} index in the text, the searching phase tries to match the segment (*x*+1…*x*+*m*-1) of the text with the pattern, character by character. So, the algorithm compares *m*-1 characters at each x^{th} index, until it reaches the $(i\text{-}1)^{th}$ element of the Occurrence_List array. This means, it takes (*i*)*(*m*-1) time. Thus, the searching phase takes **O((*i*****m*)-*i*)** time in the worst case of FC-RJ algorithm, where ***i*** is the number of expected occurrences of the pattern in the text, and *m* is the length of the pattern. In terms of *n*, when *i* equals to (*n*-*m*+1), the worst case time complexity of the searching phase is **O(*n*****m*)**. The algorithm performs at most (*im*)-*i* text character comparisons during the searching phase.

The preprocessing phase of the FC-RJ algorithm searches the first *n*-*m* portion of the text for the expected occurrences of the pattern. Therefore, FC-RJ algorithm requires **O(*n*)** extra space for the Occurrence_List array, in addition to the original text and pattern. If the size of the Occurrence_List array (*i*) is specified dynamically; the preprocessing phase will require ***i*** additional space instead of *n*-*m*+1.

4.2. FLC-RJ Algorithm

The concept of the First and Last Characters-Rami and Jehad (FLC-RJ) algorithm follows the concept of FC-RJ algorithm. It seems more efficient to attempt matching the pattern only with the sub-strings of the text that start with the pattern's first character and also end with the pattern's last character.

This technique decreases the number of character comparisons in the text. It can be achieved by simply adding another condition (restriction) in the preprocessing phase of FC-RJ algorithm.

Because this algorithm searches for the first and last characters of the pattern in the text; it requires that the pattern to be of length more than one character to work efficiently. If the pattern consists of only one character, then this character will be considered as the first and the last character of the pattern and it will be compared twice instead of one time at each comparison operation with the text characters. To avoid occurring of this case, the algorithm behaves as FC-RJ algorithm in such case. In other words, if the pattern consists of only one character; FLC-RJ algorithm will search the text only for the first character of the pattern, and it will behave exactly as FC-RJ algorithm.

4.2.1. Pseudocode of FLC-RJ Algorithm

The pseudocode of the preprocessing phase of FLC-RJ algorithm is as follows:

```
procedure PRE-FLC-RJ(array T[n],array P[m])
   var j:=i:=0 as integer
   Create array: Occurrence_List[n-m+1]
   if m > 1 then
     for j from 0 to n-m do
       if T(j)==P(0) AND T(j+m-1)==P(m-1) then
         Occurrence_List(i):= j
         i:= i + 1
   else
     for j from 0 to n-m do
       if T(j) == P(0) then
         Occurrence_List(i):= j
         i:= i + 1
   SEARCH-FLC-RJ(T[n], P[m], i, Occurrence_List[n-m+1])
end procedure
```

The searching phase should not compare the characters that are already matched during the preprocessing phase. This implies that, the first and the last characters of each segment in the text will not be compared with the characters of the pattern during the searching phase, since they have been matched during the preprocessing phase and there is no need to be compared again.

The searching phase of FLC-RJ algorithm is as follows:

```
procedure SEARCH-FLC-RJ(array T[n],array P[m], i,
                         array Occurrence_List[n-m+1])
  if i > 0 then
    if m==1 then output the content of Occurrence_List()
    else
      var c:=x:=0, count:=1, as integer
      var value as Boolean
      while c < i do
        value:= true
        for x from Occurrence_List(c)+1
               to  Occurrence_List(c)+m-2 do
          if T(x)≠P(count) then
            value:= false
            break the for loop
          count:= count+1
        if value==true then
          output(Occurrence_List(c))
        c:=c+1
        count:=1
  else output("The pattern is not found!")
end procedure
```

Example 4.2: A single pattern matching example using FLC-RJ algorithm:

Assume that the same text and pattern of Example 4.1 are used in this example utilizing FLC-RJ algorithm.

Text

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	M	A	C	C	O	A	M	B	A	C	H	A	M	A	B	C	O	A	M	A	L	C	O

Pattern

0	1	2	3	4	5
A	M	A	B	C	O

Then the preprocessing phase will determine the indices of the expected occurrences of the pattern in the text by comparing the first character of the pattern, which is (A) in this example, with the first *n-m* characters of the text, since the pattern cannot be occurred after the first *n-m* characters in the text, using a variable *j*. If the current *j*th character in the text is matched with the pattern's first character; the last character of the pattern will be compared with the (*j*+m-1)th character of the text, since the segment (*j*…*j*+*m*-1) represents the length of the pattern (*m*). If the first and the last characters of a segment in the text equal the first and the last characters of the pattern respectively, then this segment will be considered as an expected occurrence of the pattern in the text, and the index of this segment in the text will be saved in the Occurrence_List array.

The Occurrence_List of this example will be as follows:

Occurrence_List

0	1	2
0	12	18

In the searching phase, the algorithm makes 3 matching attempts to search for the pattern in the text using the elements values of the Occurrence_List array as indices, as follows:

First attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
	1	2	3																				
A	M	A	B	C	O																		

Mismatch, go to index 12.

Second attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
													1	2	3	4							
												A	M	A	B	C	O						

An occurrence of the pattern at index 12. Go to index 18.

Third attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
																			1	2	3		
																		A	M	A	B	C	O

Mismatch.

The algorithm performed (10) character comparisons in the example.

As shown in this example, it is clear that FLC-RJ algorithm decreases the number of character comparisons as compared to FC-RJ algorithm, because comparing the first and last characters of the pattern eliminates some of the mismatches even before the searching phase started.

4.2.2. Analysis of FLC-RJ Algorithm

The preprocessing phase scans the first (*n*-*m*) characters of the text to determine the expected occurrences of the pattern in the text. As the preprocessing phase of the FC-RJ algorithm, the worst case of the preprocessing phase of the FLC-RJ algorithm arises when the pattern consists of only two characters, since it will check the occurrences of the first character of the pattern in the whole text except the last character in such case. Thus, it is linear in **O(*n*)** time in the worst and average cases. In the best case, when the input text and pattern are of the same length, the preprocessing phase will compare the first character of the pattern with the first character of the text, and if a mismatch occurred, it will stop searching and the algorithm will finish. Therefore, the preprocessing phase of the FLC-RJ algorithm takes **O(1)** time in the best case.

The searching phase uses the Occurrence_List array to reach the indices of the text that represent the expected occurrences of the pattern using the variable (*x*), which starts with the value (0) and ends with (*i*-1).

The best case of the searching phase of FLC-RJ algorithm arises when the variable (*i*) equals to zero. In other words, when there are no occurrences of the pattern in the text, in this case, the time complexity of the searching phase of FLC-RJ algorithm is **O(1)**.

The FLC-RJ algorithm uses the Occurrence_List to search the text for the pattern. The number of places that the algorithm starts searching at is (*i*), which represents the number of expected occurrences of the pattern in the text. At each x^{th} index in the text, the searching phase tries to match the segment (x+1…x+m-2) of the text with the pattern, character by character. It does not compare the first and the last character of the pattern and the text's segments; since they already have been matched during the preprocessing phase. So, the algorithm compares *m*-2 characters at each x^{th} index, until it reaches the (i-1)th element of the Occurrence_List array. This means, it takes (*i*)*(*m*-2) time. Thus, the searching phase takes **O((*i***m*)-2*i*)** time in the worst case of FLC-RJ algorithm, where ***i*** is the number of expected occurrences of the pattern in the text, and *m* is the length of the pattern. In terms of *n*, when *i* equals to (*n*-*m*+1), the worst case time complexity of the searching phase is **O(*n***m*)**. The algorithm performs at most (*im*)-2*i* text character comparisons during the searching phase.

The preprocessing phase of the FLC-RJ algorithm searches the first *n*-*m* portion of the text for the expected occurrences of the pattern. Therefore, FLC-RJ algorithm requires **O(*n*)** extra space for the Occurrence_List array, in addition to the original text and pattern. If the size of the Occurrence_List array (*i*) is specified dynamically; the preprocessing phase will require ***i*** additional space instead of *n*-*m*+1.

4.3. FMLC-RJ Algorithm

First, Middle, and Last Characters-Rami and Jehad (FMLC-RJ) algorithm adds another restriction to a sub-string of the text to be considered as an expected occurrence of the pattern. It seems more efficient to attempt matching the pattern only with the sub-strings of the text that start with the pattern's first character and end with the pattern's last character, and at the same time, they have middle characters equal the pattern's middle character.

This technique decreases the number of character comparisons in the text during the searching phase. It can be achieved by adding another condition in the preprocessing phase of FLC-RJ algorithm.

This algorithm requires the pattern to be of length more than two characters to work efficiently. Moreover, it should cover the case when the pattern consists of only one or two characters ($m < 3$). The preprocessing phase of the FMLC-RJ algorithm behaves as the preprocessing phase of the FLC-RJ algorithm if the length of the pattern is two characters and as the preprocessing phase of the FC-RJ algorithm if it is only one character.

The position of the middle character can be determined by getting the floor value of dividing the length of the pattern over two. Thus, the index of the middle character in the pattern is defined as: ***mid* = *floor* (*m*/2)**.

4.3.1. Pseudocode of FMLC-RJ Algorithm

The preprocessing phase of FMLC-RJ algorithm can be expressed as follows:

```
procedure PRE-FMLC-RJ(array T[n],array P[m])
  var j:=i:=0, mid:= floor(m/2) as integer
  Create array: Occurrence_List[n-m+1]
  if m > 2 then
    for j from 0 to n-m do
      if T(j)==P(0)AND T(j+mid)==P(mid)AND T(j+m-1)==P(m-1) then
        Occurrence_List(i):= j
        i:= i + 1
  else
    if m > 1 then
      for j from 0 to n-m do
        if T(j)==P(0) AND T(j+m-1)==P(m-1) then
          Occurrence_List(i):= j
          i:= i + 1
    else
      for j from 0 to n-m do
        if T(j) == P(0) then
          Occurrence_List(i):= j
          i:= i + 1
  SEARCH-FMLC-RJ(T[n], P[m], i, Occurrence_List[n-m+1])
end procedure
```

The searching phase compares the characters of the pattern with the characters of each expected occurrence except the first, middle, and last characters, which have been matched in the preprocessing phase.

The searching phase of FMLC-RJ can be illustrated as follows:

```
procedure SEARCH-FMLC-RJ(array T[n],array P[m], i,
                                       array Occurrence_List[n-m+1])
  if i > 0 then
    if m==1 then output the content of Occurrence_List()
    else
      var c:=x:=0, count:=1, as integer
      var value as Boolean
      while c < i do
        value:= true
        for x from Occurrence_List(c)+1
              to  Occurrence_List(c)+ m-2 do
          if count==mid then increment x and count by 1
          if T(x)≠P(count) then
            value:= false
            break the for loop
          count:= count+1
        if value==true then
          output(Occurrence_List(c))
        c:=c+1
        count:=1
  else output("The pattern is not found!")
end procedure
```

Example 4.3: A single pattern matching example using FMLC-RJ algorithm:

Suppose that the same text and pattern of Example 4.1 are used in this example using FMLC-RJ algorithm, as follows:

Text

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	M	A	C	C	O	A	M	B	A	C	H	A	M	A	B	C	O	A	M	A	L	C	O

Pattern

0	1	2	3	4	5
A	M	A	B	C	O

The preprocessing phase searches for the segments of the text where the first, middle, and last characters of these segments equal the first, middle, and last characters of the pattern respectively.

Since this condition only achieved at index (12) in the text of this example, then the Occurrence_List will be consisting of one element, as follows:

Occurrence_List

0
12

This means, there is only one expected occurrence of the pattern at index (12) of the text. In the searching phase, the algorithm will make only one matching attempt, instead of (7) attempts of FC-RJ algorithm and (3) attempts of FLC-RJ algorithm, to search for the same pattern in the same text using the elements values of the Occurrence_List as indices, as follows:

First attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
													1	2		3							
												A	M	A	B	C	O						

An occurrence of the pattern at index 12.

The algorithm performed (3) character comparisons in this example.

In Example 4.3, it is clear that FMLC-RJ algorithm decreased the number of character comparisons of both, FC-RJ and FLC-RJ algorithms, since this algorithm performed only 3 character comparisons while FC-RJ performed 17, and FLC-RJ performed 10 character comparisons to search for the same pattern in the same text used in the three algorithms.

4.3.2. Analysis of FMLC-RJ Algorithm

The preprocessing phase of the FMLC-RJ algorithm scans the first (*n*-*m*) portion of the text to determine the expected occurrences of the pattern in the text. As the preprocessing phase of the FC-RJ and FLC-RJ algorithms, the worst case of the preprocessing phase of the FMLC-RJ algorithm arises when the pattern consists of only three characters, since it will check the occurrences of the first character of the pattern in the whole text except the last two characters in such case. Thus, it is linear in **O(*n*)** time in the worst and average cases. In the best case, when the input text and pattern are of the same length, the preprocessing phase will compare the first character of the pattern with the first character of the text, and if a mismatch occurred, it will stop searching and the algorithm will finish. Therefore, the preprocessing phase of the FMLC-RJ algorithm takes **O(1)** time in the best case.

The best case of the searching phase of FMLC-RJ algorithm arises when the variable (*i*) equals to zero. In other words, when there is no any expected occurrence of the pattern in the text, the best case time complexity of the searching phase of FMLC-RJ algorithm is **O(1)**.

The FMLC-RJ algorithm uses the Occurrence_List to search the text for the pattern. The number of places that the algorithm starts searching at is (i), which represents the number of expected occurrences of the pattern in the text. At each xth index in the text, the searching phase tries to match the segment (x+1…x+m-2), except the middle character, of the text's segments with the pattern, character by character. It does not compare the first, middle, and last characters of the pattern with those of the text's segments; since they already have been matched during the preprocessing phase. So, the algorithm compares m-3 characters at each xth index, until it reaches the (i-1)th element of the Occurrence_List array. This means, it takes (i)*(m-2) time. Thus, the searching phase takes **O((*i**m*)-2*i*)** time in the worst case of FMLC-RJ algorithm, where ***i*** is the number of expected occurrences of the pattern in the text, and m is the length of the pattern. In terms of n, when i equals to (n-m+1), the worst case time complexity of the searching phase is **O(*n**m*)**. The algorithm performs at most (im)-3i text character comparisons during the searching phase.

The preprocessing phase of the FMLC-RJ algorithm searches the first n-m portion of the text for the expected occurrences of the pattern. Thus, FMLC-RJ algorithm requires **O(*n*)** extra space for the Occurrence_List array, in addition to the original text and pattern. If the size of the Occurrence_List array (i) is specified dynamically; the preprocessing phase will require ***i*** additional space instead of n-m+1.

