

ANIMATED COMPARISON OF STRING
MATCHING ALGORITHMS

By

JIANYU ZHONG

Bachelor of Arts

Zhongshan University

Guangdong, P.R.China

1994

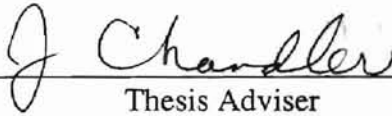
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1999

ANIMATED COMPARISON OF STRING
MATCHING ALGORITHMS


Chandler
1995
10/10/95
10/10/95
10/10/95

my major advisor, Dr. John P.
stage to us and time he has given me
more expenses of other committee
of that fact, my consent and

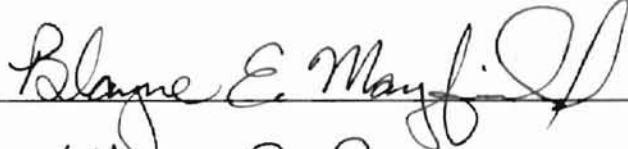
Thesis Approved:



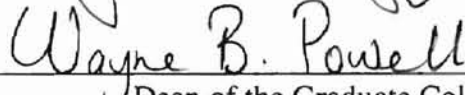
Thesis Adviser



Chair



Chair



Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my major advisor, Dr. John P. Chandler, for his intelligent guidance, enthusiastic suggestions and time he has given me toward the completion of my thesis work. My appreciation extends to other committee members, Dr. G. E. Hedrick and Dr. B. E. Mayfield, for their helpful advisement and suggestions.

In addition, I wish to express my sincere appreciation to those who gave me assistance and encouragement at times of difficulty: Mr. Chaopang Fan, Ms. Jing Tian, and all of my friends.

Finally, I would also like to give my special thanks and appreciation to my father, mother, and other members of my family whose encouragement and understanding were invaluable during the time of my study.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. ALGORITHM ANIMATION AND JAVA	3
2.1 Computer Animation	3
2.2 Algorithm Visualization and Animation	4
2.3 Java Programming Language..	4
2.3.1 How Java Works	4
2.3.2 Java Applications and Java Applets.	5
2.3.3 Description of Java..	6
III. STRING MATCHING ALGORITHMS.	8
3.1 Definition..	8
3.2 Notation and Terminology..	9
3.3 Overview of String Matching Algorithms..	10
3.4 Description of String Matching Algorithms..	12
3.4.1 Naïve String Matching Algorithm..	13
3.4.2 String Matching with Finite Automata	14
3.4.3 The Pseudo KMP Algorithm..	18
3.4.4 The True KMP Algorithm.	21
3.4.5 The Boyer-Moore Algorithm.	25
3.4.5.1 The Bad-Character Heuristic..	25
3.4.5.2 The Good-Suffix Heuristic..	27
3.4.5.3 The Boyer-Moore Matcher..	30
IV. ALGORITHM ANIMATION DESIGN AND IMPLEMENTATION..	33
4.1 The Overview of Function Areas	33
4.2 System Class Hierarchy.	35
V. COMPARISON OF STRING MATCHING ALGORITHMS.	39
5.1 Random Text and Pattern.	40
5.2 English Text and Pattern	45
5.3 Binary String Text and Pattern..	46
VI. SUMMARY, CONCLUSIONS AND SUGGESTED FUTURE WORK.	48
6.1 Summary..	48
6.2 Conclusion..	50
6.3 Suggested Future Work	50

SELECTED BIBLIOGRAPHY	51
---------------------------------	----

LIST OF TABLES

Table		Page
3.1	Dirichlet function for $P = \text{beahabab}$.	18
	Dirichlet function for $P = \text{beahabab}$.	22
3	Dirichlet function for $P = \text{beahabab}$.	32
3.4	Dirichlet function for $P = \text{beahabab}$.	32

LIST OF TABLES

Table		Page
3-1	The prefix function for $P = bcababab$	18
3-2	The next function for $P = bcababab$	22
3-3	The last-occurrence function for $P = bcababab$	32
3-4	The good-suffix function for $P = bcababab$	32

LIST OF FIGURES

Figure	Page
2-1 The Java translation and execution process.	6
3-1 Trace of naïve string matching algorithm.	14
3-2 String-matching automaton for “bcababab”.	17
3-3 Trace of string matching with finite automata.	17
3-4 Trace of pseudo KMP algorithm	20
3-5 Shift in the KMP algorithm	21
3-6 Trace of true KMP algorithm.	24
3-7 Bad character a does not appear in pattern P	26
3-8 Bad character a appears to the left of j in pattern P	26
3-9 Good-suffix shift that no u reappears anywhere in pattern P	28
3-10 Good-suffix shift that string u reappears in pattern P	29
3-11 Trace of Boyer-Moore algorithm.	32
4-1 The diagram of function areas.	34
4-2 Diagram of classes.	35
5-1 Impact of alphabet size on random string file with $lpl = 2$	41
5-2 Impact of alphabet size on random string file with $lpl = 4$	42
5-3 Impact of alphabet size on random string file with $lpl = 8$	43
5-4 Impact of alphabet size on random string file with $lpl = 16$	44
5-5 Impact of pattern length on English text file.	45

INTRODUCTION

With the popularity of computers, text is the main form for storing information. If the user wants to search for a particular text pattern he wants within a huge text file, then how can he do it efficiently? He searches from the raw text:

with the string matching problem, which finds one or more occurrences of a particular text in a text. An important issue in text editing and processing is efficient algorithms to improve the effectiveness of text-editing programs greatly. Shorter algorithms are more efficient components of most operating systems. They are

needed to search for particular patterns in files.

CHAPTER 1

1.1. Introduction 1

1.2. 1

1

animation system can be posted on the **CHAPTER I** Web (WWW) so that people who are interested in it can share it

This thesis is composed of **INTRODUCTION** current chapter is the introduction. The next chapter briefly introduces algorithm animation and the Java programming language. **With the popularity of computers, text is the main form for storing information. If the user wants to search for a particular text pattern he wants within a huge text file, then it would be very time consuming if he searches from the raw text.** Comparisons of these algorithms. **As a result, the string matching problem, which finds one or more occurrences of a pattern in a text, is a very important issue in text editing and processing. Efficient algorithms can improve the effectiveness of text-editing programs greatly. Therefore, string matching algorithms are basic components of most operating systems [4]. They are also used widely in the area of searching for particular patterns in DNA sequences [6].**

In this study, five important string matching algorithms are presented:

- The naïve string matching algorithm [6]
- String matching with finite automata [6]
- The pseudo-Knuth-Morris-Pratt (KMP) algorithm given in one text book [6]
- The true KMP algorithm [4]
- The Boyer-Moore algorithm [6]

The most interesting part in this study is that the comparison of these algorithms is based, not only on theoretical analysis, but also on visualization through algorithm animation by using Java. By the animated comparison of these algorithms, we can see how far the text pattern window moves against the text for each character comparison, and thus get an idea of how effective each algorithm is. In addition, this algorithm

animation system can be posted on the World Wide Web (WWW) so that people who are interested in it can share it.

This thesis is composed of six chapters. The current chapter is the introduction. The next chapter briefly introduces algorithm animation and the Java programming language. Chapter III presents and analyzes in details the five previously mentioned string matching algorithms. Chapter IV gives the descriptions of how the string matching algorithm animation system is designed and implemented. The comparisons of these algorithms are given in Chapter V. Last, Chapter VI summarizes the thesis and gives the suggested future work.

Technique in which the illusion of movement is achieved rapidly,

displaying a series of individual drawings that have slightly different stages

of a process. Animation uses a computer's processing power to create a variety

of algorithms that include generation of continuous movement, as

well as the ability to create a series of images that are displayed in a sequence

of frames, which are then displayed in a sequence, creating the illusion of

movement.

Figure 1.1: A sequence of frames from a 1990s cartoon.

2.2 ALGORITHM VISUALIZATION / CHAPTER II TION

Algorithm visualization consists of using visualization and animation techniques to help people understand algorithms work [14].

Algorithm animation is the dynamic form of algorithm visualization. It creates animated graphs. In this chapter, brief introductions of computer animation, algorithm animation and the reasons why Java is selected to implement this animation system are presented.

Such systems include Java, Matlab, Zeus,

2.1 COMPUTER ANIMATION

Animation is a technique in which the illusion of movement is created by rapidly displaying a series of individual drawings that have slightly different changes [13]. Computer animation uses a computer's processing power to encompass a variety of applications that include generation of drawings, movement, and color [13].

There are two major classifications in computer animation. One is computer-assisted animation (also referred to as key-framed animation) which is used to aid the traditional two-dimensional animation stages [2]. The other one is computer-generated animation (also referred to as modeled animation) used to create three-dimensional images and thus is more complex [2].

Today, computer animation techniques are used widely in various applications such as the visualization of mathematical and scientific data, and most sophisticated games [2]. Recently, a new domain of animation applications arose in a large public arena: more and more real-time animations are presented on the WWW [21].

2.2 ALGORITHM VISUALIZATION AND ANIMATION

Algorithm visualization consists of using visualization and animation techniques to help people understand how some sophisticated software algorithms work [14]. Algorithm animation is the dynamic form of algorithm visualization. It creates animated graphical views of the operations of an algorithm step by step [23]. Many systems have been developed for algorithm animation, a large number of which are described in the various taxonomies [16] [15] [18]. Such systems include Balsa, Balsa-II, Zeus, TANGO, Polka-3D, and ANIM.

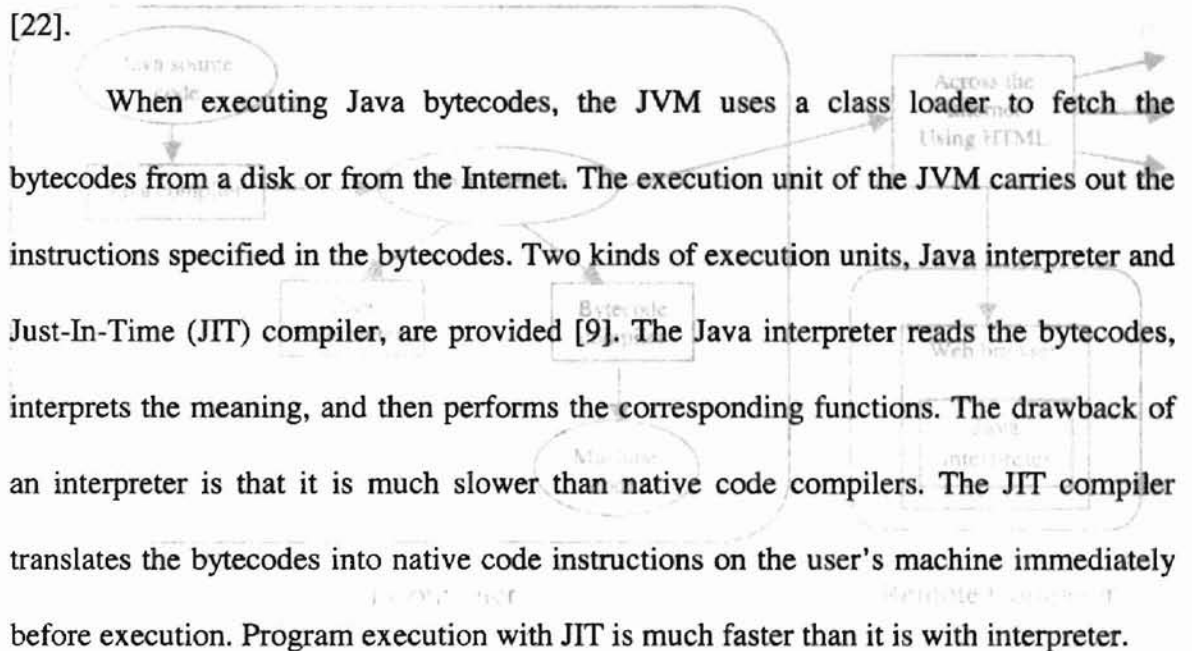
2.3 JAVA PROGRAMMING LANGUAGE

Java is an object-oriented programming language that was introduced by Sun Microsystems in late 1995. "Now Java is poised to take the rest of the programming world by storm" [7].

