PATTERN-MATCHING ALGORITHMS ON INDETERMINATE STRINGS

PATTERN-MATCHING ALGORITHMS ON INDETERMINATE STRINGS

BY

SHU WANG, M.Sc. B.Eng.

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
MCMASTER UNIVERSITY
HAMILTON, ONTARIO, CANADA
JAN 2006

MCMASTER UNIVERSITY
DEPARTMENT OF
COMPUTING AND SOFTWARE

The undersigned hereby certify that they have read and recommend to the Faculty of Engineering for acceptance a thesis entitled "**PATTERN-MATCHING ALGORITHMS ON INDETERMINATE STRINGS**" by **Shu Wang** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: Jan 2006

Supervisor: _____
Dr. William F. Smyth

Readers: _____
Dr. Emil Sekerinski

_____
Dr. Jiming Peng

MCMASTER UNIVERSITY

Date: **Jan 2006**

Author: **Shu Wang**

Title: **PATTERN-MATCHING ALGORITHMS ON INDETERMINATE STRINGS**

Department: **Computing and Software**

Degree: **M.Sc.** Convocation: **May** Year: **2006**

Signature of Author

Table of Contents

List of Tables

List of Figures

Abstract

In computing, strings are sequences of simple elements (letters) drawn from some alphabet. A string is one of the most basic and important data structures in computer science, it exists everywhere in computer systems. Disk files, contents of memory, source and object code of computer programs, e-mail messages are all examples of strings. Also in bioinformatics, nucleotides in DNA can be viewed as strings drawn from an alphabet of four basic symbols – A, C, G and T. Algorithms that deal with strings are correspondingly very important in the field of computer science as well as in bioinformatics, data compression, cryptanalysis and other scientific areas.

A pattern-matching algorithm is one of the most fundamental string algorithms. Simply speaking, it outputs the occurrences of a string (called the pattern) within another string (called the text). Exact pattern-matching algorithms have been extensively studied in the last three decades and many algorithms have been proposed. Among them are two well known algorithms, the Knuth-Morris-Pratt (KMP) algorithm [KMP77], with good worst-case performance, and the Boyer-Moore (BM) algorithm [BM77], with good average-case performance. A recent attempt to combine the virtues of the two algorithms results in the Franek-Jennings-Smyth (FJS) algorithm [Jen02, FJS05b, FJS05a], which has both excellent average-case and worst-case performance.

Indeterminate strings are a new class of strings proposed in [HS03] driven by the increasingly common application of string algorithms on biological (DNA) sequences. Unlike letters in a normal alphabet, an indeterminate letter matches a set of letters (under certain constraints) rather than just one during pattern-matching. Perhaps

the most straightforward example of an indeterminate letter is the don't care letter '$*$', that matches all letters in the alphabet. The main purpose of an indeterminate string is to increase the flexibility of pattern-matching. It can also be seen as a model of a DNA sequence with a polymorphism property, which is a commonplace in molecular biology.

An indeterminate string is a relatively new idea and not too many known algorithms work on it, except for a particular version of the ShiftOr algorithm [WM92]. However as we will see, even this version can only handle a special case of indeterminate pattern-matching. Faster and more general indeterminate pattern-matching algorithms are desired. In this thesis we first give rigorous definitions of indeterminate strings and then develop our new indeterminate pattern-matching algorithms, adapted and modified from existing exact pattern-matching algorithms. There are many different constraints and also different models of patten-matching. We present our solution with all these variations in order of increasing complexity. We present our rationale of development and investigate the possibility of developing new algorithms from different categories of existing determinate algorithms. After describing our algorithms we conduct comprehensive experiments and compare the results carefully. In the last chapter we present our conclusions and point out the directions of future research.

Acknowledgements

First of all, I would like to thank Dr. William F. Smyth, my supervisor, for his enormous support for my work. This thesis would not be what it is without his constant encouragement and foresighted guidance. He shared so many great ideas with me and carefully corrected my mistakes and typos. I learned so much, not only from his academic knowledge but also from his personality. Again, thanks for the help – financially, intellectually and emotionally.

I should also thank Dr. Emil Sekerinski and Dr. Jiming Peng, who review this thesis and gave me valuable feedbacks and suggestions, and Dr. Jan Holub who contributed significantly in the development of indeterminate pattern-matching algorithms. Thanks to Dr. Franek and Christopher G. Jennings , who contributed indirectly in this thesis.