4.4. ASCII-Based-RJ Algorithm

The preprocessing phase of the ASCII-Based-Rami and Jehad (ASCII-Based-RJ) algorithm finds the indices of the characters in the text that do not occur in the pattern. Assume that a character at index (z) in the text does not occur in the pattern, then the pattern cannot start at any position in the segment (z-m+1…z) of the text, where m is the length of the pattern. Thus, this segment, and all such segments, will be excluded during searching for the first character of the pattern in the text.

The preprocessing phase creates a zero-based array called `ASCII_Arr` of size (95) elements (indexed from 0 to 94). This size represents the number of the printable characters in the **ASCII** table (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). These characters are from (space), which has the code (32), to (~), which has the code (126), in the ASCII table.

The algorithm scans the characters of the pattern and for each character it increments the value of the element in the `ASCII_Arr` array using the actual ASCII code for that character minus (32) as index. For example, if the character (m), which has the code (109) in the ASCII table, occurs in the pattern, then the element in the `ASCII_Arr` array at index (77), (109-32=77), will be incremented by one. Therefore, the index of the (space) is (0), and the index of (~) is (94) in `ASCII_Arr` array.

After that, the algorithm creates a new array, called `SKIP_Arr`, to hold the indices of the text that the pattern cannot start occurring at. These indices are determined by scanning the text's characters from right to left, and the segment (z-m+1…z) for each index (z) in the text that contains a character does not appear in the pattern will be ignored during searching for the pattern's first character in the text. The range from (z-m+1) to (z) represents (m), which is the length of the pattern.

The algorithm determines whether a character in the text occurs in the pattern or not by checking the corresponding element in the `ASCII_Arr` array (ASCII code of that character minus 32). If the value of the element in that index is zero, then this character (z) of the text did not appear in the pattern. Thus, the segment (z-m+1…z) in the `SKIP_Arr` array will hold the value (-1) to denote that this segment will be ignored during searching for the pattern's first character.

At this stage, the algorithm checks the elements of the `SKIP_Arr` array to search for the occurrences of the pattern's first character in the text. If the value of the element is 0 (the initial value), then it checks the text at index equal to the current index of the `SKIP_Arr` array, and if the character of that index in the text is equal to the pattern's first character, the index will be saved in the Occurrence_List using a variable (i). The element will be skipped if it is (-1).

This technique decreases the number of text character comparisons. Furthermore, it decreases the number of expected occurrences of the pattern in the text. As a result, it decreases the time complexity of searching for a pattern in a text.

4.4.1. Pseudocode of ASCII-Based-RJ Algorithm

The pseudocode of the preprocessing phase is expressed as follows:

```
procedure PRE-ASCII-BASED(array T[n],array P[m])
1   var j:=i:=y:=z:=0, x:=n-1 as integer
2   Create array: ASCII_Arr[95] initialized by 0's
3   Create array: SKIP_Arr[n] initialized by 0's
4   Create array: Occurrence_List[n-m+1] initialized by 0's
5   for j from 0 to m-1 do
6     Increment ASCII_Arr(ASCII_CODE(P(j))-32)
7   for x from n-1 downto 0 do
8     if ASCII_Arr(ASCII_CODE(T(x))-32)==0 AND x >= m-1 then
9       for y from x-m+1 to x do
10        if SKIP_Arr(y) == 0 then
11          SKIP_Arr(y):=-1
12        else
13          Break the loop
14    else
15    if ASCII_Arr(ASCII_CODE(T(x))-32)==0 AND x < m-1 then
16      for y from 0 to x do
17        if SKIP_Arr(y) == 0 then
18          SKIP_Arr(y):=-1
19        else
20          Break the loop
21  for z from 0 to n-m do
22    if SKIP_Arr(z) ≠ -1 AND T(z)==P(0) then
23      Occurrence_List(i):= z
24      i:=i+1
25  SEARCH-ASCII-BASED(T[n], P[m],i, Occurrence_List[n-m+1])
end procedure
```

The searching phase of ASCII-Based-RJ Algorithm is as follows:

```
procedure SEARCH-ASCII-BASED(array T[n], array P[m], i,
                              array Occurrence_List[n-m+1])
1   if i > 0 then
2     if m==1 then output the content of Occurrence_List()
3     else
4       var c:=x:=0, count:=1, as integer
5       var value as Boolean
6       while c < i do
7         value:= true
8         for x from Occurrence_List(c)+1
                 to  Occurrence_List(c)+ m-1 do
9           if T(x)≠P(count) then
10            value:= false
11            break the for loop
12          count:= count+1
13        if value==true then
14          output(Occurrence_List(c))
15        c:=c+1
16        count:=1
17  else output("The pattern is not found!")
end procedure
```

Example 4.4: A single pattern matching example using ASCII-Based-RJ algorithm:

Assume that the same text and pattern of Example 4.1 are used in this example as follows:

Text

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	M	A	C	C	O	A	M	B	A	C	H	A	M	A	B	C	O	A	M	A	L	C	O

Pattern

0	1	2	3	4	5
A	M	A	B	C	O

Step 1: The algorithm scans the characters of the pattern, and for each character in the pattern it increments the value of the ASCII_Arr by one at index equals to the code of that character in the ASCII table minus (32).

In other words, the value of the corresponding element in the ASCII_Arr of each character in the pattern will be incremented by one, using the ASCII code of that character minus (32) as index in the ASCII_Arr.

Table 4.1 shows the ASCII_Arr array after incrementing the corresponding elements of each character in the pattern. Actually, the ASCII_Arr is of length (95), indexed from (0) to (94). But for simplicity, we have shown only the indices of the characters of the pattern in Table 4.1, while the rest of indices will contain (0), which is the initial value of the elements of the ASCII_Arr.

As shown before, for each character in the text, there is a corresponding character in the ASCII_Arr at index equals to the ASCII code of that character minus (32).

Table 4.1: The values of `ASCII_Arr` array (indexed from 0 to 94)

Character	**Index in `ASCII_Arr` (ASCII Code-32)**	**Frequency**
A	65 – 32 = **33**	2
M	77 – 32 = **45**	1
B	66 – 32 = **34**	1
C	67 – 32 = **35**	1
O	79 – 32 = **47**	1

Step 2: The algorithm scans the characters of the text from right to left starting at the last character. If the corresponding value of the current character (z) of the text in the ASCII_Arr is zero, then this means that this character (in the text) does not occur in the pattern. Thus, the range from (z-m+1…z) in the text will be ignored during the searching phase, since the pattern cannot starts at any index of this range in the text.

The algorithm saves this range in the SKIP_Arr, which is of length *n*, where *n* is the length of the text. The algorithm overwrites the values of the elements of indices of the range (*z*-*m*+1…*z*) in the SKIP_Arr by the value (-1) to be skipped during searching for the first character of the pattern. The `SKIP_Arr` array is shown in Table 4.2. Note that, the indices that have a gray background will be excluded during the searching phase, since the pattern can not occur in these indices.

Table 4.2: The `SKIP_Arr` array

Index	**Value**	**Index**	**Value**
0	0	12	0
1	0	13	0
2	0	14	0
3	0	15	0
4	0	16	-1
5	0	17	-1
6	-1	18	-1
7	-1	19	-1
8	-1	20	-1
9	-1	21	-1
10	-1	22	0
11	-1	23	0

Since the SKIP_Arr is of length *n*, which is the length of the text, the algorithm uses it to search for the pattern's first character in the text only at indices that have the value (0) in the SKIP_Arr, while ignoring the indices that have the value (-1).

<u>Step 3</u>: The algorithm saves the indices of the text where the pattern's first character occurs in the Occurrence_List array to be used during the searching phase.

The Occurrence_List array will be as follows:

Occurrence_List

0	1	2	3
0	2	12	14

<u>Step 4</u>: The algorithm uses the searching phase of FC-RJ algorithm and makes 4 matching attempts as follows:

First attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
	1	2	3																				
A	M	A	B	C	O																		

Mismatch, go to index 2.

Second attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
			1																				
		A	M	A	B	C	O																

Mismatch, go to index 12.

Third attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
													1	2	3	4	5						
												A	M	A	B	C	O						

An occurrence of the pattern at index 12. Go to index 14.

Fourth attempt:

A	M	A	C	C	O	A	M	B	A	M	H	A	M	A	B	C	O	A	M	A	L	C	O
															1								
														A	M	A	B	C	O				

Mismatch.

The algorithm performed (10) character comparisons in the example.

4.4.2. Analysis of ASCII-Based-RJ Algorithm

The worst case of the preprocessing phase of ASCII-Based-RJ algorithm arises when each character of the text does not occur in the pattern, or, when for each *m* characters in the text, only the last character of that segment does not occur in the pattern. In this case, the lines (from 7 to 14) of the pseudocode of the preprocessing phase of ASCII-Based-RJ algorithm take O(2*n*) time, which is simplified as **O(*n*)**.

The best case of the preprocessing phase of ASCII-Based-RJ algorithm occurs when all characters of the text appear in the pattern. This means, there is no any character exists in the text and does not occur in the pattern. In this case, the preprocessing phase takes **O(*n*)** time.

The preprocessing phase builds the Occurrence_List array, which holds the indices of the expected occurrences of the pattern in the text. During the searching phase, the algorithm uses the Occurrence_List to match the expected occurrences of the pattern in the text with the characters of that pattern.

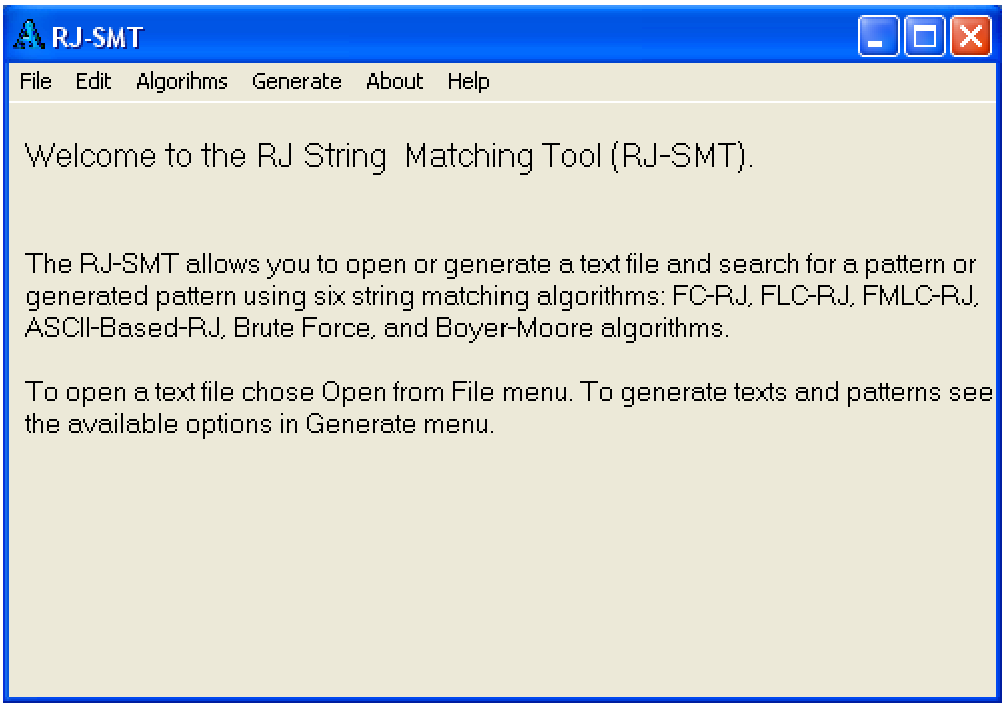
In the best case, when there is no any expected occurrence of the pattern in the text, the searching phase of ASCII-Based-RJ algorithm takes a constant time in **O(1)**. Therefore, the overall time complexity of the algorithm is O(*n*).

In the worst case, the searching phase of the algorithm scans (*m*-1) characters (*i*) times, where (*i*) is the number of expected occurrences of the pattern in the text. Thus, the algorithm takes **O((*i***m*)-*i*)** time in the worst case.

The ASCII-Based-RJ algorithm requires (95) additional space for the ASCII_Arr array, (*n*) space for the SKIP_Arr array, and (*n*-*m*+1) space for the Occurrence_List array. So, it needs (2*n*-*m*+96), which is **O(*n*)** extra space, in addition to the original text and pattern.

Chapter Five: The SMT-RJ Simulator

To compare between the performance of our algorithms with other common algorithms; we have built a simulator, which is referred to as String Matching Tool-Rami and Jehad (SMT-RJ), using Visual Basic 6.0. The simulator represents a string matching tool (a text editor). In this tool, FC-RJ, FLC-RJ, FMLC-RJ, and ASCII-Based-RJ algorithms have been implemented. Figure 5.1 shows the interface of the SMT-RJ.



Welcome to the RJ String Matching Tool (RJ-SMT).

The RJ-SMT allows you to open or generate a text file and search for a pattern or generated pattern using six string matching algorithms: FC-RJ, FLC-RJ, FMLC-RJ, ASCII-Based-RJ, Brute Force, and Boyer-Moore algorithms.

To open a text file chose Open from File menu. To generate texts and patterns see the available options in Generate menu.

Figure 5.1: Interface of the SMT-RJ.

Furthermore, the Naïve (brute force) algorithm as mentioned in (Charras and Lecroq, 2004) and Boyer-Moore algorithm (Boyer and Moore, 1977) have been implemented in the tool to be compared with the proposed algorithms. These algorithms have been selected because they have been shown to perform well in (Boyer and Moore, 1977; Charras and Lecroq, 2004) compared to other existing algorithms.

We have exhaustively tested the implemented algorithms on random test data. To gather the test patterns, we wrote a program which randomly selects a substring of a given length from the source string, see Figure 5.2. As the lengths of the test patterns that Boyer and Moore used in (Boyer and Moore, 1977), and to get a high confidence interval and low error percentage, we used this program to select 300 patterns of length *m* for each *m* from 1 to 14. We then used the implemented algorithms in our tool to search for each of the test patterns in its source string.