2.3.1 HOW JAVA WORKS

As in many other programming languages, Java uses a compiler to convert source code into executable programs. The difference is, the Java compiler generates architecture-independent bytecodes, which are the instructions that run on the Java Virtual Machine (JVM). In order to have Java programs run on a platform, that platform must have a JVM running on it [22]. JVM is a Java processor chip that is usually implemented in software rather than hardware [25]. Consequently, JVM can run not only on many types of computers (such as IBM-compatible PC, Macintosh, Unix workstation and server, and mainframe), but also on web browsers (such as Netscape Communicator,

Microsoft Explorer), which in turn run on top of varying operating systems and hardware [22].



When executing Java bytecodes, the JVM uses a class loader to fetch the bytecodes from a disk or from the Internet. The execution unit of the JVM carries out the instructions specified in the bytecodes. Two kinds of execution units, Java interpreter and Just-In-Time (JIT) compiler, are provided [9]. The Java interpreter reads the bytecodes, interprets the meaning, and then performs the corresponding functions. The drawback of an interpreter is that it is much slower than native code compilers. The JIT compiler translates the bytecodes into native code instructions on the user's machine immediately before execution. Program execution with JIT is much faster than it is with interpreter.

2.3.2 JAVA APPLICATIONS AND JAVA APPLETS

Two kinds of Java programs, Java applications and Java applets, are available. A Java application is a Java program that can be run without the use of a Web browser. It can be executed just using the Java interpreter. In contrast, a Java applet is a Java program that is embedded into an HTML document so that it can be executed using a Web browser. Java applets are considered to be a type of resources that can be shared through the Web just like text, graphics, and sound. Therefore, they can be reached anywhere in the world as long as there is a link to the Web page [12]. To illustrate how Java applications and Java applets work, the following figure is provided by Lewis and Loftus [12]:

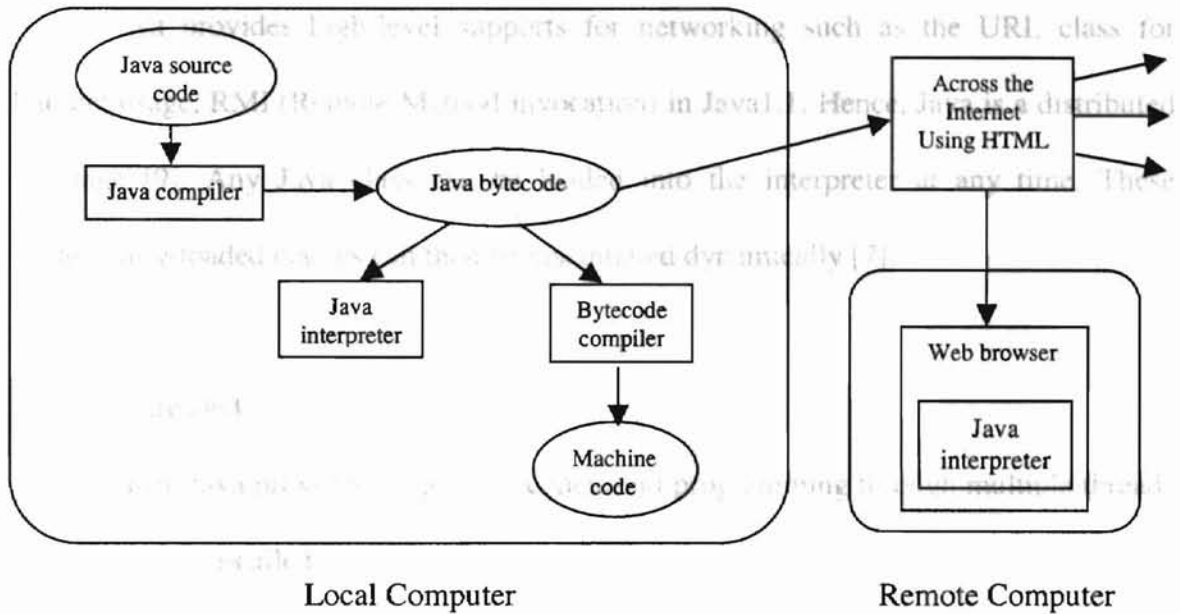


Figure 2-1 The Java translation and execution process

2.3.3 DESCRIPTION OF JAVA

The authors of Java described Java as a simple, object-oriented, distributed, robust, secure, architecture neutral, portable, interpreted, high-performance, multithreaded, and dynamic language [9].

- Architecture Neutral and Portable

Java programs are compiled to an architecture-neutral bytecode format. A Java application can run on any system, as long as that system implements the Java Virtual Machine. A standard bytecode format is defined, therefore Java is portable [9].

- **Distributed and Dynamic**

Java provides high-level supports for networking such as the URL class for Internet usage, RMI (Remote Method Invocation) in Java1.1. Hence, Java is a distributed language [9]. Any Java class can be loaded into the interpreter at any time. These dynamically-loaded classes can then be instantiated dynamically [7].

- **Multithreaded**

Since Java provides support for concurrent programming through multiple threads of execution, so-called lightweight processes, Java can handle different tasks in a very short interactive time. This feature improves the interactive performance of graphical applications for users [9].

- **Robust and Secure**

Java eliminates some types of programming errors such as pointer-related bugs. Also, it allows extensive compile-time checking to prevent potential type-mismatch problems. All of these make Java more robust than some other programming languages [9]. In a distributed environment, security is the key issue for any programming language. Java provides several layers of security controls to protect against malicious code. For example, the Java program is not allowed to run if it fails the bytecodes' check for the instructions that could make the underlying machine insecure [22].

String matching algorithms (CHAPTER III) (KMP, Moore Algorithm) in this study

of the text and pattern, then compare the characters of the text

STRING MATCHING ALGORITHMS

with the pattern. If the characters do not match, the pattern is shifted until the next match

As mentioned above, string matching algorithms are important in a lot of applications that we use everyday [24]. They are very useful in text editors and text retrieval tools. Moreover, string matching algorithms are used as part of a more complex algorithm, such as the Unix program “diff” that determines the differences between two similar text files [24].

3.1 DEFINITION

String matching finds one or more occurrences of a string, called the pattern, in a text [4]. Typically, the text is a document and the pattern is a particular word or phrase defined by the user [6]. In this study, the pattern is an array denoted by $P = P[1..m]$ with length m ; the text is also an array denoted by $T = T[1..n]$ with length n . The alphabet from which the elements of P and T are drawn is denoted by Σ .

The *String Matching Problem* is formally defined as follows [24]:

Given a text string T , with $|T| = n$, and pattern string P , with $|P| = m$, where $m, n > 0$ and $m \leq n$, if pattern P occurs as a sub-string of text T , then determine the position within the text of the first occurrence of pattern, i.e. return the least value of s such that $T[s+1..s+m] = P[1..m]$. The problem is extended such that the positions of all the occurrences of P within T are to be found.

String-matching algorithms (except the Boyer-Moore Algorithm) in this study first align the left ends of the text and pattern, then compare the characters of the text aligned with the characters of the pattern. After a whole match of the pattern or a mismatch, they *shift* the pattern to the right. This procedure is repeated until the right end of the pattern goes beyond the right end of the text. The pattern P occurs with shift s in text T if $T[s+1..s+m] = P[1..m]$ and $s \in [0, n-m]$. The shift s is *valid* if P occurs with shift s in T ; otherwise, it is *invalid*.

3.2 NOTATION AND TERMINOLOGY

- **CONCATENATION:** The *concatenation* of two strings x and y , denoted by xy , consists of the characters from x followed by the characters from y .
- **PREFIX:** A string u is a *prefix* of a string w if there exists a string v such that $w = uv$.
- **SUFFIX:** A string v is a *suffix* of a string w if there exists a string u such that $w = uv$.
- **EMPTY STRING:** The zero-length *empty string*, denoted by ϵ , is both a suffix and a prefix of every string.
- P_k denotes the prefix $P[1..k]$ of the pattern $P[1..m]$. Thus, $P_0 = \epsilon$ and $P_m = P = P[1..m]$.
- T_k denotes the prefix $T[1..k]$ of the text $T[1..n]$.
- **DETERMINISTIC FINITE STATE AUTOMATON (DFA) M** is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where:
 - Q is a finite set of states,
 - $q_0 \in Q$ is the start state,
 - $A \subseteq Q$ is a set of accepting states,

- Σ is a finite input alphabet;
- δ is a function from $Q \times \Sigma$ into Q , called the transition function of M .

3.3 OVERVIEW OF STRING MATCHING ALGORITHMS

The intuitive approach to the string matching problem is simply to match the pattern within the text at successive positions. A successful comparison of the text character and the pattern character causes both indices to be advanced by one; an unsuccessful comparison causes the text cursor to be backed up and the pattern cursor to be reset to one. This is the naïve string-matching algorithm. Its worst case behavior is $\Theta(mn)$, although it often takes only a few more than n comparisons in practice.

In early 1970s, Cook derived a theorem about two-way deterministic pushdown automata [5]. This led to the result that there exists an algorithm solving the string matching problem in $O(m+n)$ time in the worst case. Then Aho, Hopcroft, and Ullman discussed the relation of string matching to the theory of finite automata [1]. The searching of the pattern within the text takes only $\Theta(n)$ time after the automaton is built.

In 1970, Knuth derived a string matching algorithm which was modified by Pratt so that its running time was independent of the size of the alphabet. Almost at the same time, Morris invented the resulting algorithm independently. They published their work jointly in 1977. This is known as the KMP algorithm [11]. This algorithm avoids the backtracking in the text in the event of a mismatch by taking advantage of known information that is contained in the auxiliary *next* table. The algorithm performs $O(n + m)$ time in the worst case.

In 1974, Boyer and Moore discovered a much faster string matching algorithm. The algorithm was later published in a revised form in 1977, taking into account suggestions from Kuiper, Knuth and Floyd [3]. In this method, the character comparisons between pattern and text are performed from right to left for each attempted shift. The actual pattern shift is determined by taking the larger of values from two precomputed auxiliary tables, the *last-occurrence function* (bad-character heuristic) and *good-suffix-function* (good-suffix heuristic) [6]. The worst-case running time is $O((n-m+1)m + |\Sigma|)$.

Cormen, Leiserson, Rivest misinterpreted the KMP algorithm in their book "*Introduction to Algorithms*", in which they gave another much more complicated and less efficient string matching algorithm instead [6]. This algorithm is called pseudo-KMP algorithm in my study, in contrast to the true one. The pseudo-KMP algorithm often makes the same comparison of a pattern character to a text character more than once, thus making this method much slower. Also, the pseudo-KMP algorithm can make an occasional comparison that the true KMP algorithm never makes at all.

In addition, there are many string matching algorithms derived from the above mentioned algorithms. For example, Horspool gave a simplified form of Boyer-Moore algorithm, called the Boyer-Moore-Horspool algorithm, in 1980 [8]. Sunday developed three variations based on the Boyer-Moore algorithm, called the Quick Search algorithm, Maximal Shift algorithm, and Optimal Mismatch algorithm, in 1990 [19]. In 1987, Karp and Rabin also put forward an algorithm involving the use of hashing which reduces the task of comparing two strings to the simpler one of comparing two integers [10].

The string matching algorithms mentioned above can be classified depending on the order they perform the comparisons between text characters and pattern characters.

The most natural way to perform the comparisons is from left to right, which is the direction English speaking people read. The automaton method, the pseudo-KMP algorithm and true KMP algorithm belong to this category. Some algorithms perform the comparisons from right to left, such as the Boyer-Moore algorithm, which leads to a very efficient algorithm in practice. Finally there exist some algorithms for which the order in which the comparisons are done is not relevant; an example is the naïve string matching algorithm.

Takaoka mentioned in his article that most string matching algorithms work through two phases. First is the preprocessing phase that obtains the shift tables; the second phase is the main matching process against the text based on the tables [20]. In the first phase, there are two versions to compute the shift tables: the off-line version and the on-line version [20]. The off-line version computes the tables after the whole pattern has been input, while the on-line version computes the tables as the new character of the pattern is being input. Some algorithms such as KMP compute the shift table from left to right, and hence are suitable for the on-line version [20].