Of course, I am grateful to my parents and my whole family, who always love me and fully support my decision to pursuit graduate studies. Thanks for their patience all these years.

Thanks to (alphabetically) Fang Cao, Gang Chen, Huarong Chen, Lei Hu, Liuxing Kan, Munira Yusufu, Ning Liu, Pablo, Qian Yang, Vera, Wei Xu, Wen Yu, Xiang Ling, Yu Sun, Yun Zhai, and many others in the Computing and Software department, for their friendship.

Last but not least, special thanks to Tracy, who always brings me great happiness although sometimes distractions, for never destroying my computer as she always threatened to.

Hamilton, Ontario, Canada — Shu Wang
December, 2005

Chapter 1

Introduction

In this thesis we develop a collection of new algorithms that perform pattern-matching on indeterminate strings. First in this introductory chapter we give a brief overview of strings, string-searching algorithms and indeterminate strings. Then in later chapters we detail our development and experimental results.

1.1 Strings

In computer programming and some branches of mathematics, ***strings*** are sequences of various simple elements. These elements are selected from a predetermined set, called an ***alphabet***. Each element in this alphabet set is called a ***letter*** or ***character***. One of the most straightforward examples of an alphabet is the English alphabet which consists of 26 (if we don't distinguish upper and lower case) or 52 (otherwise) letters. Every English word is a actually a ***concatenation*** of these letters so it can be seen as a string drawn from the English alphabet. A normal English text is usually a combination of English words, punctuation (such as "; , ." and space symbol) and

various symbols (such as "@ $ %"). So it is a string drawn from an alphabet that contains English letters, punctuation and symbols.

An electronic computer actually uses only two symbols to store and process information; they are symbol '1', which can be represented by an electrical high voltage, and symbol '0', representable by a low voltage. So in a computer system essentially all the files, memory contents, I/O signals are strings drawn from a simple binary alphabet {0, 1}. In order to properly represent real world languages using a computer, various encoding systems are designed. Among them probably the most well-known one is the ASCII (American Standard Code for Information Interchange), which uses 128 7-digit binary numbers to represent 94 most commonly used printable characters (code 33 to 126) in the English text and 34 other control symbols. Some extensions of ASCII use 8 digits (one byte) to represent 256 extended characters. Although ASCII and its extensions are sufficient for representing the English language, some other languages such as Chinese, Japanese and Korean (known collectively as CJK) which are logographic languages, require far more than 256 characters, and thus exceed the limit of what one byte of digits can represent. As a solution letters of these languages are usually encoded into 2-byte digit codes, resulting in encoding systems different from ASCII. Actually even for the same language different encoding systems exist, sometimes causing conflicts with each other. Unicode [UNI] is an attempt to unify all the encoding systems for different platforms, programs and languages. It defines more

than 10,000 symbols and is getting more and more popular as an industry standard.

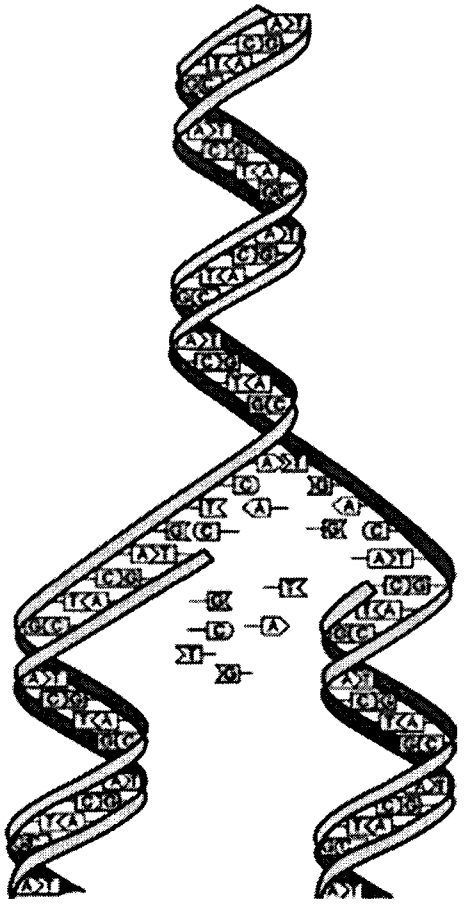

Figure 1.1: Illustration of DNA sequence. Image from [HGP].