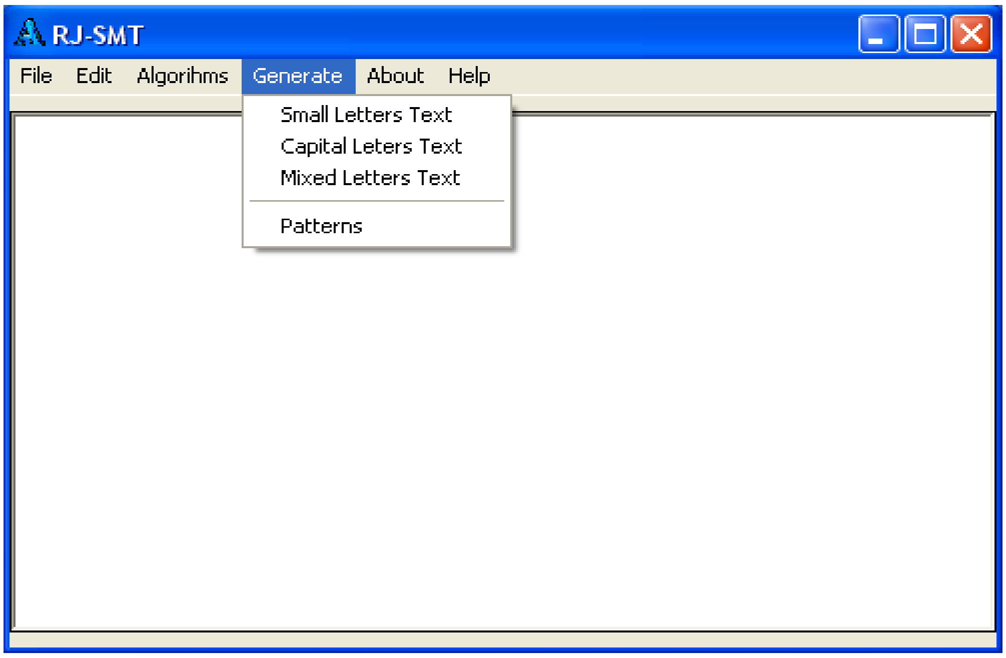
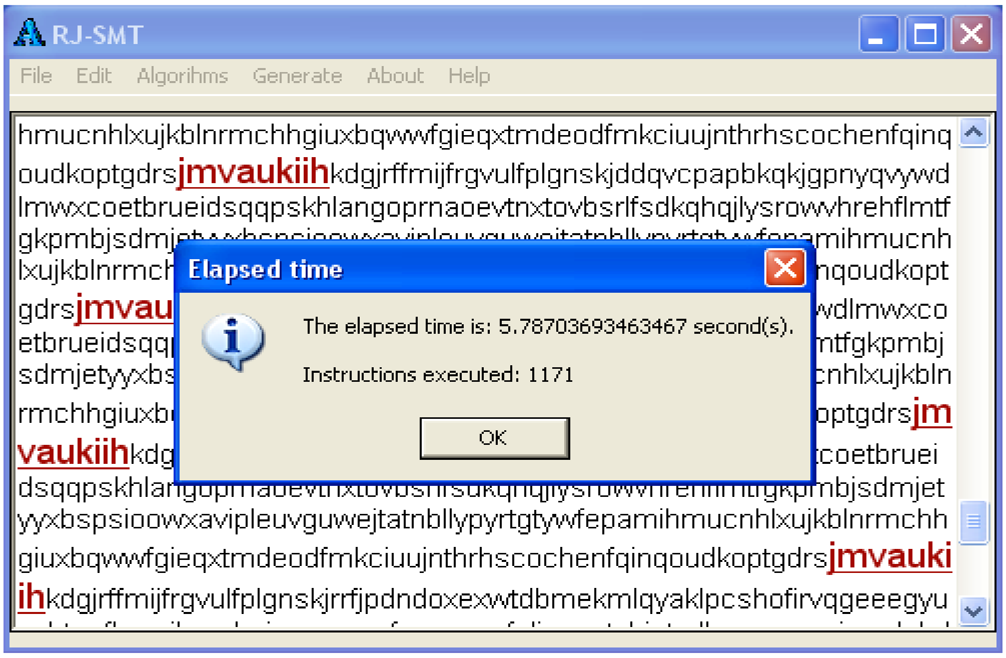


Figure 5.2: Generating random text and patterns.

All of the characters for both the patterns and the text were in the main memory, rather than a secondary storage medium, during running the tool.

We have measured the cost of each implemented algorithm in two ways: The first is the total number of instructions that actually got executed; using an integer variable which incremented after each executed instruction. The second is the execution time in seconds; using a built-in function which gets the execution time in seconds. See Figure 5.3.

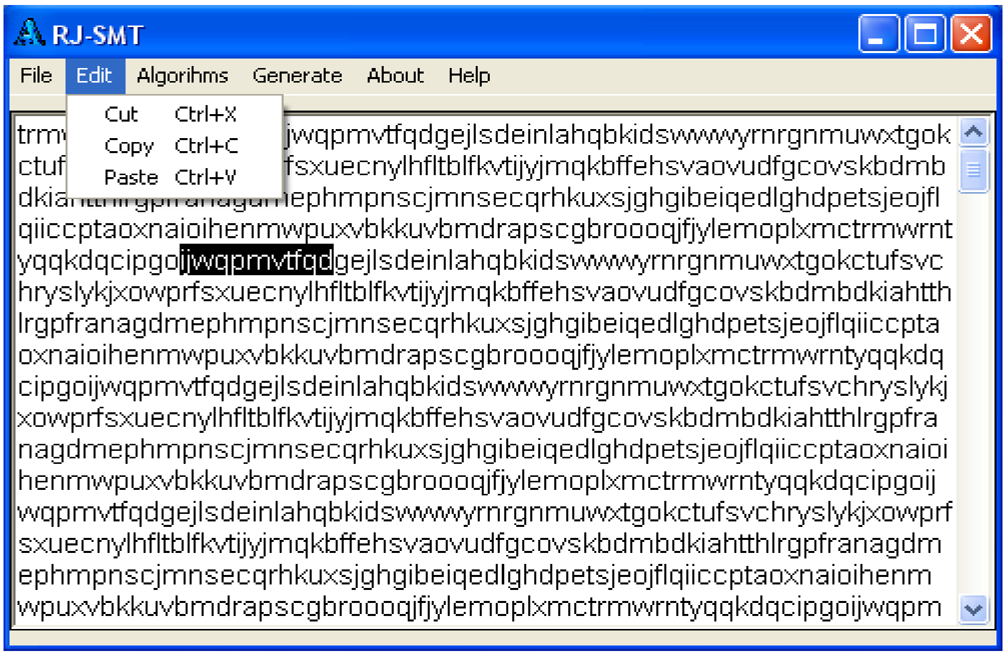


```
The elapsed time is: 5.78703693463467 second(s).

Instructions executed: 1171
```

Figure 5.3: The execution time and the number of executed instructions of an algorithm.

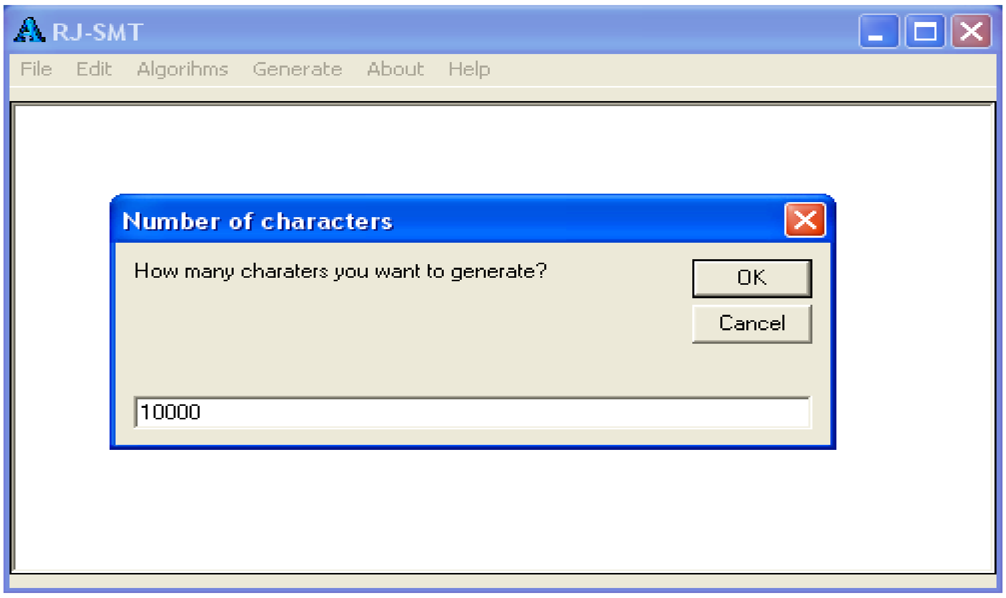
We then averaged these measures across all 300 samples for each pattern length. We have performed these experiments for a string of length 10,000 randomly generated characters. The SMT-RJ is easy-to-use toolkit for string matching. In addition to the tested algorithms, it provides some of standard options that exist in the large text editors, such as: Print, Cut, Copy, Paste, Help, Close, and Exit. See Figure 5.4.



```
trm                 jwqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgok
ctuf                fsxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmb
dkia                ephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojfl
qiiccptaoxnaioihenmwpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrmwrnt
yqqkdqcipgoijwqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctufsvc
hryslykjxowprfsxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdkiahtth
lrgpfranagdmephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiiccpta
oxnaioihenmwpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrmwrntyqqkdq
cipgoijwqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctufsvchryslykj
xowprfsxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdkiahtthlrgpfra
nagdmephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiiccptaoxnaioi
henmwpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrmwrntyqqkdqcipgoij
wqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctufsvchryslykjxowprf
sxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdkiahtthlrgpfranagdm
ephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiiccptaoxnaioihenm
wpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrmwrntyqqkdqcipgoijwqpm
```

Figure 5.4: Edit menu options.

The SMT-RJ allows the user to generate three types of text files: English small letters, English capital letters, and a mixture of English small and capital letters and special symbols, of any length (number of characters) specified by the user. See Figure 5.5.

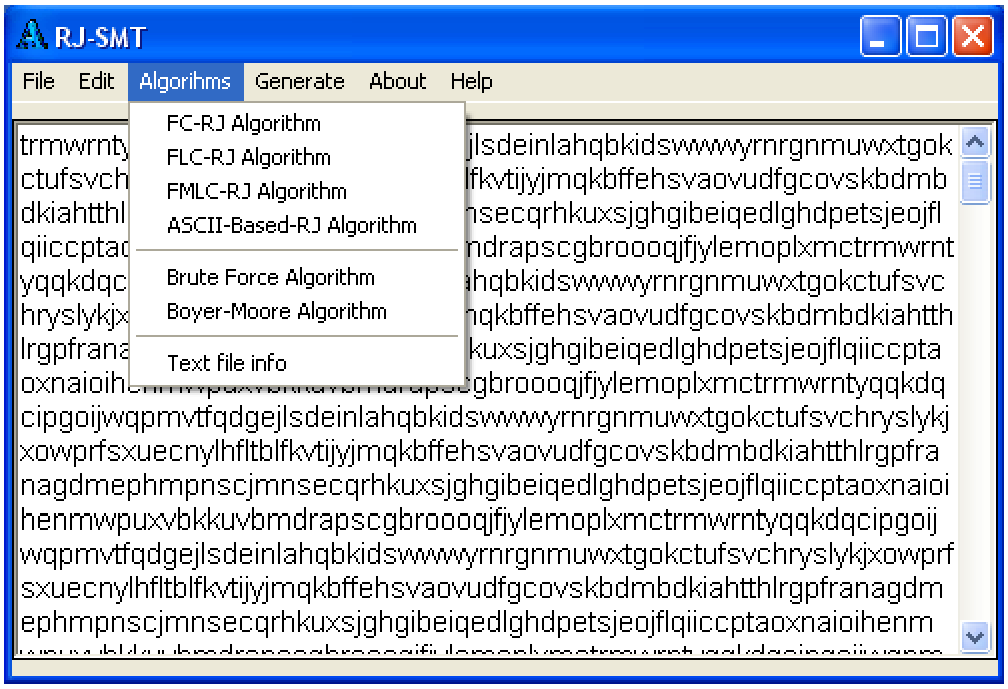


How many charaters you want to generate?

10000

Figure 5.5: Generating text files.

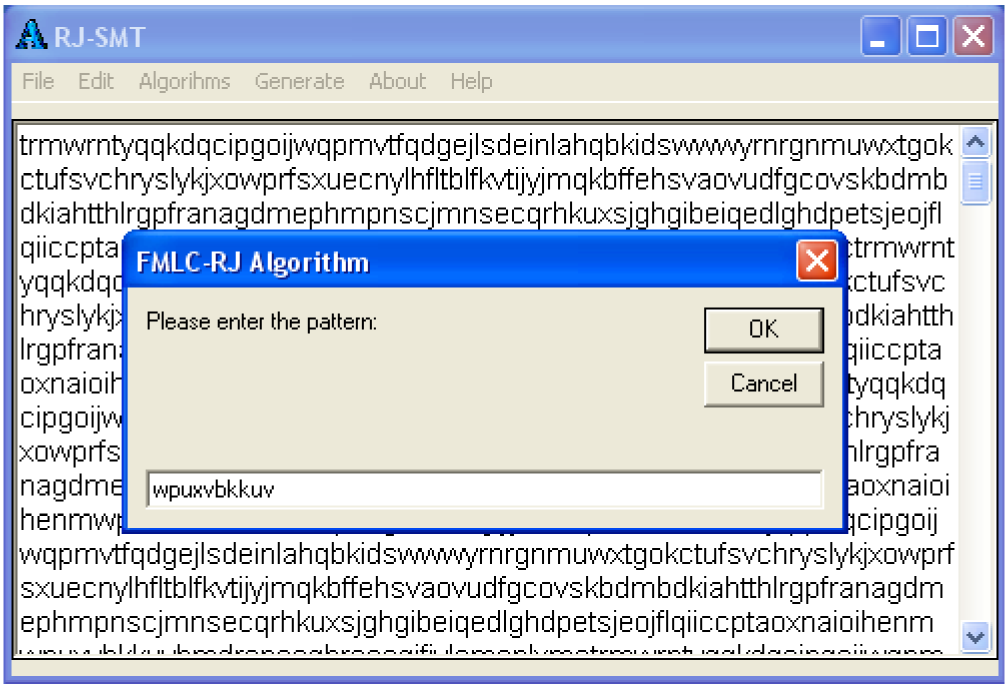
The user is allowed to use any of the implemented algorithms to search for a pattern. See Figure 5.6.



```
trmwrnty                          jlsdeinlahqbkidswwwyrnrgnmuwxtgok
ctufsvch                          lfkvtijyjmqkbffehsvaovudfgcovskbdmb
dkiahtthl                         nsecqrhkuxsjghgibeiqedlghdpetsjeojfl
qiiccptac                         ndrapscgbroooqjfjylemoplxmctrmwrnt
yqqkdqc                           ahqbkidswwwyrnrgnmuwxtgokctufsvc
hryslykjx                         nqkbffehsvaovudfgcovskbdmbdkiahtth
lrgpfrana                         kuxsjghgibeiqedlghdpetsjeojflqiiccpta
oxnaioih                          gbroooqjfjylemoplxmctrmwrntyqqkdq
cipgoijwqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctufsvchryslykj
xowprfsxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdkiahtthlrgpfra
nagdmephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiiccptaoxnaioi
henmwpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrmwrntyqqkdqcipgoij
wqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctufsvchryslykjxowprf
sxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdkiahtthlrgpfranagdm
ephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiiccptaoxnaioihenm
```

Figure 5.6: The implemented algorithms in SMT-RJ.