3.4 DESCRIPTION OF STRING MATCHING ALGORITHMS

In this study, five important algorithms are presented:

- (1) The naïve string-matching algorithm;
- (2) String matching with finite automata;
- (3) The pseudo-Knuth-Morris-Pratt (KMP) algorithm;
- (4) The true KMP algorithm;
- (5) The Boyer-Moore algorithm.

The naïve algorithm is first introduced, then how the other increasingly sophisticated algorithms improve the performance efficiency is discussed.

3.4.1 NAÏVE STRING MATCHING ALGORITHM

The naïve algorithm is the brute-force algorithm that finds all valid shifts by checking the condition $P[1..m] = T[s+1..s+m]$ for each of the $n-m+1$ possible values of s ($0 \leq s \leq n-m$). It shifts the pattern exactly one position to the right after a mismatch or a full match, and the next pattern character compared is always $P[1]$. The algorithm is shown as follows [6]:

NAÏVE-STRING-MATCHER (T, P)

```
1  n ← length [T];
2  m ← length [P];
3  for s ← 0 to n-m do
4      if (P[1..m] = T [s+1 ..s+m])
5          then print "Pattern occurs with shift" s;
```

Obviously, the time complexity of NAÏVE-STRING-MATCHER is $\Theta((n-m+1)m)$ in the worst case (when searching for a^m in a^n for instance). When $m = \lfloor n/2 \rfloor$, the worst case running time is $\Theta(n^2)$. The reason that the naïve string-matcher is inefficient is because information gained about the text for one value of s is ignored totally in considering other values of s . The advantages of this algorithm, however, are that no preprocessing phase and no extra space are needed. Also, the comparisons can be done in any order.

The following is a typical example that is examined by every algorithm described in this study. It gives the number of comparisons by each algorithm so that they can be compared.

Algorithm: Naïve String Matching Algorithm Text: b c a t c b c a b a b a b t a t a c a b t a c b Pattern: b c a b a b a b	
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4	b c a t c b c a b a b a b t a t a c a b t a c b
<pre> b c a b b b b b c a b a b a b b b b c b b c b b c b b b h (c a h a b a b) </pre>	
Number of matched string found: 1 Number of comparisons: 30 *Note: The characters in () do not perform the comparisons.	

Figure 3-1 Trace of naïve string matching algorithm

3.4.2 STRING MATCHING WITH FINITE AUTOMATA

In this algorithm, it is necessary to build a string-matching automaton for the pattern P in a preprocessing step before it can be used to search the text string. In order to construct the string-matching automaton, three functions are needed [6].

- The *transition function*, δ , is the mapping from $Q \times \Sigma$ to Q such that when the automaton is in state q and reads input character a , it moves from state q to state $\delta(q, a)$.
- The *final-state function*, ϕ , is the mapping from Σ^* to Q such that $\phi(w)$ is the state the automaton M finishes scanning the string w . Therefore, the automata M accepts a string w if and only if $\phi(w) \in A$.
- The *suffix function*, σ , is a mapping from Σ^* to $\{0, 1, \dots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x . This means we need to find out the longest suffix of x that is also a prefix of P .

Based on these functions, the string-matching automaton of a given pattern $P[1..m]$ is constructed as follows [6]:

- (1) The Q is the set $\{0, 1, \dots, m\}$ with $m+1$ states. The *start state* q_0 is state 0, and the only *accepting state* is state m .
- (2) For any state q and character a , the transition function δ is defined as: $\delta(q, a) = \sigma(P_q a)$.

For every finite automaton, we can construct a two-dimensional table to represent the transition function δ . The following is the procedure that computes the transition function δ for a given pattern $P[1..m]$ [6]:

COMPUTE-TRANSITION-FUNCTION (P, Σ)

```

1   m ← length[P]
2   for q ← 0 to m do
3       for each character a ∈ Σ do
4           k ← min(m+1, q+2)
```

```

5      repeat k ← k-1      //find the largest k
6          until Pk is the suffix of Pqa
7      δ(q, a) ← k      //δ(q, a) = σ(Pqa)
8      return δ

```

Since the outer loops of line 2 and 3 takes time $m|\Sigma|$, the inner loop of line 5 runs $m+1$ times in the worst case, and the test on line 6 must compare up to m characters, the time complexity of this procedure is $O(m^3|\Sigma|)$. This means it would take a lot of time to build the automaton if Σ is large. On the other hand, it needs $\Theta((m+1)|\Sigma|)$ extra space for the storage of the transition table.

Once the automaton is built, searching for the pattern in a text consists in parsing the text with the automaton beginning with the initial state. Each time the final state is encountered, an occurrence of the pattern has been found. So each character in the text is examined only once. Therefore, the searching phase takes only $\Theta(n)$ time. The algorithm is shown as follows [6]:

FINITE-AUTOMATON-MATCHER(T, δ , m)

```

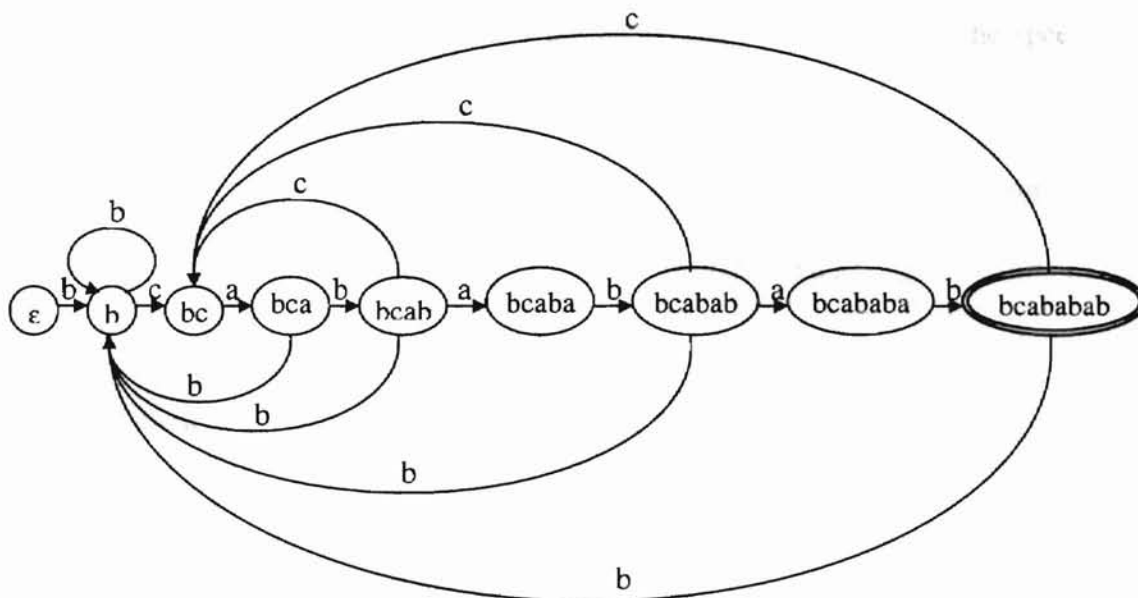
1      n ← length[T]
2      q ← 0
3      for i ← 1 to n do
4          q ← δ (q, T[i])      //use T[i] as an index into transition table
5          if q = m      //find one occurrence
6              s ← i-m
7      print "Pattern occurs with shift" s

```


The following is automaton based on the example mentioned above:

$\Sigma = \{a, b, c, t\}$

$Q = \{0 (\epsilon), 1 (b), 2 (bc), 3 (bca), 4 (bcab), 5 (bcaba), 6 (bcabab), 7 (bcababa), 8 (bcababab)\}$



Missing transitions are leading to the initial state

Figure 3-2 String-matching automaton for "bcababab"

Having the above automaton at hand, it is very easy to search within the text. The

Figure 3-3 is the trace of the operation on the text T= "bcatcbcabababtatacabtab".

Algorithm: String matching with finite automata	
Text: b c a t c b c a b a b a b t a t a c a b t a c b	
Pattern: b c a b a b a b	
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4	
Text: b c a t c b c a b a b a b t a t a c a b t a c b	
state: 1 2 3 0 0 1 2 3 4 5 6 7 8 0 0 0 0 0 0 0 1 0 0 0 1	
Number of matched string found: 1	
Number of comparisons: 24	

Figure 3-3 Trace of string matching with finite automata

3.4.3 THE PSEUDO-KNUTH-MORRIS-PRATT (KMP) ALGORITHM

The brute force algorithm shifts only one position after each comparison of the pattern and text. Actually, it is possible to improve the length of shifts by remembering some portions of the text that match the pattern. This can save comparisons between characters of the text and characters of the pattern, and thus increases the speed of the search.

The pseudo-KMP algorithm uses an auxiliary function $\pi[1..m]$ precomputed from the pattern to gain the information about how the pattern matches against shifts of itself. This auxiliary function is called the *prefix function*. The prefix function for the pattern P is the π function that maps $\{1, 2, \dots, m\}$ to $\{0, 1, \dots, m-1\}$ such that $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q . For example, for the pattern $P = bcababab$, the π function is computed as follows:

i	1	2	3	4	5	6	7	8
P[i]	b	c	a	b	a	b	a	b
$\pi[i]$	0	0	0	1	0	1	0	1

Table 3-1 The prefix function for $P = bcababab$

The following procedure is used to obtain the prefix function [6]:

COMPUTE-PREFIX-FUNCTION(P)

- 1 $m \leftarrow \text{length}[P]$
- 2 $\pi[1] \leftarrow 0$
- 3 $k \leftarrow 0$

```

4   for q ← 2 to m do
5       while k > 0 and P[k+1] ≠ P[q] do
6           k ← π[k]
7       if P[k+1] = P[q]
8           then k ← k+1
9       π[q] ← k
10  return π

```

From this procedure, we can see that the function π is an one-dimensional array that has only m entries, which means it needs $\Theta(m)$ extra space. The running time of this procedure is $O(m)$.

Assume in a shift s , the first q characters in the pattern match the text characters ($P[1..q] = T[s+1..s+q]$). We can use the π function to look up the value of $\pi[q]$. Since $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q , the next potential valid shift s' should be $s+(q-\pi[q])$. At the new shift s' , there is no need to compare the first $\pi[q]$ characters of P with the corresponding characters of T since they are guaranteed to be equal by the π function. The following is the PSEUDO-KMP-MATCHER [6]:

PSEUDO-KMP-MATCHER (T, P)

```

1   n ← length[T]
2   m ← length[P]
3   π ← COMPUTE-PREFIX-FUNCTION(P)
4   q ← 0

```

```

5   for i ← 1 to n do
6       while q > 0 and P[q+1] ≠ T[i] do
7           q ← π[q]
8       if P[q+1] = T[i]
9           then q ← q+1
10      if q = m
11          then print "Pattern occurs with shift" i - m
12          q ← π[q]

```

The searching phase can be performed in $O(n+m)$ in the worst case.

Algorithm: The pseudo-KMP algorithm	
Text: b c a t c b c a b a b a b t a t a c a b t a c b	
Pattern: b c a b a b a b	
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4	b c a t c b c a b a b a b t a t a c a b t a c b
b c a b	b
	b
	b c a b a b a b
	(b) c
	b
	b
	b (c a b a b a b)
Number of matched string found: 1	
Number of comparisons: 28	
*Note: The underscored characters perform comparison twice.	
The characters in () do not perform the comparison.	

Figure 3-4 Trace of pseudo-KMP algorithm

3.4.4 THE TRUE KMP ALGORITHM

The true KMP algorithm improves the performance of PSEUDO-KMP-MATCHER by avoiding making the same comparison of a pattern character to a text character more than once. In PSEUDO-KMP-MATCHER, if the first q characters in the pattern match the text characters ($P[1..q] = T[s+1..s+q] = u$), then we shift by $q - \pi[q]$ for the next potential match. Now, we know that the first mismatch occurs between the character $A = T[s+q+1]$ and the character $B = P[q+1]$. It is possible to avoid another immediate mismatch by checking whether the character C , which follows the longest prefix of P that is also a proper suffix of P_q (denoted by v), is the same as the character B . If $C = B$, it means C does not match A either. Therefore, it is unnecessary to compare these two characters (A and C). Such prefix v is called the *border* of u since it occurs at both ends of u . The above can be illustrated in Figure 3-5.