One other important and well-known example of a string is the nucleotide sequences that make up DNA (deoxyribose nucleic acid). DNA is often referred to as the molecule of heredity, as it is responsible for the genetic propagation of most inherited traits. During reproduction, DNA is replicated and transmitted to the offspring. Figure 1.1 illustrates the DNA sequence as a string drawn from an alphabet of only

four letters. These letters are A, C, G and T, standing for adenine, cytosine, guanine, and thymine respectively. The Human Genome Project [HGP] was launched in 1990 as a cooperation by several countries to determine the sequences of the 3 billion chemical base pairs that make up human DNA. Finishing in 2003, now most of the human gene sequences are available at many public databases in the form of ASCII text files.

1.2 String Searching Algorithms

Strings are omnipresent in computers and many other systems; moreover, the information stored in strings is growing at an extremely fast pace. For example, the DNA sequence information retrieved from various species and stored in GeneBank [GB] doubles about every 10 months. The content available on the internet is growing at a similar speed. It is not surprising that algorithms which perform operations on strings are among the most fundamental and important algorithms.

Simply speaking, string-searching algorithms search for the leftmost or for all occurrences of a pattern string within a text string. A normal computer user may use this feature many times a day. He might search for a keyword inside the huge amount of internet data using a search engine; he might search for a particular phrase in an article using functions provided by a text editor; and he might use virus detection software to search for viruses in order to protect the system. The recent competition between Google and Microsoft over desktop search tools which allow users to search

keywords from local e-mail files, instant messages, database files and potentially any files reflects the fact that efficient string-searching algorithms are crucial to much software.

Over the last three decades, dozens of algorithms have been proposed to solve the pattern-matching problem. These algorithms generally consist of distinct functions, as we discuss below:

1.2.1 Preprocessing

Pattern-matching algorithms can be categorized by how they preprocess the text and pattern strings. Because usually we assume the pattern string is relatively small compared with the text string, we only classify pattern-matching algorithms by whether the text string is preprocessed or not.

Many pattern-matching applications require the text string to be preprocessed. The web search engines, databases, and the recent desktop (local) search tools fall into this category. For example, Google uses many WebCrawlers to index the entire content of the internet, then after this kind of preprocessing, a user can submit keyword (pattern) strings to the server and get results back in less than one second. The relatively new desktop search tools, unlike traditional file searching tools, index (preprocess) the entire content of the local hard disk when first installed, and update their indexes regularly.

If the task is to search for pattern(s) inside a single text string and preprocessing

is allowed, then the text string can be transformed into a data structure which is more convenient for searching and the searching can be completed in linear time. Many algorithms can perform these kinds of transformation such as suffix tree and suffix array construction algorithms. However, this kind of string searching with preprocessing is not within the scope of this thesis; we'll focus on pattern-matching described as follows.

In some cases the text is not available for preprocessing before the searching. For example, the US Federal Bureau of Investigation was using an internet content filter called Carnivore [Jen02] to collect potential evidence messages. The filter was placed on the ISP gateway, which every internet user packet must pass through. Those packets that matched against certain patterns were to be intercepted and others not. Preprocessing is obviously not possible in this case, moreover the filter must work in real-time to cope with the huge amount of data that travels through the ISP server each day. Also in text editors and virus scanning softwares, where preprocessing is not desirable or not permitted, it is better or sometimes mandatory to use pattern-matching algorithms without preprocessing. All traditional pattern-matching algorithms fall into this category such as the famous KMP and BM algorithms and the more recent FJS algorithm. In this thesis we focus on pattern-matching algorithms that do not preprocess the text string.

1.2.2 Approximation

Pattern-matching algorithms can also be categorized by whether one is doing exact matching or not. Simply speaking, exact pattern-matching outputs the positions of the occurrence(s) of the pattern in the text. It is the most common form of pattern matching and the foundation of other forms of searching. Most early pattern-matching algorithms were exact string searching algorithms.

Driven by the more and more common applications of pattern-matching algorithms in bioinformatics, musical analysis and time series matching, inexact or approximate pattern-matching algorithms are more and more widely used. They output not only the exact occurrences but also the occurrences with some degree of error. For example, when searching for a particular melody from a music database, some minor change in the tone may be considered insignificant and can be tolerated by the searching algorithm.