The system allows the user to insert a pattern and to generate random patterns to be used in the searching process. See Figure 5.7.

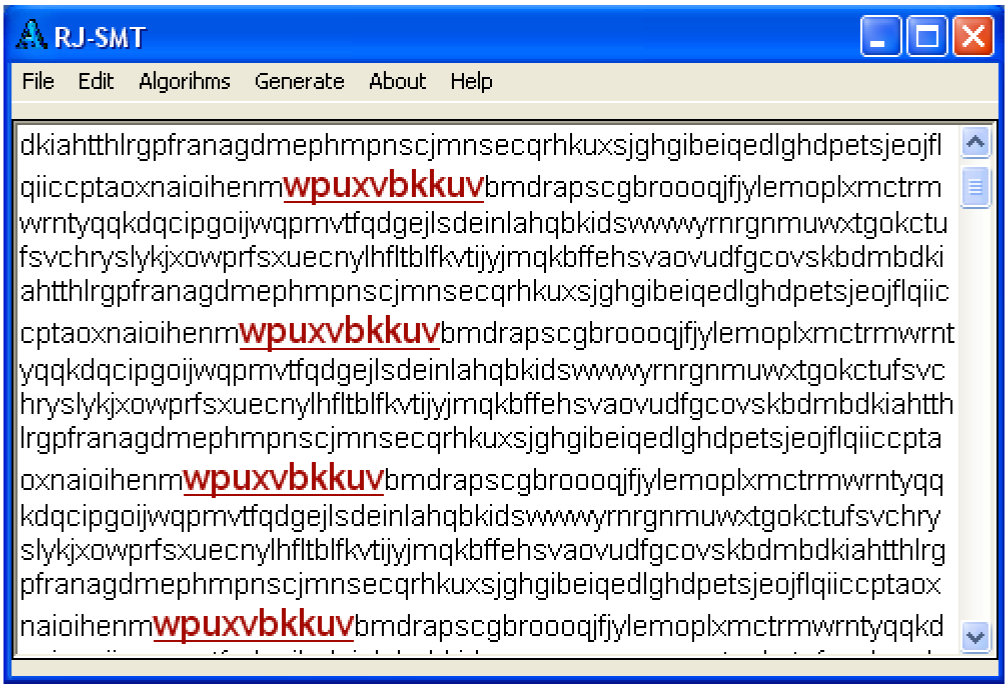


Please enter the pattern: wpuxvbkkuv

```
trmwrntyqqkdqcipgoijwqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgok
ctufsvchryslykjxowprfsxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmb
dkiahtthlrgpfranagdmephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojfl
qiiccpta                                                  ctrmwrnt
yqqkdqc                                                   kctufsvc
hryslykj                                                  bdkiahtth
lrgpfran                                                  qiiccpta
oxnaioih                                                  tyqqkdq
cipgoijw                                                  chryslykj
xowprfs                                                   hlrgpfra
nagdme                                                    aoxnaioi
henmwp                                                    qcipgoij
wqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctufsvchryslykjxowprf
sxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdkiahtthlrgpfranagdm
ephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiiccptaoxnaioihenm
```

Figure 5.7: Inserting a pattern.

If the searched pattern is found, the font color and font size for the pattern will be changed in all places that it has been found in the source text to make sure that the tool gives correct results. See Figure 5.8.



```
dkiahtthlrgpfranagdmephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojfl
qiiccptaoxnaioihenmwpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrm
wrntyqqkdqcipgoijwqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctu
fsvchryslykjxowprfsxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdki
ahtthlrgpfranagdmephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiic
cptaoxnaioihenmwpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrmwrnt
yqqkdqcipgoijwqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctufsvc
hryslykjxowprfsxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdkiahtth
lrgpfranagdmephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiiccpta
oxnaioihenmwpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrmwrntyqq
kdqcipgoijwqpmvtfqdgejlsdeinlahqbkidswwwyrnrgnmuwxtgokctufsvchry
slykjxowprfsxuecnylhfltblfkvtijyjmqkbffehsvaovudfgcovskbdmbdkiahtthlrg
pfranagdmephmpnscjmnsecqrhkuxsjghgibeiqedlghdpetsjeojflqiiccptaox
naioihenmwpuxvbkkuvbmdrapscgbroooqjfjylemoplxmctrmwrntyqqkd
```

Figure 5.8: Changing font color and size of the found pattern.

The main specifications of the computer that the experiments were done on are as follows:

- Processor: Intel Core 2 Due, 2.00 Giga Hertz.
- RAM: DDR2, 3.00 Giga Bytes.
- Cache Memory: 2 Mega Bytes.
- Operating System: Windows Vista Home Premium.

Chapter Six: Results and Discussion

In this chapter, we analyze and discuss the simulation results that we obtained from testing the FC-RJ, FLC-RJ, FML-RJ, ASCII-Based-RJ, Brute Force as mentioned in (Charras and Lecroq, 2004), and Boyer-Moore (Boyer and Moore, 2007) algorithms using the SMT-RJ. We used the same testing data for each algorithm and the experiments were on the same computer and operating system.

Table 6.1 shows the experimental results of the tested algorithms. The execution time in seconds is denoted by (T) while the number of executed instructions is abbreviated by (Inst) for each algorithm.

Table 6.1: Experimental results of the tested algorithms.

Pattern length	FC-RJ		FLC-RJ		FMLC-RJ		ASCII-Based		Brute Force		Boyer-Moore	
	T	Inst	T	Inst	T	Inst	T	Inst	T	Inst	T	Inst
1	2.70	1087	2.90	50	3.00	400	3.45	10100	3.30	30000	3.50	29470
2	3.28	1215	2.87	34	2.90	50	2.91	7000	3.50	30414	3.20	15592
3	3.51	1185	2.95	54	2.50	83	3.11	9991	4.36	30421	3.61	10635
4	3.67	1157	2.98	68	2.71	82	2.80	10000	4.85	30425	3.49	9225
5	3.70	1226	3.10	69	3.10	96	2.90	9975	4.95	30461	3.20	9645
6	4.00	1279	3.70	90	3.50	56	3.20	7000	4.98	30452	3.40	7945
7	4.30	1236	3.50	78	3.30	96	3.00	10009	5.20	30480	3.20	6967
8	4.40	1241	4.00	47	3.80	63	3.20	9996	5.30	30459	3.51	7672
9	4.70	1159	4.30	76	4.10	170	3.60	10000	5.40	30501	3.70	7400
10	4.90	1236	4.50	84	4.30	132	3.70	10012	5.50	30507	3.80	6660
11	5.20	1226	4.80	125	4.50	192	3.80	7300	5.80	30539	4.10	6230
12	5.70	1339	5.20	55	5.00	143	4.10	8400	6.00	30543	4.40	7549
13	5.80	1579	5.50	98	5.20	224	4.20	9900	6.30	30528	4.35	8566
14	6.30	1600	5.70	80	5.40	240	4.40	10016	6.80	30535	4.50	10000

In Table 6.1, the execution time (T) and the number of executed instructions (Inst) of an algorithm represent the average of 300 runs of the algorithm using the same pattern length (*m*) and random characters of the pattern at each run.

Although the FLC-RJ algorithm behaves as FC-RJ algorithm in the case where the pattern is of length only one character, and the FMLC-RJ algorithm behave as FLC-RJ if the pattern was of length two characters; but they consumed more time than the FC-RJ algorithm in the case where the pattern is of length one character. That because, the FLC-RJ and FMLC-RJ algorithms have to check the length of the pattern before they start searching for the first character of the pattern. In other words, the differences in the execution times are because that the FLC-RJ and FMLC-RJ algorithms use an *if*-

statement to check if the length of the pattern is three or more characters, as in the FMLC-RJ algorithm, or if it is two or more characters, as in the FLC-RJ algorithm.

Figure 6.1 shows the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 utilizing FC-RJ algorithm.

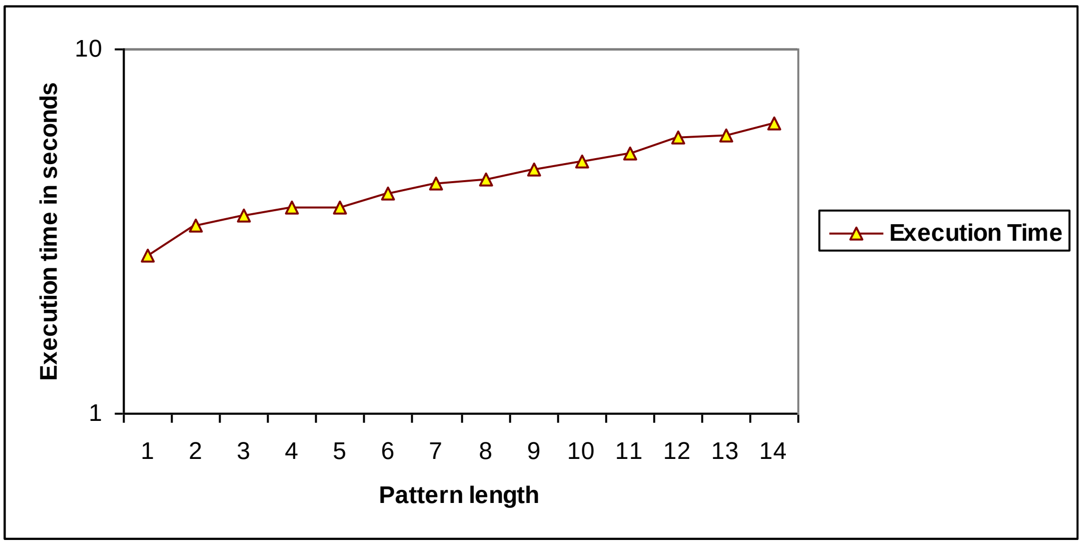


Figure 6.1: Experimental results of FC-RJ algorithm (Execution time in seconds).

It is apparent that the best performance of the FC-RJ algorithm is when the length of the pattern was one character. This result is reasonable, since the algorithm only outputs the content of the Occurrence-List array if the pattern's length is only one character. It is obvious that the execution time increases as the pattern gets longer. Figure 6.2 shows the average number of the executed instructions for each patterns sample of each pattern length from 1 to 14 when the FC-RJ algorithm is used. In the figure, it is clear that the number of executed instructions increases as the pattern gets longer.

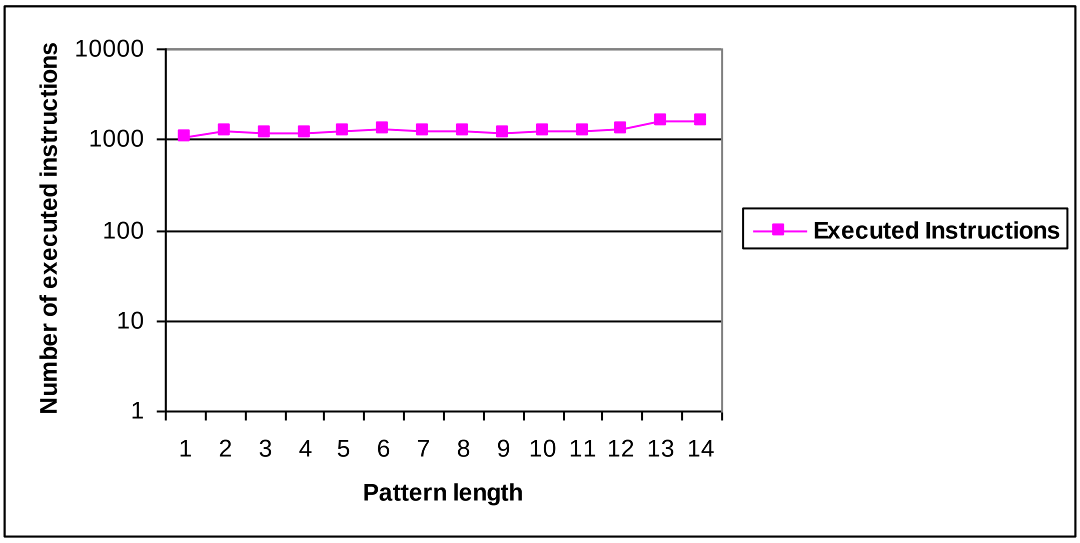


Figure 6.2: Experimental results of FC-RJ algorithm (Number of executed instructions).

Figure 6.3 shows the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 when the FLC-RJ algorithm is used.

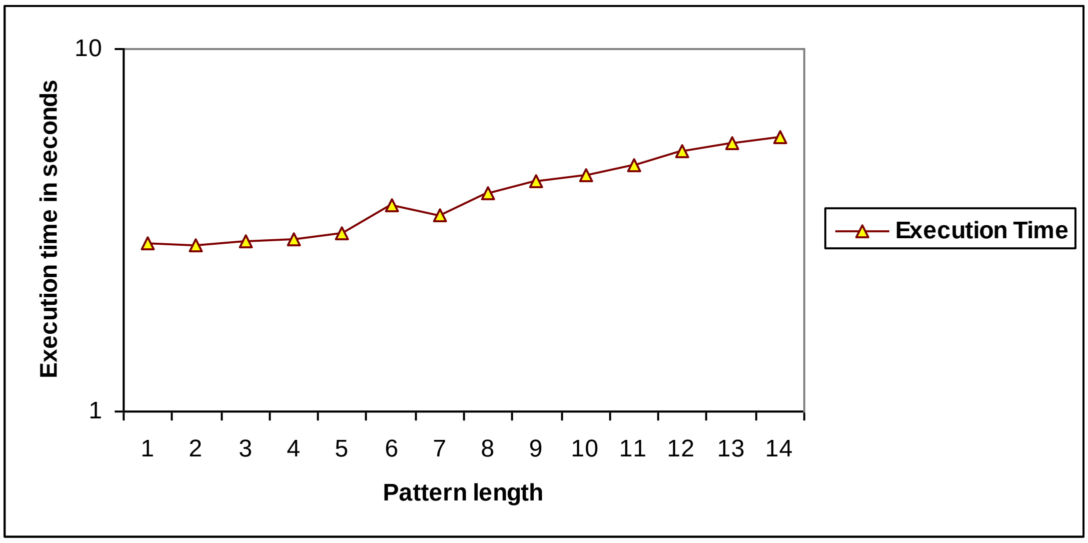


Figure 6.3: Experimental results of FLC-RJ algorithm (Execution time in seconds).