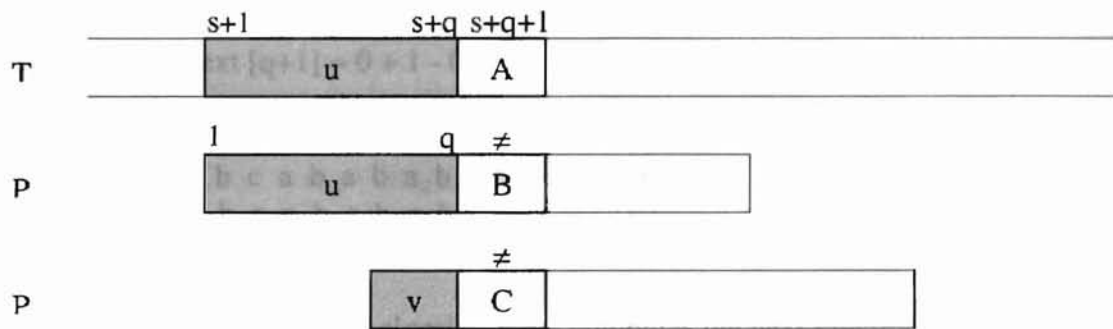


Figure 3-5 Shift in the KMP algorithm (v border of u and $C \neq B$)

Therefore, we introduce a *next* function of which $next[q]$ is the length of the longest border of $P[1..q]$ followed by a character different from $P[q+1]$. Then, after a shift, the comparisons can resume between characters $T[s+q+1]$ and $P[q+1 - next[q+1]]$

without missing any occurrence of pattern in the text, and avoiding a backtrack on the text. Table 3-2 is the next function for $P = bcababab$.

i	1	2	3	4	5	6	7	8	9
P[i]	b	c	a	b	a	b	a	b	
next[i]	0	1	1	0	2	0	2	0	2

Table 3-2 The next function for $P = bcababab$

If the text is $T = bc at c b c a b a b a b t a t a c a b t a c b$, then

First try:

$T = bc at c b c a b a b a b t a t a c a b t a c b$

≠

$P = bc a b a b a b$

Shift by $q+1 - next[q+1] = 3 + 1 - 0 = 4$

Second try:

$T = bc at c b c a b a b a b t a t a c a b t a c b$

≠

$P = bc a b a b a b$

Shift by $q+1 - next[q+1] = 0 + 1 - 0 = 1$

Third try:

$T = bc at c b c a b a b a b t a t a c a b t a c b$

$P = bc a b a b a b$

Shift by $q+1 - next[q+1] = 8 + 1 - 2 = 7$, and so on.

The following is the algorithm that computes the next function [24]:

COMPUTE-NEXT (P, next)

1 $m \leftarrow \text{length}[P]$

2 $j \leftarrow 1$

3 $t \leftarrow 0$

4 $next[1] \leftarrow 0$

```

5   while j < m do
6       while t > 0 and P[j] ≠ P[t] do
7           t ← next[t]
8       t ← t+1
9       j ← j+1
10      if P[j] = P[t]
11          then next[j] ← next[t]
12      else
13          next[j] ← t
14  return next

```

The table next can be computed in $O(m)$ time. The following algorithm is used for the searching phase [24]:

TRUE-KMP (T, P)

```

1   n ← length [T]
2   m ← length [P]
3   i ← 0
4   j ← 0
5   while k ≤ n do
6       while j > 0 and P[j] ≠ t[k] do
7           j ← next[j]
8       k ++
9       j ++

```

10 if $j \geq m$ then

11 OUTPUT(k-j)

12 j = next[j]

Algorithm: The true KMP algorithm	
Text: b c a t c b c a b a b a b t a t a c a b t a c b	
Pattern: b c a b a b a b	
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4	b c a t c b c a b a b a b t a t a c a b t a c b
b c a b	
	b
	b c a b a b a b
	(b) c
	b
	b
	b
	b (c a b a b a b)
Number of matched string found: 1	
Number of comparisons: 18	
*Note: The characters in () do not perform the comparisons.	

Figure 3-6 Trace of true KMP algorithm

Notice that the comparison P[1] and T[4] is made by the pseudo-KMP algorithm while it is not made by the true KMP. This is because after true KMP checks $P[4] \neq T[4]$, it knows $P[1] = P[4]$. So it is unnecessary to compare P[1] and T[4] since they are guaranteed to be unequal.

3.4.5 THE BOYER-MOORE ALGORITHM

By using the Boyer-Moore algorithm, the pattern is scanned across the text from left to right, but the actual comparisons between the pattern characters and the text characters are performed from right to left. In case of a mismatch or a full match, it uses two heuristics to shift the pattern to the right, allowing it to avoid many examinations of the text characters. These two heuristics are called the *bad-character heuristic* and the *good-suffix heuristic*. When a mismatch occurs, the Boyer-Moore algorithm chooses the larger value from the shift amounts proposed by these two heuristics without missing any occurrence of pattern in the text.

3.4.5.1 The bad-character heuristic

Assume that a mismatch occurs between the character $T[s + j] = a$ of the text and the character $P[j] = b$ of the pattern at the shift s . This means $T[s + j + 1.. s + m] = P[j + 1.. m] = u$ and $T[s + j] \neq P[j]$ for some j , where $1 \leq j \leq m$. Then $T[s + j]$ is the bad character. Let k be the largest index in the pattern such that $P[k]$ is the rightmost occurrence of the bad character $T[s + j]$ in the pattern, if any such k exists. Otherwise, $k = 0$. Then the bad character heuristic proposes to shift the pattern to right by $j - k$. There are three cases we need to consider [6].

- $k = 0$

If k is equal to 0, this means the bad character $T[s + j] = a$ doesn't occur in the pattern at all, as shown in Figure 3-7. So the next shift increases the current shift s by j without missing any valid shifts.

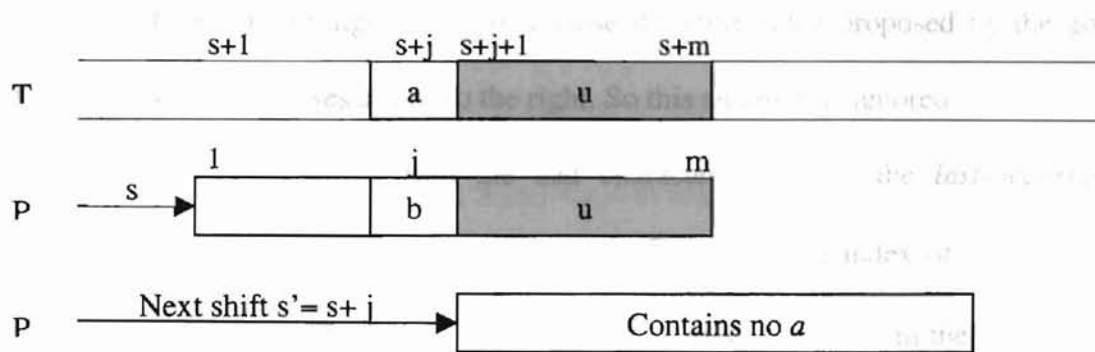


Figure 3-7 Bad character a does not appear in pattern P

- $k < j$

If the rightmost occurrence of the bad character $T[s + j] = a$ in the pattern is to the left of position j , then the value $j - k$ is positive ($j - k > 0$) and the pattern advances $j - k$ positions to the right before the bad character matches any pattern character. This situation is shown as Figure 3-8.

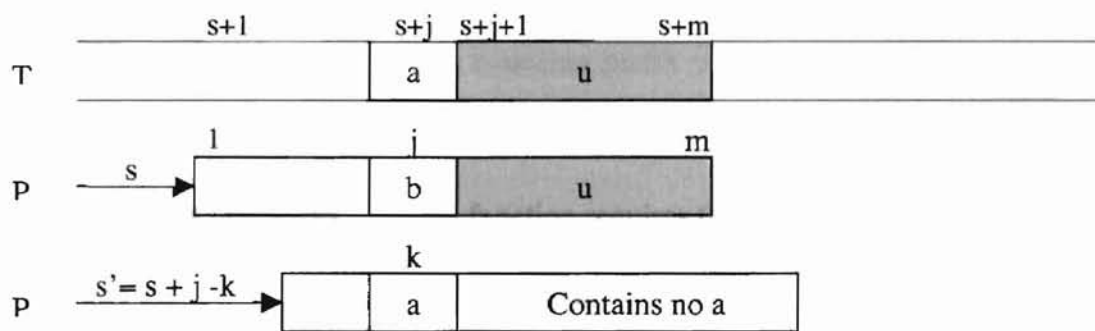


Figure 3-8 Bad-character a appears to the left of j in pattern P

- $k > j$

If the rightmost occurrence of the bad character $T[s + j] = a$ in the pattern is at position $k > j$, then $j - k < 0$, which means the bad-character heuristic proposes to decrease

s. Then the Boyer-Moore algorithm will choose the shift value proposed by the good-suffix heuristic that proposes a shift to the right. So this situation is ignored.

Formally the function from the bad character heuristic, the *last-occurrence function* (λ), is defined as follows. For each $a \in \Sigma$, $\lambda(a)$ is the index of the rightmost position in the pattern at which character a occurs. If a does not occur in the pattern, then $\lambda(a)$ is equal to 0. The following algorithm is used to compute the *last-occurrence function* [6]:

COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, S)

```
1   for each character  $a \in \Sigma$ 
2        $\lambda[a] = 0$ 
3   for  $j = 1$  to  $m$ 
4        $\lambda[P[j]] = j$ 
5   return  $\lambda$ 
```

The λ function needs space of size $|\Sigma|$ and the running time for this procedure is obviously $O(|\Sigma| + m)$. Notice that this function requires that each character be available as a numerical value for indexing into the λ table. The methods previously discussed never required this, only needing a verdict of “equal” or “not equal” for each pair of characters compared.

3.5.4.2 The good-suffix heuristic

Similarly, suppose a mismatch occurs between the character $T[s + j] = a$ of the text and the character $P[j] = b$ of the pattern at the shift s , where $j < m$. This means

$T[s+j+1.. s + m] = P[j + 1.. m] = u$ and $T[s + j] \neq P[j]$. The function for the good-suffix heuristic, the *good-suffix function* (γ), is defined as follows:

$$\gamma[j] = m - \max (\{\pi [m]\} \cup \{k: \pi [m] < k < m \text{ and } P[j+1 .. m] \text{ is a suffix of } P_k \text{ and } j - (m - k) > 0 \text{ implies } P[j] \neq P[j - (m - k)]\}).$$

The good-suffix heuristic proposes to increase the current shift s by $\gamma[j]$. The above formula is constructed based on two situations described as follows.

- No string u reappears in the pattern

If the string u does not reappear anywhere in the pattern, then we can use the prefix function π mentioned in the pseudo-KMP algorithm to compute the next shift. In Figure 3-9, string v is the longest suffix of P_m that is also a prefix of the pattern. The length of string v is $\pi[m]$ by the definition of π . For the next shift, it aligns the longest suffix v of string u in the text with a matching prefix of the pattern. This amount of the next shift is increased by $\gamma[j] = m - \pi[m]$, which is the largest value that we can advance the current shift s safely. (This means $\gamma[j] \leq m - \pi[m]$.)

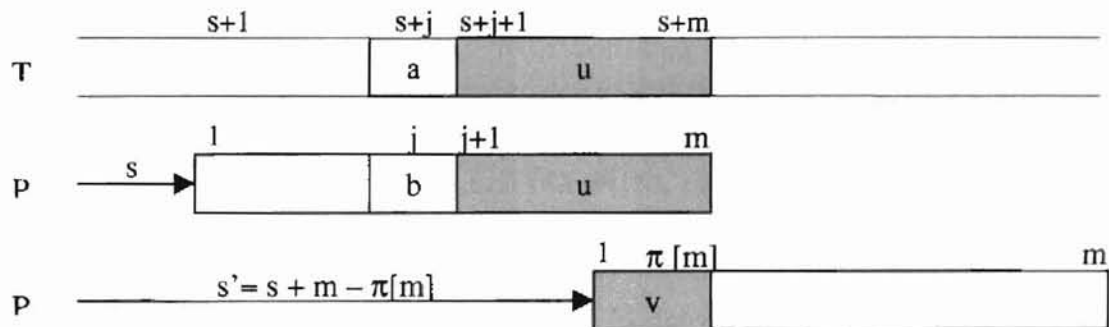


Figure 3-9 Good-suffix shift that no u reappears anywhere in pattern P

- String u reappears in the pattern preceded by a character different from b

As shown in Figure 3-10, if the string u reappears in the pattern preceded by a character different from b , then the next potential match will occur by aligning the string u in the text with its rightmost occurrence in the pattern. Since $P[j+1 ..m] = u$ is the suffix of P_k , this implies that $k > \pi[m]$. The increased amount of the next shift proposed by the good-suffix function is $\gamma[j] = m - k$.