1.2.3 Indeterminism of the Letters in the Alphabet

String searching algorithms can be categorized by the indeterminism of the alphabet. The concept of an indeterminate string was first proposed in [HS03] driven by the requirement from computational molecular biology. Simply speaking, an indeterminate string contains indeterminate letters that match a set of letters rather than just one letter as in the usual case. The readers are probably already familiar with the don't care symbol '$*$', which is a special case of indeterminate letters. It matches

all letters in the alphabet during pattern-matching. A general indeterminate letter matches some or all letters in the alphabet, subject to the user definition.

The main purpose of indeterminate strings is to increase the flexibility of pattern-matching. For example, if the user requires that during pattern-matching all the lower-case English letters match not only the same letters in the alphabet, but also their upper-case equivalents, then all the lower-case English letters in this case are defined as indeterminate letters. Also in bioinformatics, it is usually the case that some nucleotide acid bases in DNA sequences are unknown or undetermined, in other words, polymorphism may occur in some positions. For example, in the fasta file format [FAS] that is used as a standard to store DNA sequence information, besides the four basic alphabet A, C, G and T, there are several other letters such as 'R' which stands for 'G' or 'A'; and letter 'B' stands for 'G', 'T' or 'C'. These can all be seen as examples of indeterminate letters.

Indeterminate pattern-matching is similar to approximate pattern-matching in the sense that it also outputs both exact and not exact occurrences of the pattern string. However, it is not entirely the same as approximate pattern-matching because indeterminate letters can also appear in the text string, which make it especially suitable for situations such as the previous DNA example. Indeterminate stings can be represented by regular expressions. For instance an indeterminate letter 'R' that stands for 'G' or 'A' can be represented by a regular expression such as "[GA]".

However, it is more desirable to define a single letter to replace such a set of letters.

1.3 The New Algorithms

The idea of indeterminate strings is relatively new and not many algorithms are available that work on indeterminate strings. The only known existing algorithm that can partially perform indeterminate pattern-matching is a particular version of the ShiftOr algorithm, proposed in [WM92]. However, ShiftOr was not intentionally designed to solve the indeterminate pattern-matching problem so it allows the indeterminate letters to appear in the pattern only. In this thesis we develop new and more general indeterminate pattern-matching algorithms. We try to develop our new algorithms from three different categories of determinate algorithms. That is, we investigate the possibility of developing new indeterminate pattern-matching algorithms from BMS, ShiftOr and FJS respectively.

There are many different versions of the new algorithm due to different constraints that may be posed on the matching and the resulting different definitions of indeterminism. We present all these variations in order of increasing complexity.

After describing our algorithms we show the experimental results and compare them with each other. We conclude that the new algorithms based on BMS in most cases are faster than ShiftOr and moreover provide a more flexible way to conduct pattern-matching.

Chapter 2

Definitions and Notation

In this chapter we give more rigorous definitions of the terminology that we use throughout this thesis.

2.1 Strings and Alphabet

In order to properly define a string we have to first define the ***alphabet***. Simply speaking, an alphabet is a finite set of distinct symbols, and it is usually denoted by a Greek symbol Σ. We call the symbols of an alphabet ***letters***. An alphabet and its letters can be represented as $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_k\}$, where $k = |\Sigma|$ stands for the ***size of the alphabet***.

There are three types of alphabet with increasingly strict conditions. A ***general alphabet*** is an alphabet that given $\sigma_1, \sigma_2 \in \Sigma$, it is possible to decide if $\sigma_1 = \sigma_2$ in constant time. An ***ordered alphabet*** is an alphabet that given $\sigma_1, \sigma_2 \in \Sigma$ and some order relation '$<$', it is possible to decide if $\sigma_1 < \sigma_2$ or $\sigma_2 < \sigma_1$ or $\sigma_1 = \sigma_2$ in constant time. An ordered alphabet Σ is also an ***indexed alphabet*** if every

$\sigma \in \Sigma$ can be used as an index in a finite array: thus if $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ with $\sigma_1 < \sigma_2 < \cdots < \sigma_k$, an array $A = A[\sigma_1..\sigma_k]$ can be defined so that $A[\sigma], \sigma \in \Sigma$, can be accessed in constant time. So for simplicity, without loss of generality, we therefore suppose that an indexed alphabet Σ is the set of integers $\{1, 2, ..., k\}$. This kind of alphabet is also called an ***integer alphabet***.