The best performance of the FLC-RJ algorithm is when the length of the pattern was two characters. The reason behind this result is that the algorithm only outputs the content of the Occurrence-List array if the pattern's length is two characters.

Figure 6.4 describes the average number of the executed instructions for each patterns sample of each pattern length from 1 to 14 when the FLC-RJ algorithm is used.

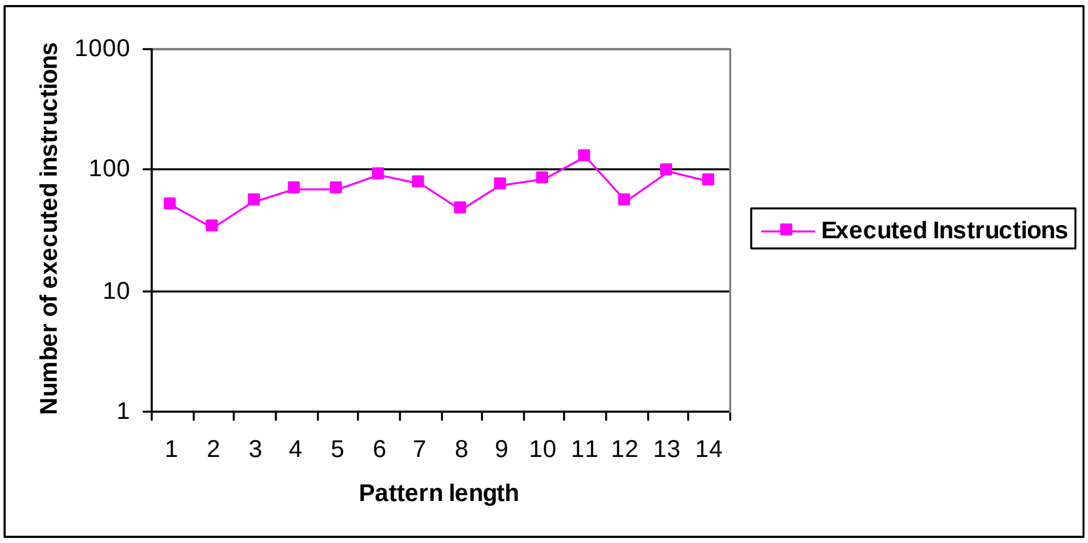


Figure 6.4: Experimental results of FLC-RJ algorithm (Number of executed instructions).

It is clear that the FLC-RJ algorithm executes less number of instructions when the length of the pattern (*m*) is relatively short, especially, when the length of the pattern was only two characters. In this case, the algorithm only outputs the content of the Occurrence_List array.

Figure 6.5 shows the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 utilizing FMLC-RJ algorithm.

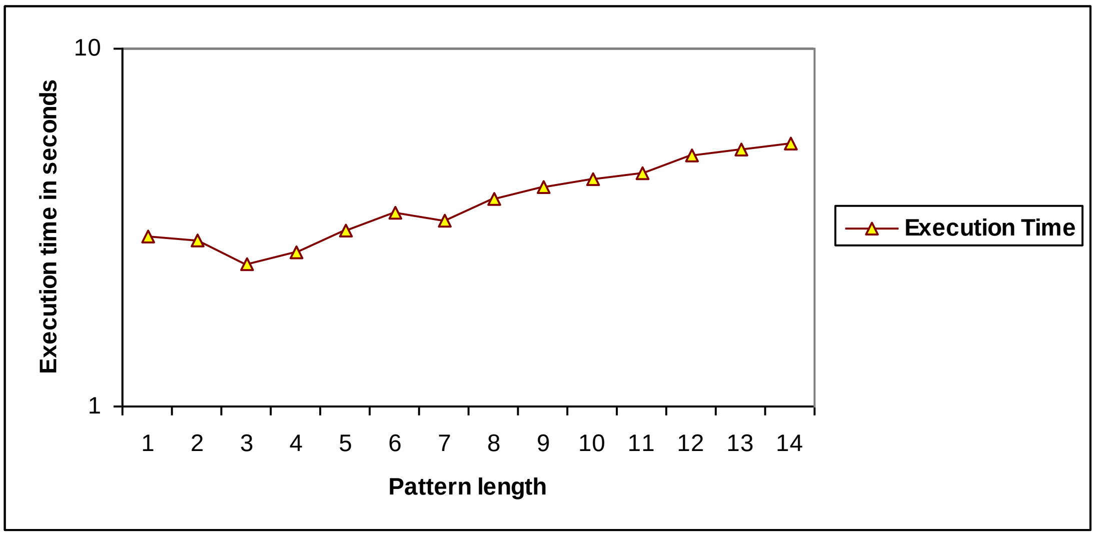


Figure 6.5: Experimental results of FMLC-RJ algorithm (Execution time in seconds).

The best performance of the FMLC-RJ algorithm is when the length of the pattern was three characters. The algorithm searches for the first, middle, and last characters of the pattern and then outputs the content of the Occurrence-List array as a result. The execution time is done in the preprocessing phase in this case.

Figure 6.6 shows the average number of the executed instructions for each patterns sample of each pattern length from 1 to 14 when the FMLC-RJ algorithm is used.

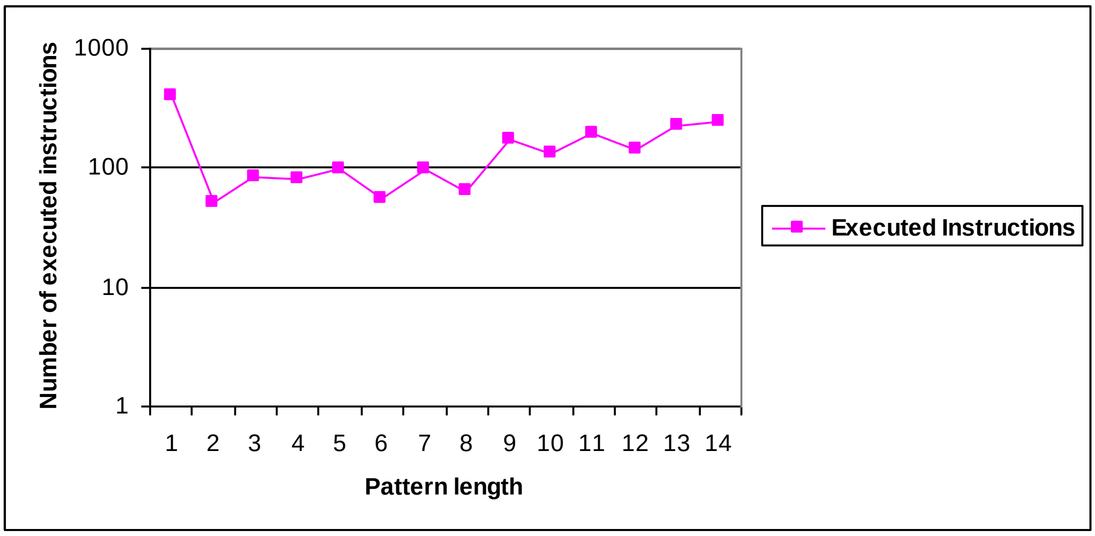


Figure 6.6: Experimental results of FMLC-RJ algorithm (Number of executed instructions).

When the length of the pattern (*m*) was one character, the FMLC-RJ algorithm passes through two checks; the first is if *m* equals (1) then it behaves as the FC-RJ algorithm, and the seconds check is if *m* equals (2) then it behaves as the FMLC-RJ algorithm. Therefore, the algorithm executes greater number of instructions in these two cases.

Figure 6.7 shows the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 when the ASCII-Based-RJ algorithm is used.

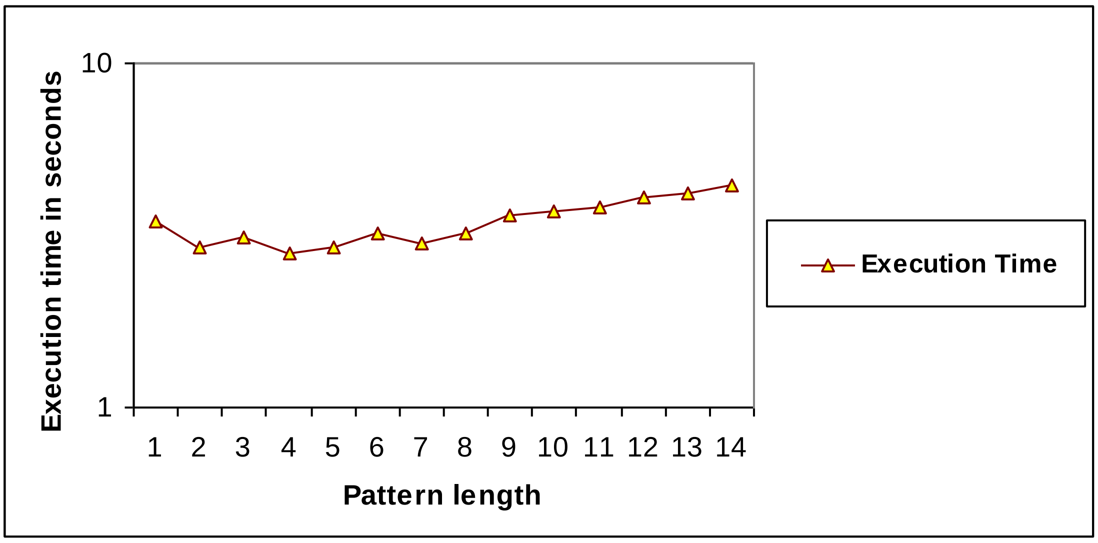


Figure 6.7: Experimental results of ASCII-Based-RJ algorithm (Execution time in seconds).

The results reveal that the best performance of the ASCII-Based-RJ algorithm is when the pattern was relatively long (more than 3 characters). This result is reasonable, because the segments of the text that will be excluded during the searching phase increase as the pattern gets longer.

Figure 6.8 shows the average number of the executed instructions for each patterns sample of each pattern length from 1 to 14 when the ASCII-Based-RJ algorithm is used.

The number of executed instructions ranges between (7000) and (1000) in all lengths of the sample patterns. The reason behind this is that the ASCII-Based-RJ algorithm is independent of the length of the pattern, and it depends on the number of the characters which appear in the pattern and do not appear in the text.

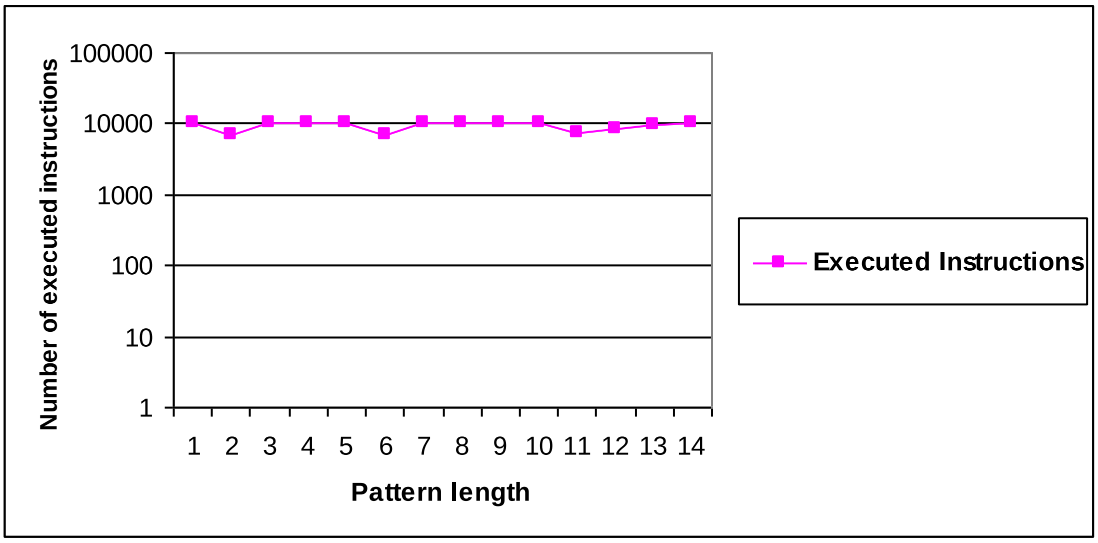


Figure 6.8: Experimental results of ASCII-Based-RJ algorithm (Number of executed instructions).

Figure 6.9 shows the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 utilizing brute force algorithm.

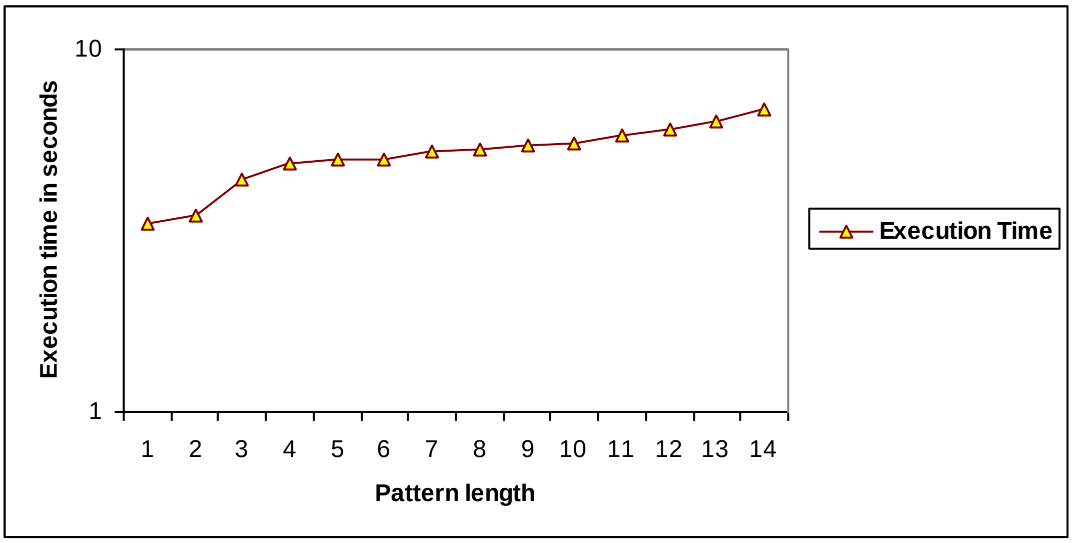


Figure 6.9: Experimental results of the brute force algorithm (Execution time in seconds).

The best performance of the brute force algorithms is when the length of the pattern was relatively short. Since the algorithm compares almost *m* characters at each index of the text, the execution time increases as *m* gets larger.

Figure 6.10 shows the average number of the executed instructions for each patterns sample of each pattern length from 1 to 14 utilizing brute force algorithm.