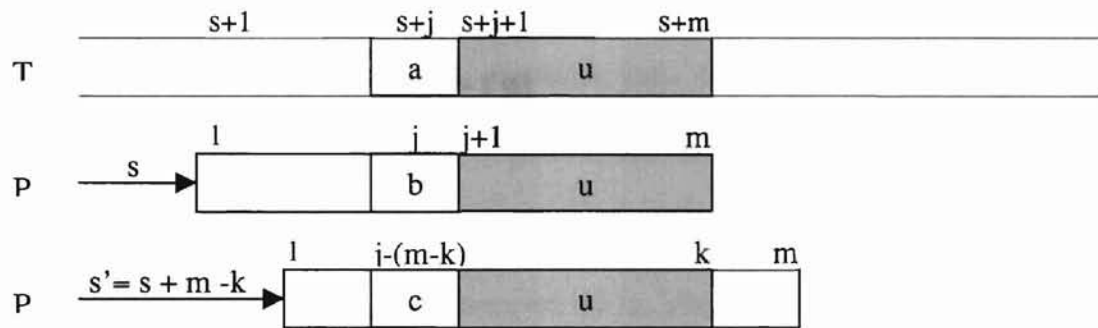


Figure 3-10 Good-suffix shift that string u reappears in pattern P

The following procedure is to compute the good-suffix function, whose running time is $O(m)$ [4].

COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

```

1   for q = 0 to m
2        $\gamma[q] = 0$ 
3   f[m] = m + 1
4   j = m + 1
5   for (i = m; i > 0; i--)
6       while (j ≤ m and P[i] ≠ P[j]) {
```

```

7         if ( $\gamma[j] = 0$ ) then  $\gamma[j] = j - i$ 
8         j = f[j] }
9     j = j - 1
10    f[i - 1] = j
11    t = f[0]
12    for (j = 0; j ≤ m; j++)
13        if ( $\gamma[j] = 0$ ) then  $\gamma[j] = t$ 
14        if (j = t) then t = f[t]
15    return  $\gamma$ 

```

3.5.4.3 The Boyer-Moore matcher

After computing two functions, the last-occurrence function and the good-suffix function, for the pattern, it is very efficient to search the text by using the procedure Boyer-Moore Matcher [6].

BOYER-MOORE-MATCHER (T, P, S)

```

1    n = length[T]
2    m = length[P]
3     $\lambda$  = COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m,  $\Sigma$ )
4     $\gamma$  = COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
5    s = 0
6    while s ≤ n - m
7        j = m

```

```

8         while j > 0 and P[j] = T [s + j]
9             j = j - 1
10        if j = 0
11            print "Pattern occurs at shift" s
12            s = s +  $\gamma[0]$ 
13        else
14            s = s + max ( $\gamma[j]$ , j -  $\lambda[T[s + j]]$ )

```

Apparently, the Boyer-Moore algorithm runs $O(nm)$ in the worst case when searching for all of the occurrences of a pattern. However, when searching for a^m in b^n , the algorithm makes only $O(n/m)$ comparisons, since each text character examined yields a mismatch, thus causing each shift increased by m . Therefore, the best case behavior of the Boyer-Moore algorithm is sub-linear.

We should not be too disappointed in the worst case behavior of the Boyer-Moore algorithm, since Rivest demonstrated that, in the worst case, any string matching algorithm must examine at least $n - m + 1$ symbols for the text string [17]. This shows that no solution to the string matching problem may have sub-linear behavior in n in the worst case.

The followings are the illustrations of how the Boyer-Moore algorithm works based on the example for $T = \text{bcabcabababtatatabctac}$ and $P = \text{bcababab}$. Table 3-3 is the last-occurrence function (λ) and Table 3-4 is the good-suffix function (γ) for the pattern. Figure 3-11 is the trace of the operation of the algorithm based on the two pre-computed functions λ and γ .

Σ	a	b	c	t
$\lambda[]$	7	8	2	0

Table 3-3 The last-occurrence function λ for $P = bcababab$

j	0	1	2	3	4	5	6	7	8
P[j]		b	c	a	b	a	b	a	b
$\gamma[j]$	7	7	7	7	2	7	4	7	1

Table 3-4 The good-suffix function γ for $P = bcababab$

Algorithm: The Boyer-Moore algorithm Text: bc at c b c a b a b a b t a t a c a b t a c b Pattern: b c a b a b a b	
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4	b c a t c b c a b a b a b t a t a c a b t a c b
(b c a b a b a) b	(b c a b a) b a b
	b c a b a b a b
	(b c a b a) b a b
	(b c a b a b) a b
Number of matched string found: 1 Number of comparisons: 17 *Note: The characters in () do not perform the comparisons.	

Figure 3-11 Trace of Boyer-Moore algorithm

ALGORITHM ANIMATION DESIGN AND IMPLEMENTATION

The Java Programming Language provides many powerful functions that greatly benefit the implementation of this string matching animation system. Because of the capability of platform independence, this animation system can run on any machine and Internet browser that has a Java byte-code interpreter. Therefore, the end users would never worry about those annoying problems which always occur in compiling and system configuration any more. The easily implemented Java thread mechanism enhances the power of this animation system to demonstrate multiple algorithms in parallel. At the beginning of this chapter, a diagram of functionality is given, which describes the implementation by the function areas. The enumerated classes diagram is given to overview the hierarchy of this Object-Oriented design. Then several major classes and their methods, as well as the relationship among them, are presented. Some implementation details are also discussed. The exception check is also mentioned briefly.

4.1 THE OVERVIEW OF FUNCTION AREAS

This system consists of six components/modules based on their functionality. The system functionality diagram overviews the entire system in Figure 4 -1. When a user clicks a button or types to input a string on the GUI, the event is captured by module StringMatcher Applet and is transferred to module ThreadRunner. The ThreadRunner determines which corresponding method in the module AnimationThread should be used

to process this event. The AnimationThread provides several animation methods and thread control methods, which make animation execute simultaneously. In the AnimationThread, some actions related to the animation stage are sent to module Canvas/Scribble, which sets up and refreshes the animation background. The module Algorithm controls the algorithms' animation in the GUI directly, by invoking its own methods or calling the methods in AnimationThread.

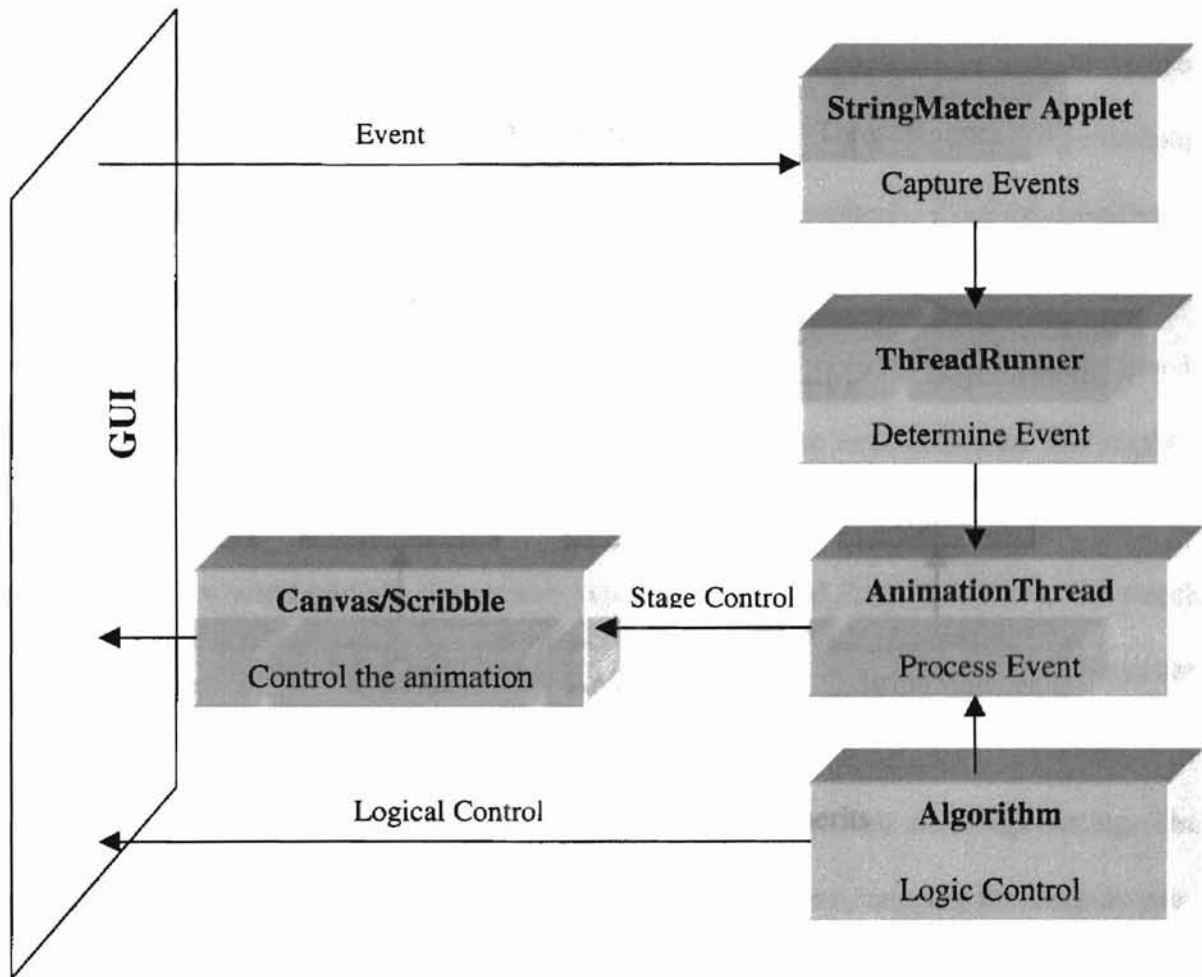


Figure 4 -1 The diagram of function areas

4.2 SYSTEM CLASS HIERARCHY

Based upon the Object Oriented Design methodology, there is one base classes in the system, GenericMatch, as well as five single classes, ExceptionCheck, Scribble, AnimationTread, StringMatcher and ThreadRunner, with no child class. The hierarchy of classes is shown in Figure 4 -2.