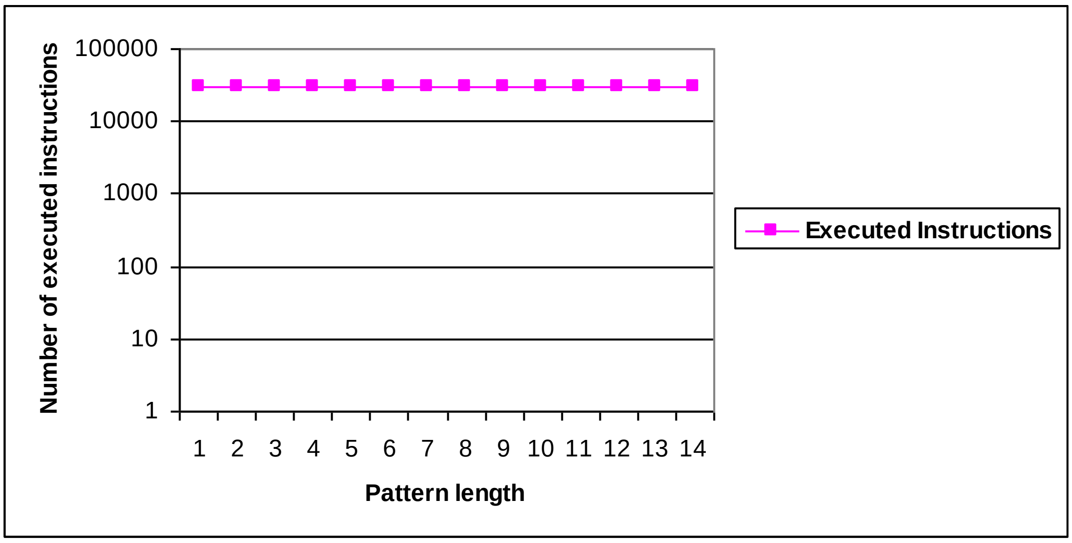


Figure 6.10: Experimental results of the brute force algorithm (Number of executed instructions).

The brute force algorithm uses a nested loop when searching for the pattern ion the text. It almost always has the same number of the executed instructions.

Figure 6.11 shows the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 when the Boyer-Moore algorithm is used.

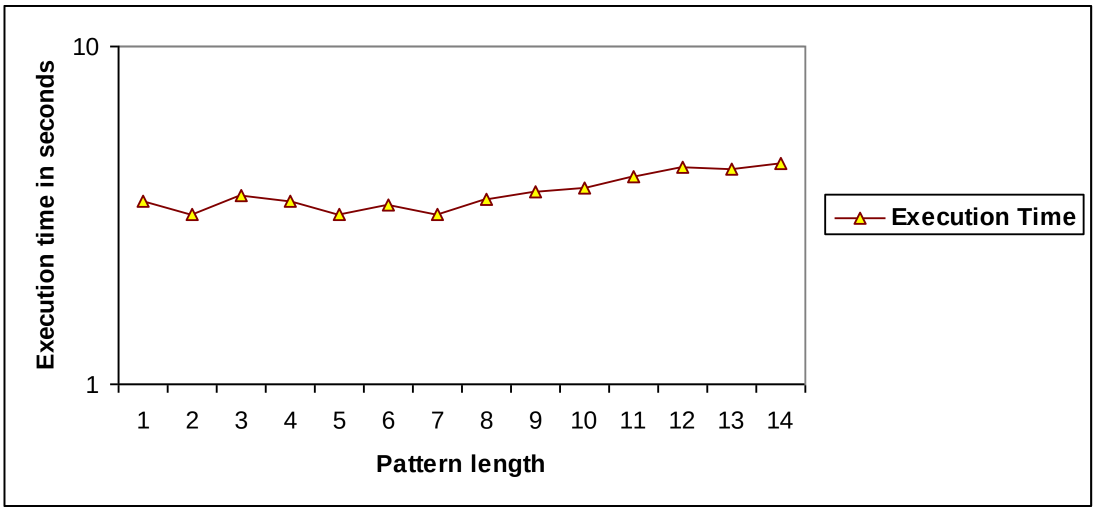


Figure 6.11: Experimental results of the Boyer-Moore algorithm (Execution time in seconds).

It can be seen in the figure that the best performance of the Boyer-Moore algorithm is when the pattern was relatively long (more than 4 characters). This result is reasonable, because the algorithm collects more information about the pattern when it is long.

Figure 6.12 shows the average number of the executed instructions for each patterns sample of each pattern length from 1 to 14 when the Boyer-Moore algorithm is used.

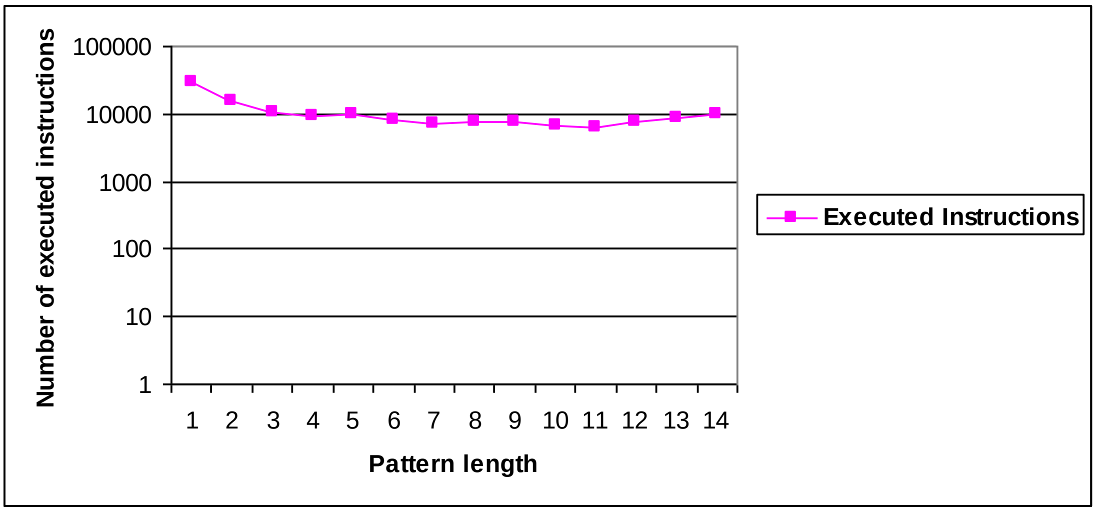


Figure 6.12: Experimental results of the Boyer-Moore algorithm (Number of executed instructions).

The number of the executed instructions decreases as the pattern gets longer. This because the Boyer-Moore algorithm uses the prefixes and suffixes instead using one character (which caused the mismatch to occur) when deciding the length of the shifts. In other words, when the length of the pattern is short, then the chance of finding a common prefix and suffix of the portion *u* is weak.

Figure 6.13 shows a comparison between execution times of the FC-RJ, FLC-RJ, FMLC-RJ, ASCII-Based-RJ, Brute Force, and Boyer-Moore algorithms for each patterns sample of each pattern length from 1 to 14.

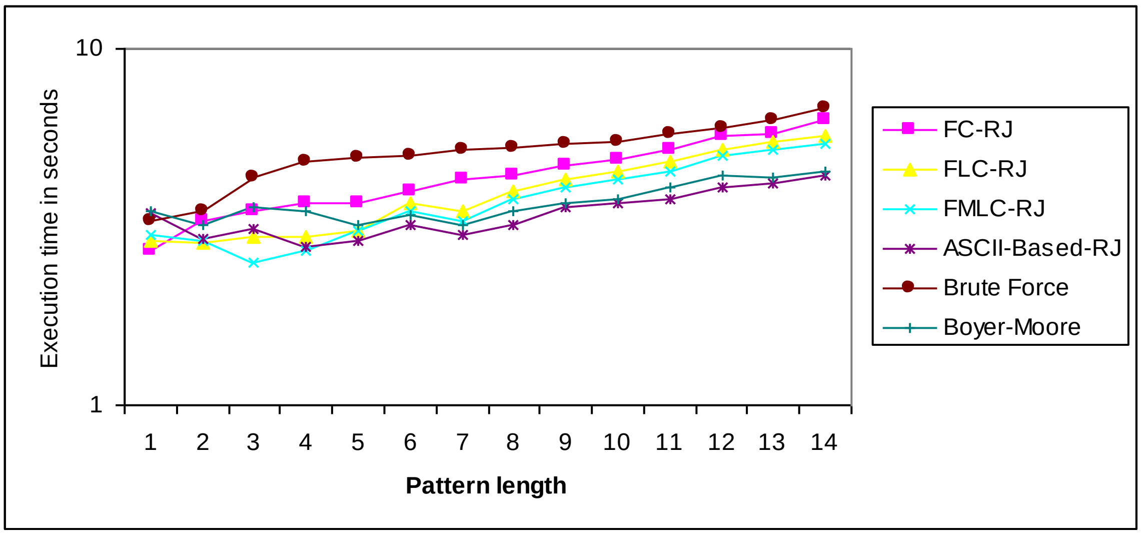


Figure 6.13: Execution times of the tested algorithms.

It is obvious that the proposed algorithms enhance the execution time of string matching as compared to the brute force and Boyer-Moore algorithms. This enhancement is calculated by considering the differences in execution times of the algorithms to search for 14 patterns samples as recorded in Table 6.1.

Table 6.2 presents the percentages of enhancements of the proposed algorithms as compared to the brute force and Boyer-Moore algorithms. Note that, the negative percentage of an algorithm denotes that the proposed algorithm was worse than the one that compared with it.

Table 6.2: Percentages of enhancements in execution time (*m* = 14 characters).

Algorithm	Enhancement on Brute Force	Enhancement on Boyer-Moore
FC-RJ	7.4 %	-28.5 %
FLC-RJ	16.2 %	-21.0 %
FMLC-RJ	20.6 %	-16.6 %
ASCII-Based-RJ	35.3 %	2.3 %

To sum up, the above performance results demonstrate that the ASCII-Based-RJ algorithm is the most flexible string matching algorithm. Overall, it is superior to all other string matching algorithms considered in this research.

Chapter Seven: Conclusions and Future Works

7.1. Conclusions

In this thesis, four new single exact pattern matching algorithms are proposed. They are: FC-RJ, FLC-RJ, FMLC-RJ, and ASCII-Based-RJ algorithms. Furthermore, a string matching tool (SMT-RJ) has been built to simulate and test the algorithms. The new algorithms, in addition to the brute force and Boyer-Moore, have been implemented and compared in the tool.

The FC-RJ, FLC-RJ and FMLC-RJ algorithms outperformed the brute force algorithm by 7.4 percent, 16.2 percent, and 20.6 percent, respectively. The reason behind this is that the new algorithms add some restrictions (conditions) to a segment in the text to be considered as an expected occurrence of the pattern, and then it can be referenced during the searching phase. If the added conditions are not satisfied for a segment in the text; it will be excluded during the searching phase.

The ASCII-Based-RJ algorithm outperformed both brute force and Boyer-Moore algorithms by 35.3 percent and 2.3 percent respectively. In some cases, FLC-RJ and FMLC-RJ algorithms gave better performance than the others. This occurred when the length of the pattern was two and three characters. In such cases, the FLC-RJ and FMLC-RJ do not enter the searching phase, and the time is consumed in the preprocessing phase, while the other algorithms search for the pattern in the text.

7.2. Future Works

The future works that may enhance the proposed algorithms and the simulator may be summarized as follows:

- Proposing new algorithms that allow the user to search for multiple patterns in a source text at the same time.
- Proposing new algorithms that allow the user to search multiple source texts for multiple patterns at the same time.
- The simulator could be able to make a performance comparison by itself, rather than the user does it manually. This option may be achieved by allowing the user to insert a pattern and letting the simulator to search for the pattern in the source string using the available algorithms, one by another, automatically.

References

1. Aho A. and Corasick M. **Efficient string matching: an aid to bibliographic search**, Communications of the ACM, 18(6), 1975, pp.333-340.
2. Alqadi Z., Aqel M. and El Emary I. **Multiple-Skip Multiple-Pattern Matching Algorithm (MSMPMA)**, International Journal of Computer Science, 34(2), 2007, pp.14-20.
3. Amintoosi M., Yazdi H., Fathy M. and Monsefi R. **Using Pattern Matching for Tiling and Packing Problems**, European Journal of Operational Research, 183(3), 2006, pp.950-960.
4. Amir A., Cole R., Hariharan R., Lewenstein M. and Porat E. **Overlap Matching**, Journal of Information and Computation, 181(1), 2002, pp.57-74.
5. Boyer R. and Moore J. **A Fast String Searching Algorithm**, Communications of the ACM, 20(10), 1977, pp.761-772.
6. Cantone D. and Faro S. **A Space Efficient Bit-Parallel Algorithm for the Multiple String Matching Problem**, International Journal of Foundations of Computer Science, 17(6), 2006, pp.1235-1251.
7. Cegielski P., Guessarian I. and Matiyasevich Y. **Multiple serial episodes matching**, Journal of Information Processing Letters, 98(6), 2006, pp.211-218.
8. Charras C. and Lecroq T. **Handbook of Exact String-Matching Algorithms**, 1st ed., King's College Publications, London-UK, 2004, pp.19-24.
9. Crochemore M., Czumaj A., Gasieniec L., Jarominek S., Lecroq T., Plandowski W. and Rytter W. **Speeding Up Two String Matching Algorithms**, Algorithmica, 12(4/5), 1994, pp.247-267.
10. Crochemore M., Hancart C. and Lecroq T. **A Unifying Look at the Apostolico–Giancarlo String-Matching Algorithm**, Journal of Discrete Algorithms, 1(1), 2003, pp.37-52.
11. Danvy O. and Rohde H. **On Obtaining the Boyer–Moore String-Matching Algorithm by Partial Evaluation**, Journal of Information Processing Letters, 99(4), 2006, pp.158-162.
12. Franek F., Jennings C. and Smyth W.F. **A simple fast hybrid pattern-matching algorithm**, Journal of Discrete Algorithms, 5(4), 2006, pp.682-695.

13. Gongshe L., Jianhua L. and Shenghong L. **New multi-pattern matching algorithm**, Journal of Systems Engineering and Electronics, 17(2), 2006, pp.437-442.
14. Idury R. and Schaffer A. **Multiple Matching of Rectangular Patterns**, Information and Computation, 117(1), 1995, pp.78-90.
15. Karp R. and Rabin M. **Efficient Randomized Pattern-Matching Algorithms**, IBM Journal on Research Development, 31(2), 1987, pp.249-260.
16. Kim S. and Kim Y. **A Fast Multiple String-Pattern Matching Algorithm**, Proc. Of the 17th AoM/IAoM Conference on Computer Science, San Diego, CA, 1999, pp.44-49.
17. Knuth D., Morris J. and Pratt V. **Fast Pattern Matching in Strings**, SIAM Journal on Computing, 6(1), 1977, pp.323-350.
18. Lecroq T. **Fast exact string matching algorithms**, Journal of Information Processing Letters, 102(6), 2007, pp.229-235.
19. Lipsky O. and Porat E. **L_1 Pattern Matching Lower Bound**, Journal of Information Processing Letters, 105(4), 2007, pp.141-143.
20. Michailidis P. and Margaritis K. **Processor Array Architectures for Flexible Approximate String Matching**, Journal of Systems Architecture, 54(1-2), 2007, pp.35-54.
21. Morris J. and Pratt V. **A Linear Pattern-Matching Algorithm**, Technical Report 40, University of California, Berkeley, 1970.
22. Navarro G. and Fredriksson K. **Average Complexity of Exact and Approximate Multiple String Matching**, Journal of Theoretical Computer Science, 321(2-3), 2004, pp.283-290.
23. Rytter W. **The Number of Runs in a String**, Journal of Information and Computation, 205(9), 2007, pp.1459-1469.
24. Salmela L. and Tarhio J. **Fast parameterized matching with q-grams**, Proceedings of the 17th Combinatorial Pattern Matching, 2006, pp.354–364.
25. Sheu T.-F., Huang N.-F. and Lee H.-P. **Hierarchical Multi-Pattern Matching Algorithm for Network Content Inspection**, Journal of Information Sciences, 178(14), 2008, pp.2880-2898.
26. Sommerville I. **Software Engineering**. 6th edition, Addison Wesley, 2001.

27. Watson B. **A New Family of Commentz-Walter-Style Multiple-Keyword Pattern Matching Algorithms**, South African Computer Journal, 30(1), 2003, pp.29-33.
28. Watson B. **A New Regular Grammar Pattern Matching Algorithm**, Journal of Theoretical Computer Science, 299(1-3), 2002, pp.509-521.
29. Watson B. and Watson R. **A Boyer–Moore-Style Algorithm for Regular Expression Pattern Matching**, Journal of Science of Computer Programming, 48(2-3), 2003, pp.99-117.
30. Wu S. and Manber U. **A Fast Algorithm for Multi-Pattern Searching**, Technical Report TR-94-17, Department of Computer Science, University of Arizona, May 1994.
31. Wu Y.-C., Yang J.-C. and Lee Y.-S. **A Weighted String Pattern Matching-Based Passage Ranking Algorithm for Video Question Answering**, Journal of Expert Systems with Applications, 34(4), 2007, pp.2588-2600.

الملخص

يمكن تعريف مشكلة مطابقة النصوص على أنها عملية إيجاد مواقع وجود نص صغير (Pattern) وحجمه (*m*) داخل نص أكبر (Text) وحجمه (*n*). وتعتبر خوارزميات مطابقة النصوص مكوّناً مهماً من المكوّنات التي يتم استخدامها في تنفيذ البرمجيات التطبيقة التي يتم تشغيلها في معظم نظم التشغيل. في العديد من برامج استرجاع المعلومات وتحرير النصوص، من الضروري أن يكون المستخدم قادراً وبسرعة على إيجاد بعض أو كل الأماكن التي يتواجد فيها نص معيّن داخل نص آخر. في هذه الدراسة تم استعراض معظم خوارزميات مطابقة النصوص المعروفة والمستخدمة حالياً، وذلك في سبيل تحسين بعضها وتقديم خوارزميات جديدة في هذا المجال.

تم في هذه الدراسة اقتراح أربع خوارزميات لمطابقة النصوص، وهي: FC-RJ و FLC-RJ و FMLC-RJ و ASCII-Based-RJ. علاوةً على ذلك، تم تطوير أداة خاصة بمطابقة النصوص (SMT-RJ)، وتم في هذه الأداة تنفيذ وفحص ومقارنة الخوارزميات الأربع الجديدة، بالإضافة إلى خوارزميتي Brute Force و Boyer-Moore.

أظهرت نتائج الدراسة أن خوارزميات FC-RJ و FLC-RJ و FMLC-RJ كانت أفضل أداءً من خوارزمية Brute Force بنسب متفاوتة. بينما كان أداء خوارزمية ASCII-Based-RJ أفضل أداءً من خوارزميتي Brute Force و Boyer-Moore بنسب متفاوتة.

Appendix A: The ASCII Table

The following table shows the ASCII code for the characters set. The non-printing characters are from the decimal 0 to 31, while the printing characters are from decimal 32 to 127.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	“	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	‘	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	TAB	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Appendix B: Major Code of the SMT-RJ System

Open sub-menu:

```
Private Sub mnuFileOpen_Click()

'Error handling
On Error GoTo errmsg
With CommonDialog1
        .CancelError = True
        .DialogTitle = "CommonDialog1"
        .Filter = "Text files only (*.txt)|*.txt|All files
                                           (*.*)|*.*"
        .FilterIndex = 1
        .Flags = cdlOFNHideReadOnly Or cdlOFNOverwritePrompt Or
                                           cdlOFNPathMustExist
        .InitDir = App.Path
        .ShowOpen
End With

Dim Filenum, i As Integer
FilePath = CommonDialog1.FileName
FileSize = FileLen(FilePath)
Label1.Visible = False
Label2.Visible = False
Label3.Visible = False
rt1.Visible = True

Dim ff As Long, sData As String
'Open and save the whole text file into sFile string variable
Filenum = FreeFile
Open FilePath For Input As Filenum
    sFile = Input(LOF(Filenum), Filenum)
Close Filenum
rt1.Text = sFile
n = Len(sFile)
```

```
'Dynamic allocation of arr() to hold the text
ReDim arr(n)
i = 0
ff = FreeFile
'Getting the file into the array arr(n) from sFile
Open FilePath For Input As #ff
  Do While Not EOF(ff)
    sData = Input$(1, ff)
    arr(i) = sData
    i = i + 1
    Loop
Close #ff
errmsg:
'End of Open sub-menu
End Sub
```

FC-RJ Algorithm Code:

```
Private Sub mnuFC_Click()
'Check if the user does not open a file
If sFile = "" Then
    MsgBox "Please insert a text before trying to search for a
                                                pattern.", 16, "Error"
Else
output = ""
Dim first As String
Dim j As Integer
Dim i As Integer
Dim temp As Long
Dim k As Integer
Dim tempp As String
k = 0
i = 0
p = InputBox("Please enter the pattern: ", "FC-RJ Algorithm")
m = Len(p)
'Load pattern into pattern() if it's length was not zero
If m <> 0 Then
Open App.Path & "\patterns\PatternFile.txt" For Output As #1
Print #1, p
Close #1
ReDim pattern(m) As String
temp = FreeFile
Open App.Path & "\patterns\PatternFile.txt" For Input As #temp
For k = 0 To m - 1
     tempp = Input$(1, temp)
     pattern(k) = tempp
Next k
Close #temp
End If
first = pattern(0)
n = Len(sFile)
```

```
'Dynamic allocation of OccuerrenceList() array
ReDim OccurrenceList((n - m) + 1) As Integer
'Search for pattern's first character in the text,
'and save the indices in the OccurrenceList() array
For j = 0 To (n - m)
    If arr(j) = first Then
        OccurrenceList(i) = j
        i = i + 1
    End If
Next j
'After the OccurrenceList is ready, call the searching phase
Call SearchFC(i)
End If
End Sub
```

Code of the searching phase of FC-RJ algorithm:

```
Public Sub SearchFC(i As Integer)
output = ""
rt1.Font.Bold = False
rt1.Font.Size = 11
'Search for the pattern if it's first character was found
If i > 0 Then
    'If the pattern's length is one character
    If m = 1 Then
        For y = 0 To i - 1
          output = output & "(" & OccurrenceList(y) & ")"
          rt1.SelStart = OccurrenceList(y)
          rt1.SelLength = Len(p)
          rt1.SelColor = 2211
          rt1.SelFontSize = 14
          rt1.SelUnderline = True
        Next y
    Else
    If m > 1 Then
         Dim c
         Dim x
         Dim l
         Dim count As Integer
         Dim value As Boolean
         c = x = l = 0
         count = 1
         While c < i
            InstCounter = InstCounter + 1
            value = True
            For l = OccurrenceList(c) + 1 To OccurrenceList(c) +
                                                          m - 1
```

```
           If arr(l) <> pattern(count) Then
                    value = False
                    Exit For
                 End If
                 count = count + 1
            Next l
            If value = True Then
                 output = output & "(" & OccurrenceList(c) & ")"
                 rt1.SelStart = OccurrenceList(c)
                 rt1.SelLength = Len(p)
                 rt1.SelColor = 2211
                 rt1.SelFontSize = 14
                 rt1.SelUnderline = True
            End If
            c = c + 1
            count = 1
        Wend
        If output = "" Then
           MsgBox "The pattern:" & vbNewLine & p & vbNewLine &
                                 "is not found.", 64, "Result"
        End If
       End If
      End If
    Else
    If output = "" Then
        MsgBox "The pattern:" & vbNewLine & p & vbNewLine & "is
                                      not found.", 64, "Result"
    End If
End If

End Sub
```

FLC-RJ Algorithm Code:

```
If sFile = "" Then
    MsgBox "Please insert a text before trying to search for a
                                    pattern.", 16, "Error"
Else
    output = ""
    Dim first, last  As String
    Dim i,j,k As Integer
    Dim temp As Long
    Dim tempp As String
    k = 0
    i = 0
    p = InputBox("Please enter the pattern: ", "FLC-RJ
                                        Algorithm")
    m = Len(p)

    If m <> 0 Then
        Open App.Path & "\patterns\PatternFile.txt" For Output
                                                     As #1
        Print #1, p
        Close #1
        ReDim pattern(m) As String
        temp = FreeFile
        Open App.Path & "\patterns\PatternFile.txt" For Input As
                                                       #temp
        For k = 0 To m - 1
            tempp = Input$(1, temp)
            pattern(k) = tempp
        Next k

         Close #temp
    End If

    first = pattern(0)
    last = pattern(m - 1)
    n = Len(sFile)
      'If the pattern's length is > 1, call the searching phase
      If m > 1 Then
        ReDim OccurrenceList((n - m) + 1) As Integer
        For j = 0 To (n - m)
            If arr(j) = first And arr(j + m - 1) = last Then
                OccurrenceList(i) = j
                i = i + 1
            End If
        Next j
        Call SearchFLC(i)
    Else
 'If the pattern's length is 1, behave as FC-RJ Algorithm

    If m < 2 Then
        ReDim OccurrenceList((n - m) + 1) As Integer
        For j = 0 To (n - m)
           If arr(j) = first Then
                OccurrenceList(i) = j
                i = i + 1
            End If
        Next j
        Call SearchFC(i)
    End If
End If
End If

End Sub
```

Code of the searching phase of FLC-RJ algorithm:

```
Public Sub SearchFLC(i As Integer)
output = ""
rt1.Font.Bold = False
rt1.Font.Size = 11
If i > 0 Then
  Dim c
  Dim x
  Dim l
  Dim count As Integer
  Dim value As Boolean
  c = x = l = 0
  count = 1
  While c < i
        value = True
        For l = OccurrenceList(c) + 1 To OccurrenceList(c) + m-2
            If arr(l) <> pattern(count) Then
                 value = False
                 Exit For
            End If
            count = count + 1
         Next l
         If value = True Then
            output = output & "(" & OccurrenceList(c) & ")"
            rt1.SelStart = OccurrenceList(c)
            rt1.SelLength = Len(p)
            rt1.SelColor = 2211
            rt1.SelFontSize = 14
            rt1.SelUnderline = True
         End If
         c = c + 1
         count = 1
  Wend
      If output = "" Then
         MsgBox "The pattern:" & vbNewLine & p & vbNewLine & "is
                                                  not found.", 64, "Result"
      End If
  Else
    If output = "" Then
      MsgBox "The pattern:" & vbNewLine & p & vbNewLine & "is
                                                  not found.", 64, "Result"
    End If
 End If
End Sub
```

FMLC-RJ Algorithm Code:

```
Private Sub mnuFMLC_Click()
If sFile = "" Then
    MsgBox "Please insert a text before trying to search for a
pattern.", 16, "Error"
Else
    output = ""
    Dim first, mid, last As String
    Dim middle, i, j, k As Integer
    Dim temp As Long
    Dim tempp As String
    k = 0
    i = 0
    p = InputBox("Please enter the pattern: ", "FMLC-RJ
                                             Algorithm")
    m = Len(p)

    If m <> 0 Then
        Open App.Path & "\patterns\PatternFile.txt" For Output
                                                         As #1
        Print #1, p
        Close #1

         ReDim pattern(m) As String
        temp = FreeFile
        Open App.Path & "\patterns\PatternFile.txt" For Input As
                                                          #temp
        For k = 0 To m - 1
            tempp = Input$(1, temp)
            pattern(k) = tempp
        Next k
        Close #temp
    End If

    first = pattern(0)
    last = pattern(m - 1)
    middle = m / 2
    mid = pattern(middle)
    n = Len(sFile)
    'If the pattern's length is > 2, call the searching phase
    If m > 2 Then
        ReDim OccurrenceList((n - m) + 1) As Integer
        For j = 0 To (n - m)
            If arr(j) = first And arr(j + middle) = mid And
                                 arr(j + m - 1) = last Then
                OccurrenceList(i) = j
                i = i + 1
            End If
        Next j
        Call SearchFMLC(i)
```

```
Else
'If the pattern's length is = 2, behave as FLC-RJ Algorithm
        If m = 2 Then
             ReDim OccurrenceList((n - m) + 1) As Integer
             For j = 0 To (n - m)
                If arr(j) = first And arr(j + m - 1) = last Then
                    OccurrenceList(i) = j
                    i = i + 1
                End If
            Next j
        Call SearchFLC(i)
        Else
'If the pattern's length is = 1, behave as FC-RJ Algorithm
            ReDim OccurrenceList((n - m) + 1) As Integer
            For j = 0 To (n - m)
                If arr(j) = first Then
                    OccurrenceList(i) = j
                    i = i + 1
                End If
            Next j
        Call SearchFC(i)
    End If
End If
End If
End Sub
```