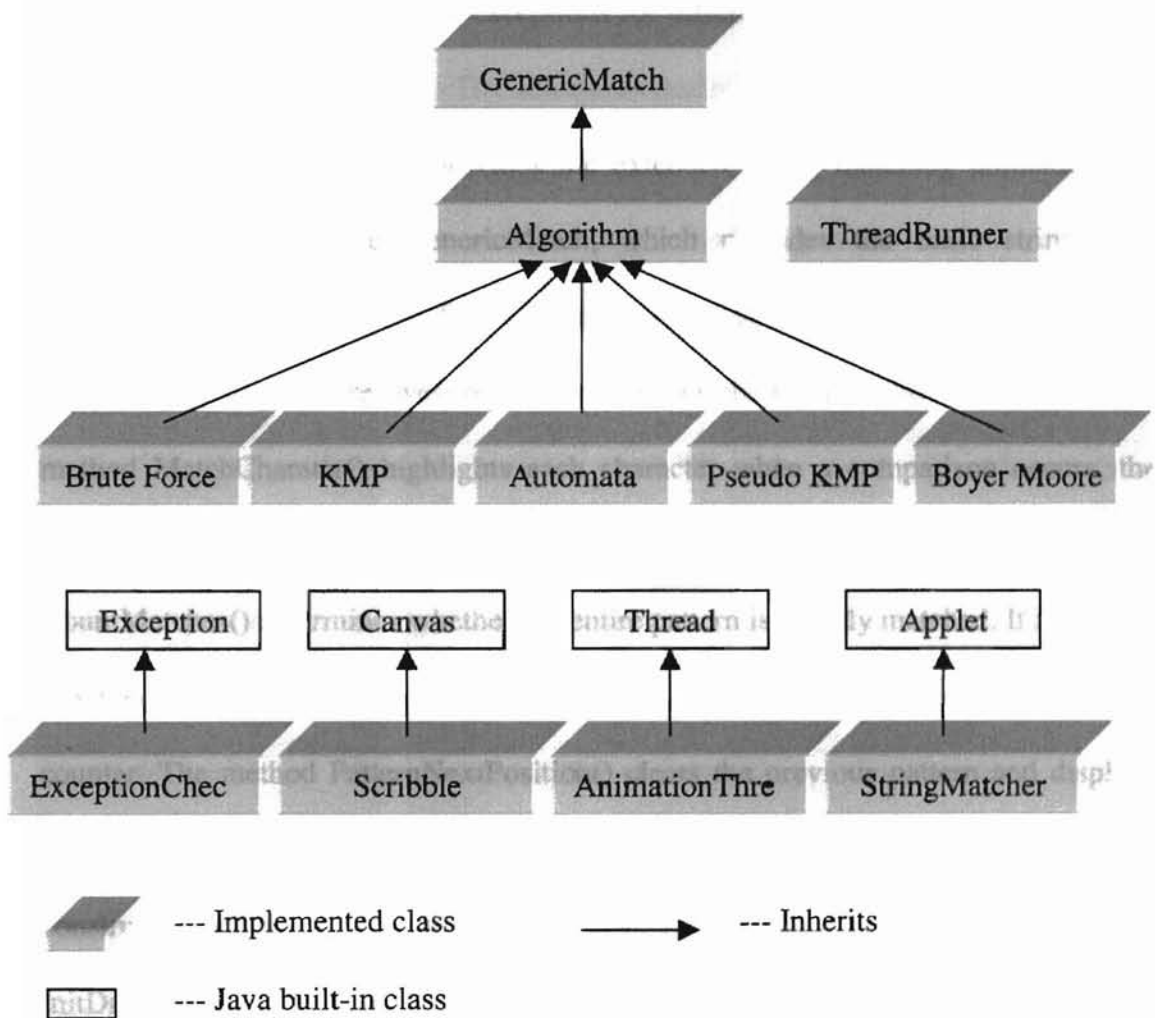


Figure 4 -2 Diagram of classes

The largest part in this diagram is class `GenericMatch` and its subclasses. The purpose of `GenericMatch` is to define several prototype methods for its subclasses. The basic string-matching behavior can be considered as matching every character between text and pattern, moving the pattern to the next position, counting how many characters match for each pattern movement. Therefore, three methods --- `MatchCharacter()`, `PatternNextPosition()` and `CountMatches()` --- are employed to implement the above steps. These three methods also throw exceptions for different kind of situations, like index of pattern or text-out-of-bounds.

The class `Algorithm` provides the extended string matching animation control interface. It extends the `GenericMatch`, which provides the basic string-matching functionality as mentioned before. The methods `MatchCharacter()`, `CountMatches()` and `PatternNextPosition()`, are overloaded based on the their super class `GenericMatch`. The method `MatchCharacter()` highlights each character when a comparison occurs, then it clears the highlight color and moves the comparison to the next character. The method `CountMatches()` determines whether the entire pattern is exactly matched. If it is matched, this method would highlight the pattern inside the text, and then adds one to the match counter. The method `PatternNextPosition()` clears the previous pattern and displays the pattern in the current position; the current position is passed in as a parameter. The constructor defines and initializes the data related to color, font and stage setting. The `InitDraw` method, which is not inherited from its super class, triggers the built-in class `Canvas` to draw the initial text and pattern string in each animation area. The method `ClearArea()` is responsible for cleaning everything up in the animation area for the next animation.

Five algorithms are implemented in this animation system. Every algorithm is a class that is derived from its super class Algorithm. Each algorithm calls CountMatches(), MatchCharater() or PatternNextPosition() in its logical path to animate the movement of every string matching step and action.

The design approach of this animation system is very clear in that it encapsulates the animation control methods as perfectly as possible. As a result, the animated algorithms do not have to know how the animation control methods work, just what the animation control methods do. The animation control methods look like puppets and the algorithms can be considered as a puppet master.

The animation stage, or so-called animation background, needs some methods to manipulate. There are several methods in class Scribble. The Scribble class creates a panel and two scrollbars, and lays them out on the panel. The method BorderLayout flushes the right and bottom sides of the panel. When the panel grows, the scrollbars of the panel also grows appropriately. The method Paint redraws everything in the panel when the window containing the panel loses focus or is resized. Otherwise, everything will disappear.

In order to demonstrate different algorithms simultaneously, the class AnimationThread is employed to create several threads for different algorithms. Then it manipulates the start and end of threads by using methods Run() and Finish(). Since the algorithms in the threads run in parallel, users can get a vivid comparison in the speed and behavior of algorithm comparison, and this greatly enhances the power of this animation system.

The class `ThreadRunner` provides several methods connected to the GUI; that is, when users click the button, the corresponding method is triggered. For example, when the user pushes the “Go” button, the method `Play()` is triggered and its threads start to run.

Like the `main()` function in C/C++, the `StringMatcher` extends the built-in class `Applet` and performs similarly as a main function. First, the method `Initialize()` calls several methods to draw animation control buttons on the panel and initializes several parameters, such as the positions and colors. Then the method `Action()` is waiting for any event such as a click on button or pushdown list. When a click event occurs, the method `ListClick` or `ButtonClick` is invoked and determines what method should be called to process such a click. The `StringMatcher` applet is an event-driven program waiting to react to any event.

COMPARISON OF STRING MATCHING ALGORITHMS

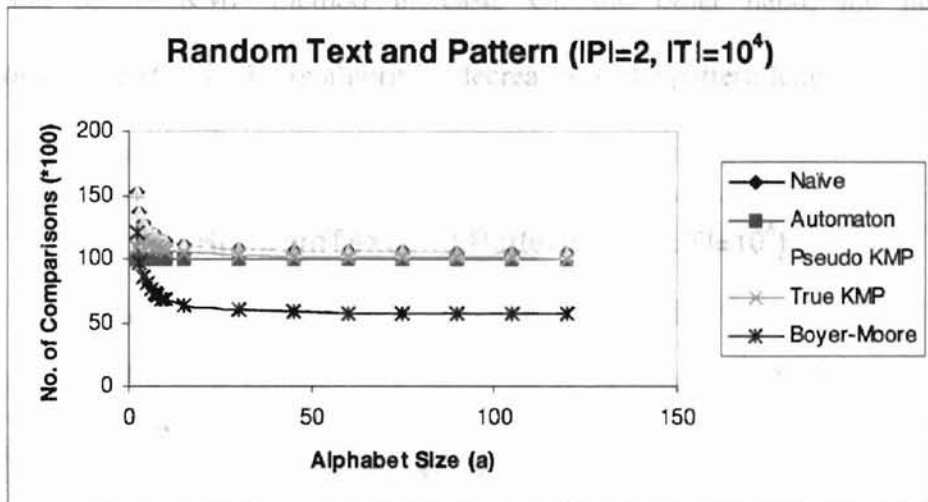
The following is the empirical evaluation of five string matching algorithms under three circumstances: searching within a random string file, English text file and binary string file. In the first category, the impact of the alphabet size and pattern length on the number of comparisons for each algorithm is examined. For the latter two categories, since their alphabet sets are relatively fixed, only the impact of the pattern length on the number of comparisons for each algorithm is analyzed. In order to obtain the correct result, the text length is chosen to be about 10,000 so that these algorithms can be fully examined. Each data point shown on the plot is the result of the average number of running the program 100 times.

As mentioned above, with the finite automaton method, once the automaton is built, the time used to search the text is only $\Theta(n)$, because with the automaton, each text character needs to be examined exactly once to determine the state it reaches. So, if the text length is 10,000, the number of comparisons of this method during the searching phase is fixed --- exactly 10,000 comparisons in the searching phase. Therefore, neither the pattern length nor the alphabet size will have any impact on its performance during the searching phase (although these two factors play key roles during its preprocessing phase). For this reason, the automaton method is ignored in this chapter. But in the next chapter of conclusions, where the preprocessing phase is also taken into account, this method is reevaluated.

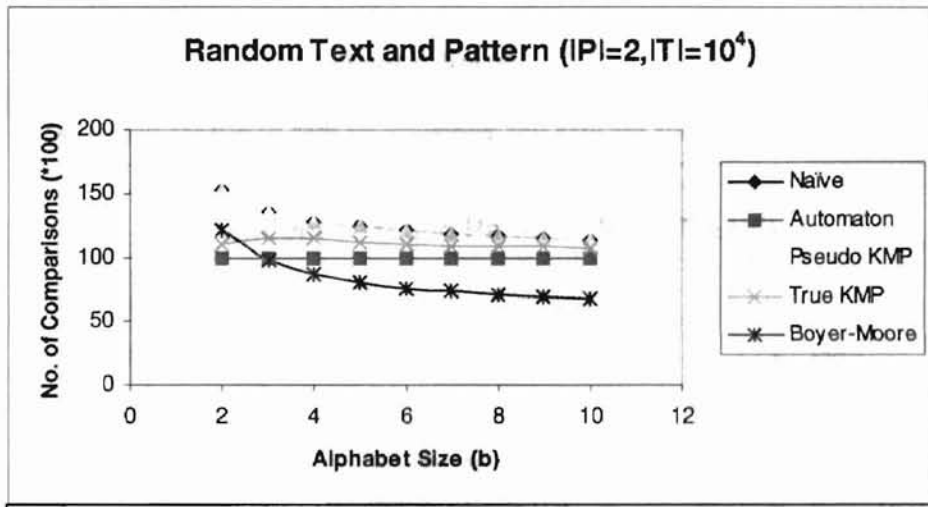
5.1 RANDOM TEXT AND PATTERN

For the texts and patterns whose characters are chosen independently from the alphabet with an uniform probability distribution, the algorithms may have different performances. In order to analyze the impact of the size of alphabet set on the number of comparisons, the length of pattern used to search the text should be the same. Here, four pattern lengths are chosen to show the typical performance of these five algorithms: $|P| = 2$, $|P| = 4$, $|P| = 8$, $|P| = 16$.

Figure 5-1 shows that with the pattern length = 2, the number of comparisons of all these algorithms (except the automata method) decreases when the alphabet size becomes larger. When the alphabet size reaches about 25, it has no obvious impact on the number of comparisons of any algorithm. Among these algorithms, Boyer-Moore is the best. Actually, the pseudo-KMP algorithm decreases to the naïve algorithm as the alphabet becomes larger. The Boyer-Moore approach works much better than the true KMP with a larger alphabet size. The true KMP, however, has better performance than Boyer-Moore when the alphabet size is small. In order to examine the impact of small alphabet size on the performance, it is necessary to amplify the portion of the plot with small alphabet size, as shown in Figure 5-1b. From this experiment, it can be seen that only when the alphabet size is larger than about 3, the Boyer-Moore method outperforms the true KMP algorithm. When the alphabet size gets larger, the non-backtracking KMP algorithm provides no significant speed advantage over the naïve method.



Note: The sequence of lines from top to bottom: Pseudo KMP, Naive, True KMP, Automaton and Boyer-Moore. Pseudo KMP, Naive, True KMP and Automaton are close.