Code of the searching phase of FMLC-RJ algorithm:

```
Public Sub SearchFMLC(i As Integer)
output = ""
rt1.Font.Bold = False
rt1.Font.Size = 11
If i > 0 Then
  Dim c, x, l as integer
  Dim middle As Integer
  Dim count As Integer
  Dim value As Boolean
  middle = m / 2
  c = x = l = 0
  count = 1
  While c < i
        value = True
        For l = OccurrenceList(c) + 1 To OccurrenceList(c) + m-2
            If count = middle Then
                l = l + 1
                count = count + 1
            Else
                If arr(l) <> pattern(count) Then
                    value = False
                    Exit For
                End If
            End If
            count = count + 1
         Next l
```

```
            If value = True Then
               output = output & "(" & OccurrenceList(c) & ")"
               rt1.SelStart = OccurrenceList(c)
               rt1.SelLength = Len(p)
               rt1.SelColor = 2211
               rt1.SelFontSize = 14
               rt1.SelUnderline = True
            End If
            c = c + 1
            count = 1
     Wend
     If output = "" Then
         MsgBox "The pattern:" & vbNewLine & p & vbNewLine & "is
                                              not found.", 64, "Result"
     End If

     Else
       If output = "" Then
         MsgBox "The pattern:" & vbNewLine & p & vbNewLine & "is
                                              not found.", 64, "Result"
       End If
   End If

   End Sub
```

ASCII-Based-RJ Algorithm Code:

```
Private Sub mnuASCIIAlg_Click()

If sFile = "" Then
    MsgBox "Please insert a text before trying to search for a
                                    pattern.", 16, "Error"
Else
    output = ""
    Dim j As Integer
    Dim i As Integer
    Dim temp As Long
    Dim k As Integer
    Dim tempp As String
    Dim Ascii_Arr() As Integer
    Dim Skip_Arr() As Integer
    Dim y, z, x As Integer

    k = i = 0
    p = InputBox("Please enter the pattern: ", "ASCII-Based-RJ
                                                Algorithm")
    m = Len(p)
    If m <> 0 Then
        Open App.Path & "\patterns\PatternFile.txt" For Output
                                                      As #1
        Print #1, p
        Close #1
        ReDim pattern(m) As String
        temp = FreeFile
        Open App.Path & "\patterns\PatternFile.txt" For Input As
                                                      #temp
        For k = 0 To m - 1
            tempp = Input$(1, temp)
            pattern(k) = tempp
        Next k
        Close #temp
    End If

 ASize = 256
 y = z = 0
 x = n - 1

ReDim Ascii_Arr(ASize) As Integer

For mmm = o To (ASize - 1)
    Ascii_Arr(mmm) = 0
Next mmm

ReDim Skip_Arr(n) As Integer

For j = 0 To m - 1
    Ascii_Arr(Asc(pattern(j))) = Ascii_Arr(Asc(pattern(j))) + 1
Next j
```

```
For x = (n - 1) To 0 Step -1
    If Ascii_Arr(Asc(arr(x))) = 0 And x >= (m - 1) Then
        For y = (x - m + 1) To x
            If Skip_Arr(y) = 0 Then
                Skip_Arr(y) = -1
            Else
                Exit For
            End If
        Next y
    Else
    If Ascii_Arr(Asc(arr(x))) = 0 And x < (m - 1) Then
        For y = 0 To x
            If Skip_Arr(y) = 0 Then
                Skip_Arr(y) = -1
            Else
                Exit For
            End If
        Next y
    End If
    End If
Next x

ReDim OccurrenceList((n - m) + 1) As Integer

For z = 0 To (n - m)
    If Skip_Arr(z) <> -1 And arr(z) = pattern(0) Then
        OccurrenceList(i) = z
        i = i + 1
    End If
Next z
Call SearchASCIIAlg(i)
End If
End Sub
```

Code of the searching phase of ASCII-Based-RJ algorithm:

```
Public Sub SearchASCIIAlg(i As Integer)
output = ""
rt1.Font.Bold = False
rt1.Font.Size = 11
If i > 0 Then
    If m = 1 Then
        For y = 0 To i - 1
            output = output & "(" & OccurrenceList(y) & ")"
            rt1.SelStart = OccurrenceList(y)
            rt1.SelLength = Len(p)
            rt1.SelColor = 2211
            rt1.SelFontSize = 14
            rt1.SelUnderline = True
        Next y
    Else
    If m > 1 Then
         Dim c
         Dim x
         Dim l
```

```
            Dim count As Integer
            Dim value As Boolean
            c = x = l = 0
            count = 1
            While c < i
               value = True
               For l = OccurrenceList(c)+1 To OccurrenceList(c)+m-1
                    If arr(l) <> pattern(count) Then
                       value = False
                       Exit For
                    End If
                    count = count + 1
               Next l
               If value = True Then
                   output = output & "(" & OccurrenceList(c) & ")"
                   rt1.SelStart = OccurrenceList(c)
                   rt1.SelLength = Len(p)
                   rt1.SelColor = 2211
                   rt1.SelFontSize = 14
                   rt1.SelUnderline = True
               End If
               c = c + 1
               count = 1
           Wend
           If output = "" Then
                MsgBox "The pattern:" & vbNewLine & p & vbNewLine &
                                      "is not found.", 64, "Result"
           End If
          End If
        End If
       Else
        If output = "" Then
             MsgBox "The pattern:" & vbNewLine & p & vbNewLine &
                                   "is not found.", 64, "Result"
        End If
       End If

   End Sub
```

Brute Force Algorithm Code:

```
Private Sub mnuBruteForce_Click()

If sFile = "" Then
    MsgBox "Please insert a text before trying to search for a
                                        pattern.", 16, "Error"
Else
    output = ""
    rt1.Font.Bold = False
    rt1.Font.Size = 11
    Dim j As Integer
    Dim temp As Long
    Dim k As Integer
    Dim s As Integer
    Dim tempp As String
    k = 0
    p = InputBox("Please enter the pattern: ", "Brute Force
                                                 Algorithm")
    m = Len(p)
    If m <> 0 Then
        Open App.Path & "\patterns\PatternFile.txt" For Output
                                                        As #1
        Print #1, p
        Close #1

        ReDim pattern(m) As String

         temp = FreeFile
        Open App.Path & "\patterns\PatternFile.txt" For Input As
                                                         #temp
        For k = 0 To m - 1
            tempp = Input$(1, temp)
            pattern(k) = tempp
        Next k
        Close #temp
    End If

    n = Len(sFile)

    For j = 0 To (n - m)
     For s = 0 To (m - 1)
        If pattern(s) <> arr(j + s) Then
            Exit For
        End If
     Next s
     If s = m Then
        output = output & "(" & (j) & ")"
        rt1.SelStart = j
        rt1.SelLength = Len(p)
        rt1.SelColor = 2211
        rt1.SelFontSize = 14
        rt1.SelUnderline = True
     End If
    Next j
    If output = "" Then
        MsgBox "The pattern:" & vbNewLine & p & vbNewLine & "is
                                                      not found.", 64, "Result"
    End If

End If
End Sub
```

Boyer-Moor Algorithm Code:

```
Private Sub mnuBM_Click()

If sFile = "" Then
    MsgBox "Please insert a text before trying to search for a
                                    pattern.", 16, "Error"
Else
    output = ""
    rt1.Font.Bold = False
    rt1.Font.Size = 11
    Dim j As Integer
    Dim temp As Long
    Dim k As Integer
    Dim i As Integer
    Dim jj As Integer
    Dim tempp As String
    Dim max As Integer
    k = 0
    p = InputBox("Please enter the pattern: ", "Boyer-Moore
                                             Algorithm")
    m = Len(p)

    If m <> 0 Then
        Open App.Path & "\patterns\PatternFile.txt" For Output
                                                     As #1
        Print #1, p
        Close #1

         ReDim pattern(m) As String
        temp = FreeFile
        Open App.Path & "\patterns\PatternFile.txt" For Input As
                                                      #temp
        For k = 0 To m - 1
            tempp = Input$(1, temp)
            pattern(k) = tempp
        Next k
        Close #temp
    End If

    ASize = 256
    ReDim bmGs(m) As Integer
    ReDim bmBc(ASize) As Integer

    'Preprocessing
    Call preBmGs
    Call preBmBc

    'Searching
    j = o
    While j <= (n - m)
        For i = (m - 1) To 0 Step -1
            If pattern(i) <> arr(i + j) Then
                Exit For
            End If
        Next i
        If i < 0 Then
            output = output & "(" & j & ")"
            rt1.SelStart = j
            rt1.SelLength = Len(p)
            rt1.SelColor = 2211
            rt1.SelFontSize = 14
            rt1.SelUnderline = True
            j = j + bmGs(0)
        Else
          If bmGs(i) >= (bmBc(Asc(arr(i + j))) - m + 1 + i) Then
          jj=bmGs(i) Else jj=(bmBc(Asc(arr(i+j)))- m + 1 + i)
          j = j + jj
          End If
    Wend
    If output = "" Then
        MsgBox "The pattern:" & vbNewLine & p & vbNewLine & "is
                                                  not found.", 64, "Result"
    End If

End Sub
```

Code of the `preBmGs()` routine of Boyer-Moore algorithm:

```
Public Sub preBmGs()
Dim i, j As Integer
Call suffixes

For i = 0 To m - 1
    bmGs(i) = m
Next i
j = 0
For i = (m - 1) To 0 Step -1
    If suff(i) = (i + 1) Then
        While j < (m - 1 - i)
            If bmGs(j) = m Then
                bmGs(j) = (m - 1 - i)
            End If
            j = j + 1
         Wend
    End If
Next i

For i = 0 To (m - 2)
    bmGs(m - 1 - suff(i)) = (m - 1 - i)
Next i

End Sub
```

Code of the `suffixes()` routine of Boyer-Moore algorithm:

```
Public Sub suffixes()
Dim f, g, i As Integer
ReDim suff(m) As Integer
suff(m - 1) = m
g = m - 1

For i = (m - 2) To 0 Step -1
    If i > g Then
        If (suff(i + m - 1 - f) < (i - g)) Then
            suff(i) = suff(i + m - 1 - f)
        End If
    Else
        If i < g Then g = i
        f = i
        While g > 0 And (pattern(g) = pattern(g + m - 1 - f))
            g = g - 1
        Wend
        suff(i) = f - g
    End If
Next i

End Sub
```

Code of the `preBmBc()` routine of Boyer-Moore algorithm:

```
Public Sub preBmBc()

Dim i As Integer
Dim q As Integer

For i = 0 To (ASize - 1)
    bmBc(i) = m
Next i

For q = 0 To (m - 2)
    bmBc(Asc(pattern(q))) = (m - q - 1)
Next q

End Sub
```

Small Letters Text Generator's Code:

```
Private Sub mnuSLO_Click()
  Dim x As Integer
  Dim NameOfFile As String
  Dim NumOfChars As Integer
  Dim max As Integer
  Dim min As Integer
  Dim random As Integer
  Dim str As String
  NumOfChars = Val(InputBox("How many charaters you want to
                       generate?", "Number of characters"))
  NameOfFile = InputBox("Insert a name for the file: " &
   vbNewLine & vbNewLine & "Note: don't type extenstion.
   The file will be saved as (.txt) by default.", "Name of
   file")

  max = 122
  min = 97

  For x = 1 To NumOfChars
      Randomize
      random = Int(Rnd * (max - min)) + min
      str = str & Chr(random)
  Next x

  Open App.Path & "\texts\" & NameOfFile & ".txt" For Output As
                                                            #2
        Print #2, str
  Close #2

End Sub
```

Capital Letters Text Generator's Code:

```
Private Sub mnuCLO_Click()
  Dim x As Integer
  Dim NameOfFile As String
  Dim NumOfChars As Integer
  Dim max As Integer
  Dim min As Integer
  Dim random As Integer
  Dim str As String
  NumOfChars = Val(InputBox("How many charaters you want to
                       generate?", "Number of characters"))
  NameOfFile = InputBox("Insert a name for the file: " &
     vbNewLine & vbNewLine & "Note: don't type extenstion. The
     file will be saved as (.txt) by default.", "Name of file")

  max = 90
  min = 65

  For x = 1 To NumOfChars
      Randomize
      random = Int(Rnd * (max - min)) + min
      str = str & Chr(random)
  Next x

  Open App.Path & "\texts\" & NameOfFile & ".txt" For Output As
                                                              #2
        Print #2, str
  Close #2

End Sub
```

Mixed Letters Text Generator's Code:

```
Private Sub mnuMixed_Click()
  Dim x As Integer
  Dim NameOfFile As String
  Dim NumOfChars As Integer
  Dim max As Integer
  Dim min As Integer
  Dim random As Integer
  Dim str As String
  NumOfChars = Val(InputBox("How many charaters you want to
                        generate?", "Number of characters"))
  NameOfFile = InputBox("Insert a name for the file: " &
    vbNewLine & vbNewLine & "Note: don't type extenstion. The
    file will be saved as (.txt) by default.", "Name of file")

  max = 126
  min = 32

  For x = 1 To NumOfChars
      Randomize
      random = Int(Rnd * (max - min)) + min
      str = str & Chr(random)
  Next x

  Open App.Path & "\texts\" & NameOfFile & ".txt" For Output As
                                                              #2
        Print #2, str
  Close #2

End Sub
```

Patterns Generator's Code:

```
Private Sub mnuGeneratePat_Click()

If n = 0 Then
    MsgBox "Please open a text file before generating
                               patterns.", 16, "Error"
Else

    Dim x, i, max, PatLen, z, RandomIndex As Integer
    Dim PatStr As String
    max = n - 20
    PatLen = 14
    i = 1

    For x = 0 To 299
        PatStr = ""
        RandomIndex = Int(Rnd * max)
        For z = RandomIndex To (RandomIndex + PatLen - 1)
            PatStr = PatStr + arr(z)
        Next z
        Open App.Path & "\patterns\Len_14\" & x + 1 & ".txt" For
                                                    Output As #i
            Print #i, PatStr
        Close #i
        i = i + 1
    Next x
    MsgBox "The patterns have been created successfully.", 64,
                                             "Random Patterns"
End If

End Sub
```

Text File Info Code:

```
Private Sub mnuFileInfo_Click()
If sFile = "" Then
    MsgBox "Please open a text file before trying to get info.",
                                                       16, "Error"
Else
    output = ""
    Dim NumOfWords As Integer
    Dim NumOfCharsWSpaces As Integer
    Dim NumOfCharsNoSpaces As Integer
    Dim NumOfSpaces As Integer
    Dim c As Integer

    For c = 0 To n - 1
        If arr(c) = " " Then
            NumOfSpaces = NumOfSpaces + 1
            If arr(c - 1) <> " " Then
                NumOfWords = NumOfWords + 1
             End If
        End If
    Next c

   If arr(n - 1) <> " " Then NumOfWords = NumOfWords + 1
   NumOfCharsWSpaces = n
   NumOfCharsNoSpaces = n - NumOfSpaces

   MsgBox "Number of words: " & NumOfWords & vbNewLine & "Number
      of characters with spaces: " & n & vbNewLine & "Number of
      characters without spaces: " & NumOfCharsNoSpaces &
      vbNewLine & "Number of spaces: " & NumOfSpaces & vbNewLine
      & "File path: " & FilePath & vbNewLine & "File size: " &
      FileSize & " bytes", 64, "File info"

End If
End Sub
```