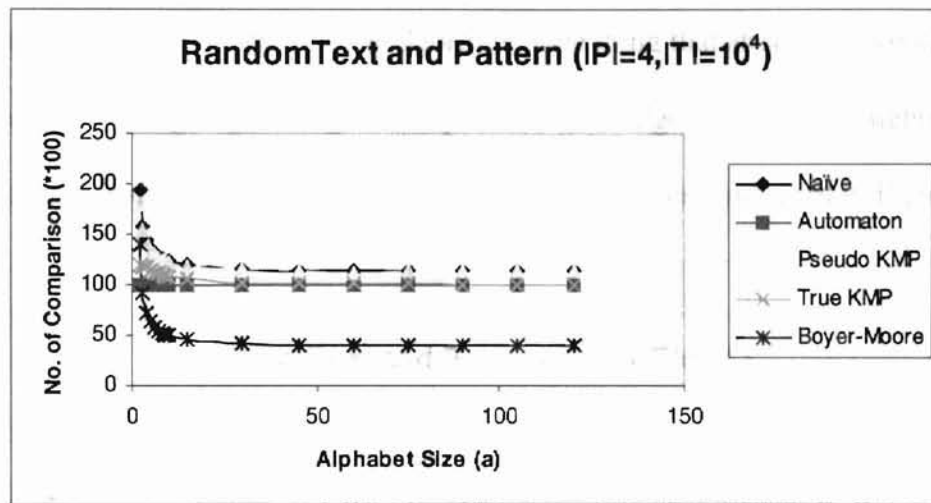


Note: Naive and Pseudo KMP are close.

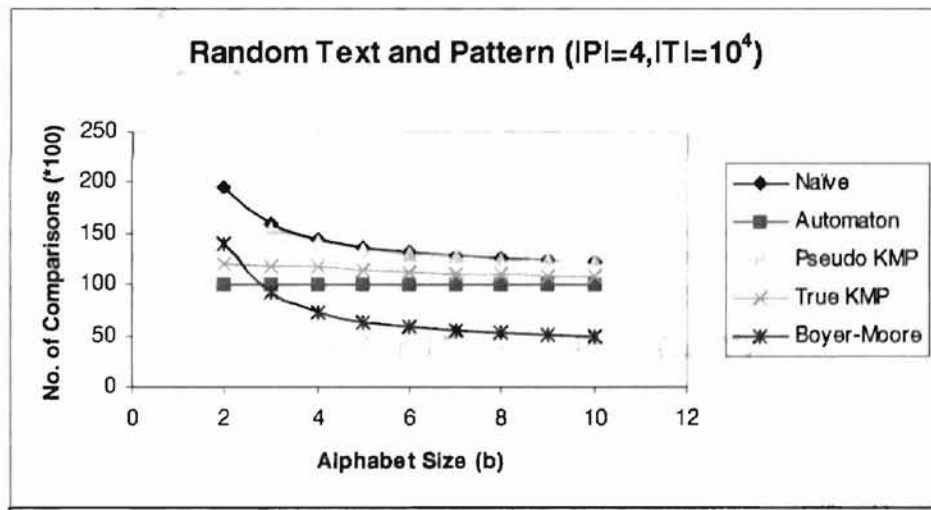
Figure 5-1 Impact of alphabet size on random string file with $|P| = 2$
 (a) Large alphabet set (b) Small alphabet set

When the pattern length is equal to 4, the impact of alphabet size on the random string file has a similar result as that of pattern length = 2, as shown in Figure 5-2. But from this figure, we can see the impact of pattern length on the performance on these algorithms. As the pattern length increases, the number of comparisons of the naïve

method and pseudo-KMP method increase. On the other hand, the number of comparisons of the Boyer-Moore algorithm decreases as the pattern length gets larger.



Note: Naïve and Pseudo KMP are overlapping; True KMP and Automaton are close.



Note: Naïve and Pseudo KMP are close.

Figure 5-2 Impact of alphabet size on random string file with $|P| = 4$
 (a) Large alphabet set (b) Small alphabet set

When the pattern length increases to 8, no significant change happens. The trend is still that as the pattern length gets larger, the number of comparisons of the naïve method and the pseudo-KMP method increases linearly. But the pattern length does not have too much impact on the true KMP algorithm. One thing that should be pointed out is that as the pattern length gets larger, the pseudo-KMP algorithm has a slightly better performance than the naïve method. The following is the corresponding plot, Figure 5-3.

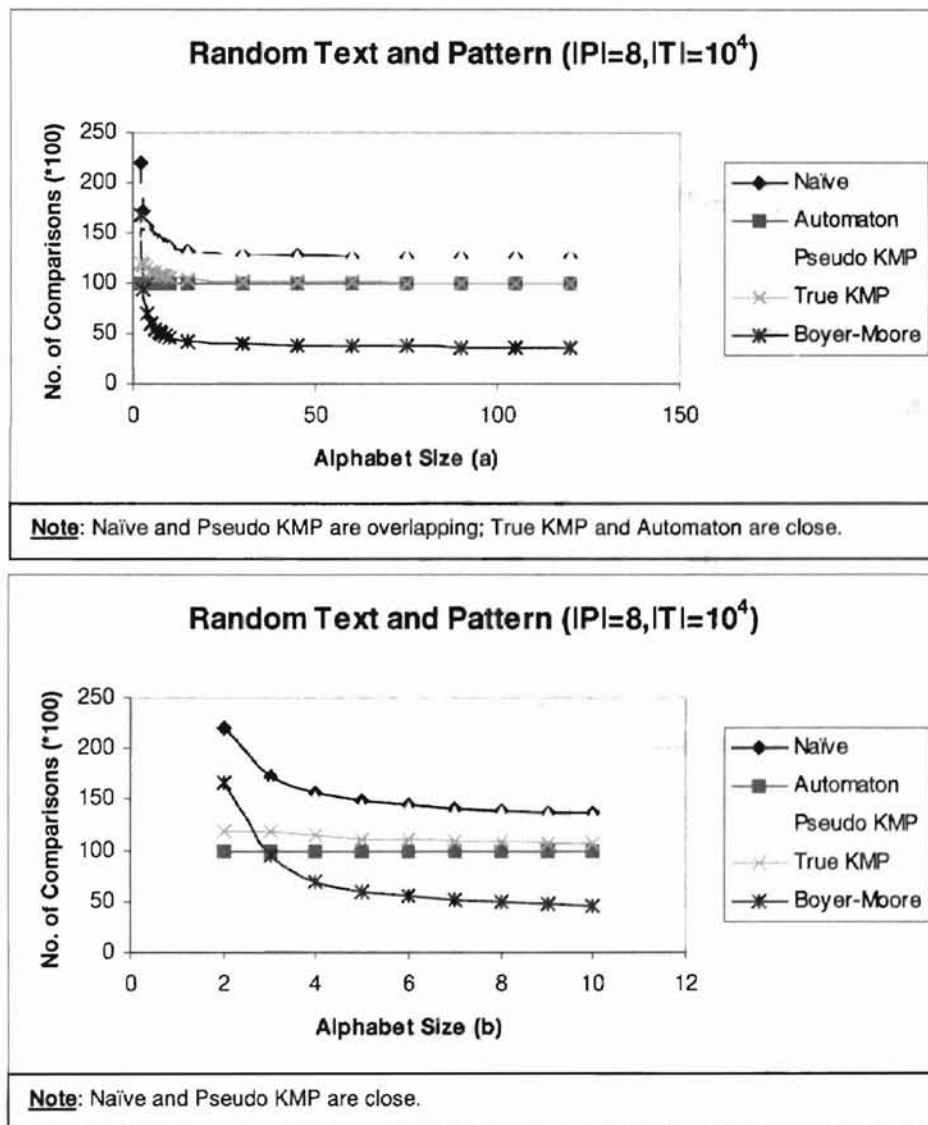


Figure 5-3 Impact of alphabet size on random string file with $|P| = 8$
 (a) Large alphabet set (b) Small alphabet set

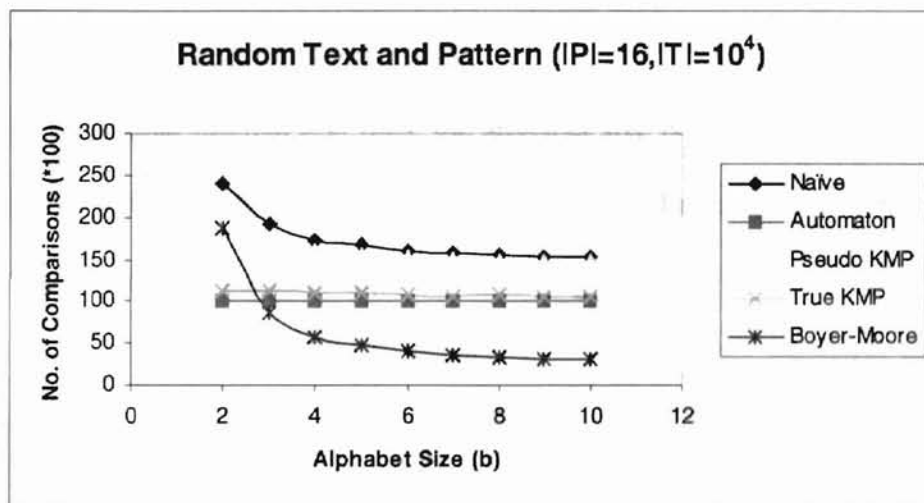
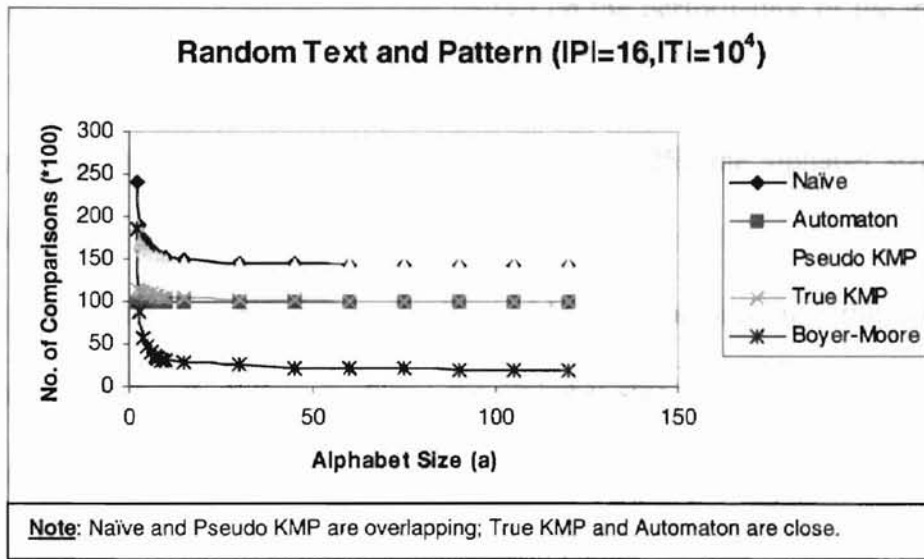


Figure 5-4 Impact of alphabet size on random string file with $|P| = 16$
 (a) Large alphabet set (b) Small alphabet set

When the pattern length reaches 16, the performances of these algorithms are shown as Figure 5-4. From this one and the above figures, we can conclude:

- When searching within the random text file, the number of comparisons of the naïve method and the pseudo-KMP method increases as the pattern length becomes

larger. In contrast, the performance of the Boyer-Moore approach improves with longer patterns. The pattern length has no obvious impact on the performance of the true KMP algorithm.

- When the alphabet reaches a reasonable size (e.g. 25), the alphabet size has no direct impact on the performance of these algorithms.
- To search within a random text, the Boyer-Moore is the best algorithm to be used when the alphabet size is large (>3). When the alphabet size is small (≤ 3), however, the true KMP algorithm is the best choice.

5.2 ENGLISH TEXT AND PATTERN

For most normal English text files, the alphabet size is fixed because they use the ASCII character set that contains 128 characters. These characters are non-uniformly distributed because some characters are used more frequently.

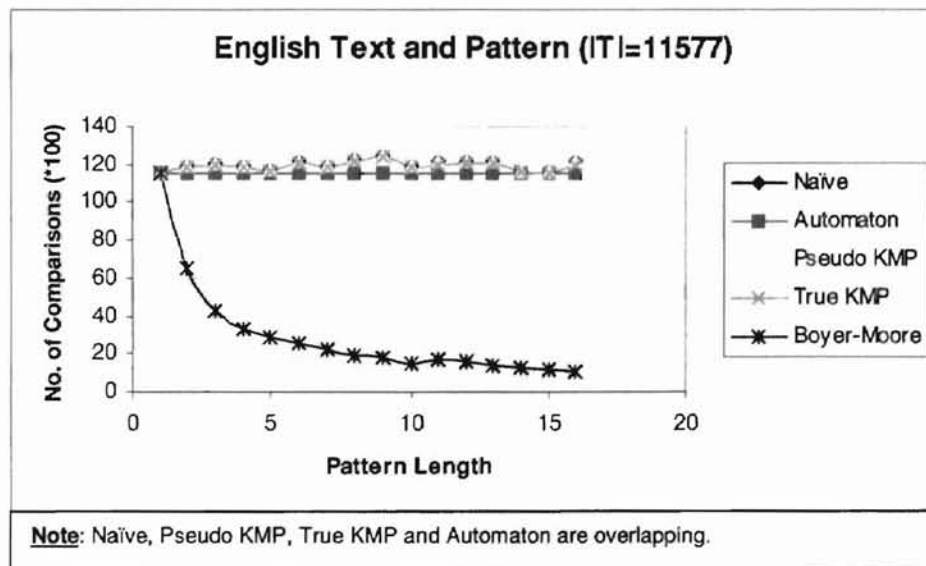


Figure 5-5 Impact of pattern length on English text file

Figure 5-5 shows the performance of these algorithms when searching within normal English text files. From this chart, we can see that the naïve method, the automaton method, the pseudo-KMP algorithm, and the true KMP algorithm have no obvious advantages over each other. Although the naïve method has the $O(nm)$ worst case running time, it is essentially linear when searching the English text file. This is because, in typical English text, a match of the first character of the pattern occurs infrequently, and the chances of the second character matching are even smaller, and so on. So the inner loop in the algorithm isn't likely to execute very often or for very long. Therefore, other methods do not have too much advantage over it when searching within an English text file.

The Boyer-Moore does very significantly better than others. Its performance is sub-linear, which requires about n/m comparisons on average from the observations on Figure 5-5. Therefore, the Boyer-Moore method should be the best algorithm to be used in actual applications.

5.3 BINARY STRING TEXT AND PATTERN

For binary string files, the size of the alphabet set is fixed, that is $\Sigma=\{0,1\}$. Figure 5-6 shows the performance of these algorithms when searching within a binary string file. Because of the small alphabet set, the true KMP algorithm gives the best performance, even much better than Boyer-Moore. The reason is that the text and the pattern are highly repeated with the small alphabet size, and the pattern may occur frequently in the text. Also because of this reason, the $O(nm)$ worst case running of the naïve method occurs.

Although the pseudo-KMP method degrades to the naïve method in previous cases, it greatly outperforms the naïve method when searching within a binary file.

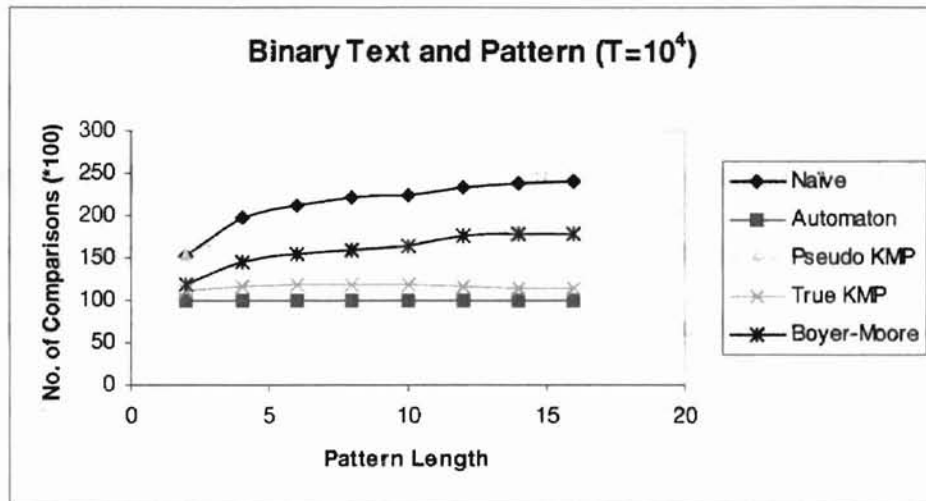


Figure 5-6 Impact of pattern length on binary string file

CHAPTER VI

SUMMARY, CONCLUSION AND SUGGESTED FUTURE WORK

6.1 SUMMARY

The naïve string matching algorithm is simple and straightforward. It has no preprocessing phase and needs no extra space. The comparisons between the text and pattern can be done in any order. The disadvantage of this method, however, is that it blindly tries each position of the text as the start of the pattern, even if it has already examined some of the characters in prior match attempts. Therefore, it requires backtracking in the text in the event of an unsuccessful match. Besides, its worst case running time is $O(nm)$, which occurs in highly repetitive text and patterns. However, such cases do not occur very often in practical applications, such as searching English text, as analyzed in the previous chapter.

Searching the pattern with an automaton requires first building the string matching automaton to recognize the language Σ^*P . The time complexity needed to build this automaton is $O(m^3|\Sigma|)$ and the space complexity needed is $O(m|\Sigma|)$. So if the alphabet size is large and the pattern is long, this method is very inefficient. Although the searching phase can be performed in $\Theta(n)$ time, this method is rarely used in practical applications. But studying this method can appreciably increase the understanding of other more sophisticated string matching algorithms.

The pseudo-KMP method uses the π function that stores the information about how the pattern matches against shifts of itself to improve the length of shifts. The time complexity and the space complexity for computing this function are both $O(m)$, which means it can be computed independently from the alphabet size. Although the pseudo-KMP algorithm runs in time $O(m + n)$, it is almost as inefficient as the naïve method. The reason is that it makes the same comparison of a pattern character to a text character more than once during the searching phase. Therefore, this method should be avoided.

Compared to the pseudo-KMP algorithm, the true KMP algorithm is simple and efficient. It computes the next function in $O(m)$ time during the preprocessing phase, which takes also $O(m)$ extra space. Although the algorithm runs in time $O(m + n)$, it avoids many unnecessary comparisons between the text and pattern. One outstanding advantage of this algorithm is that it never backs up in the text, thus making it a good example of an on-line algorithm. It works fast when the text and pattern is highly repetitive, such as in a binary string file.

The Boyer-Moore algorithm is the most efficient among these algorithms for most applications. Two functions, λ and γ , can be pre-computed in time $O(m + |\Sigma|)$ before the search phase and require an extra space in $O(m + |\Sigma|)$. Although it has the $O(nm)$ worst case time complexity, for large alphabets, the expected performance is sub-linear, requiring about n/m symbol comparisons on average. But with small alphabet size, it performs worse than the true KMP algorithm. If it is used as the on-line algorithm, it requires the text to be buffered, with a size equal to the length of the pattern.

6.2 CONCLUSION

Based on the above studies, the overall conclusions are as follows:

- If the pattern is small (i.e. $1 \leq m \leq 3$) and the alphabet is reasonably sizable, then the overhead of the preprocessing by the more sophisticated algorithms makes them less efficient than the naïve method. In this case, the naïve method may be preferable.

- If the alphabet size is small (i.e. $1 \leq |\Sigma| \leq 3$), the true KMP algorithm may perform significantly better than the Boyer-Moore algorithm.

- If the pattern is not too small and the alphabet size is reasonably large, the Boyer-Moore is obviously the best available algorithm among those studied in this paper.

Finally, the animated comparison of string-matching algorithms visualizes the actual performance of these algorithms, which provides a vivid and convenient studying tool to experience the comparisons.

6.3 SUGGESTED FUTRUE WORK

The results of comparison of these string matching algorithms may vary from different data sets. In the study, only the impact of pattern length and alphabet size is taken into consideration. Further study may be extended to the impact of the text length and the repetition factor of the pattern.

Finally, the animation system can be connected to a database system so that historical comparison results may be stored into the database. Thus, the animation system can generate the reports and the diagrams based on the comparison statistics, so that the end-user can be more knowledgeable about the average performances of these algorithms.

SELECTED BIBLIOGRAPHY

- [1] Aho A.V., Hopcroft J.E., Ullman J.D., *The design and analysis of computer algorithms*, Addison-Wesley, MA, 1974.
- [2] Auzenne, V. R., *The Visualization Quest: A History of Computer Animation*, New Jersey: Associated University Press. 1994.
- [3] Boyer R.S., Moore J.S., "A fast string searching algorithm", *Communications of the ACM*, Vol. 20, No. 10, pp.762-772. 1977.
- [4] Charras, C., and Lecroq T., *Exact String Matching Algorithms*, Universite de Rouen, France. 1995.
- [5] Cook S.A., "Linear time simulation of deterministic two-way pushdown automata", *Information Processing*, Vol. 71, pp. 75-80. 1972.
- [6] Cormen, T.H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, MIT Press. 1990.
- [7] Flanagan, D., *Java in A Nutshell*, Sebastopol, CA: O' Reilly. 1997.
- [8] Horspool R.N., "Practical fast searching in strings", *Software – Practice and Experience*, Vol. 10, No. 6, pp.501-506. 1980.
- [9] Horstmann, C.S., and Cornell, G., *Core Java 1.1 Volume1 - Fundamentals*, Sun Microsystems Press. 1997.
- [10] Karp R.M., Rabin M.O., "Efficient randomized pattern-matching algorithms", *IBM Journal of Research and Development*, Vol. 31, No. 2, pp.249-260. 1987.
- [11] Knuth D.E., Morris J.H., Pratt V.R., "Fast pattern matching in strings", *SIAM Journal on Computing*, Vol. 6, No.2, pp. 323-350, 1977.

- [12] Lewis, J., and Loftus, W., *Java Software Solutions - Foundations of Program Design*, Addison Wesley Longman. 1998.
- [13] Morrison, M., *Becoming a Computer Animator*. Indianapolis: Sams Publishing. 1994.
- [14] Muthukumarasamy, J., and Stasko, J.T., *Visualizing Program Executions on Large Data Sets Using Semantic Zooming*. <ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/95-02.ps.Z>. 1995.
- [15] Myers, B.A., "Taxonomies of Visual Programming and Program Visualization", *Journal of Visual Languages and Computing*, (1), pp. 97-123. 1990.
- [16] Price, B.A., Baeker, R.M., and Small, I.S., "A Principled Taxonomy of Software Visualization", *Journal of Visual Languages and Computing*, No.4, pp. 211-266, 1993.
- [17] Rivest R.L., "On the worst-case behaviour of string searching algorithms", *SIAM Journal on Computing*, Vol. 6, No. 4, pp. 669-674, 1977.
- [18] Roman, G., and Cox, K.C., *Program Visualization: The Art of Mapping Programs to Pictures*, Department of Computer Science, Washington University, 1992.
- [19] Sunday D.M., "A very fast substring search algorithm", *Communications of the ACM*, Vol. 33, No. 8, pp. 132-142. 1990.
- [20] Takaoka, T., "A Left-to-right Preprocessing Computation for the Boyer-Moore String-matching Algorithm", *Computer Journal*, V0039 N5, pp.413-416. 1996.
- [21] Thalmann, N., and Thalmann, D., *Interactive Computer Animation*. New Jersey: Prentice Hall. 1996.

- [22] Silberschatz, A., and Galvin, P.B., *Operating System Concepts*, Addison Wesley Longman. 1997.
- [23] Stasko, J.T., "Using Student-Built Algorithm Animations as Learning Aids", *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGSCE '97)*, San Jose, CA, 1997.
- [24] Stephen, G.A., *String Searching Algorithms*, World Scientific Publishing Company, 1995.
- [25] Vanhelsuwe, L., *Mastering Java 1.1*, Alameda, CA: SYBEX. 1997.

VITA

JIANYU ZHONG

Candidate for the Degree of

Master of Science

Thesis: ANIMATED COMPARISON OF STRING MATCHING ALGORITHMS

Major Filed: Computer Science

Biographical:

Personal Data: Born in Foshan, Guangdong, P.R. China, January 20, 1973, the daughter of Mr. Zhilian Zhong and Mrs. Sujuan Zheng.

Education: Graduated from Foshan No. 1 Middle School, Foshan, Guangdong, P.R. China, in July 1990; received Bachelor of Arts degree in Tourism and Hotel Management from Zhongshan University, Guangzhou, Guangdong, P.R. China in July 1994; Completed requirements for Master of Science degree at Oklahoma State University in December 1999.

Professional Experience: Research Assistant, Department of Hospitality Administration, Oklahoma State University, January 1997 to May 1997. Teaching Assistant, Department of Computer Science, Oklahoma State University, August 1998 to May 1999.