Simply speaking, a ***string*** is a sequence of symbols ***drawn*** from an alphabet. In other words, a string is composed of symbols that are all members of an alphabet Σ. We can define string more formally using the definitions as described in [Smy03].

The ***length of a string*** x is defined as the total number of symbols that it contains, and is denoted by $|x|$. If a string contains no symbols, then it is an ***empty string***, denoted by symbol ε.

2.2 Basic Operations and Properties of Strings

A string can be stored in a computer system in different forms. It can be stored in the form of an array, a linked list or a tree. An array is one of the simplest and fundamental data structures in many computer systems and it provides a natural way to represent a string. We can store all symbols of a string in sequence into an array and refer to each symbol using an index in constant time. Arrays permit efficient ***random access*** but are not efficient in operations such as insertion and deletion. Linked lists have the opposite trade-off. Consequently, arrays are most appropriate for storing a fixed amount of data which will be accessed in an unpredictable fashion,

and linked lists are best for a list of data which will be accessed sequentially and updated often with insertions or deletions. In this thesis we adopt the convention that all strings are stored in arrays, and we can refer to particular sequence positions in a string in constant time. We also borrow from the array notation common to many programming languages and use $x[i]$ to represent the ith symbol in the string x, where i is an integer such that $1 \leq i \leq |x|$.

Concatenation is an operation that, given strings y and z, produces a new string x by appending z to y. We denote the concatenation of y and z as $y \cdot z$ or just yz. A ***substring*** of a string w is any string y such that $w = xyz$. Here any of x, y and z may be the empty string ε. If $x = \varepsilon$, then y is also a ***prefix*** of w. If $z = \varepsilon$, then y is also a ***suffix*** of w. A string w is a substring, prefix and suffix of itself. If y is a substring, prefix or suffix of w and $|y| < |w|$, then we call y a ***proper*** substring, prefix or suffix as appropriate. Note that according to this definition ε is both a proper prefix and a proper suffix of x. For convenience we use notation $x[i..j]$ to represent the substring of x that starts from position i and ends at position j inclusive. If $i = j$, then $x[i..j] = x[i]$, and if $i > j$, we define $x[i..j] = \varepsilon$.

If b is both a proper prefix and suffix of x, then we say b is a ***border*** of x. In many applications we are particularly interested in the ***longest border*** of a string, denoted by b^*.

In many cases we are also interested in a string that has repeated substrings.

We use the exponent notation to indicate the number of copies. Thus $a^2 = aa$, $(ab)^3 = ababab$.

String algorithms almost always involve letter comparisons. When performing a letter comparison we usually say that two letters λ and μ ***match*** iff $\lambda = \mu$. The definition of ***matching of letters*** can easily be extended to ***matching of strings*** as follows: given two strings $x[1..n]$ and $y[1..n]$, x matches y iff $\forall i \in 1..n, x[i] = y[i]$.

A "***don't care***" letter, usually denoted by the '$*$' symbol, has the property of matching any letter in the alphabet Σ. For example, a string $x = a * b$ matches with string $y = *a*$. Taking the "don't care" letter into consideration, the definition of matching of letters described above should be extended in the following way: Given an alphabet Σ, let $\Sigma' = \Sigma \cup \{*\}$. Two letters $\lambda, \mu \in \Sigma'$ ***match*** each other (denoted by $\lambda \approx \mu$) iff

$$(\lambda = \mu) \vee (\lambda = *) \vee (\mu = *)$$

Note that the relation '$\approx$' is not transitive. For example, $\lambda \approx * \approx \mu$ doesn't imply $\lambda \approx \mu$.

2.3 Indeterminate Strings

Indeterminate letters can be seen as extensions of "don't care" letters while each letter matches not all but a subset of letters in the alphabet. Given a normal alphabet Σ, a letter λ is said to be an indeterminate letter if it matches more than one letter in the

alphabet Σ. For example, suppose $\Sigma = \{1, 2, 3\}$, then as described previously, the "don't care" letter '$*$' matches any letter from set {1,2,3}, but we could in addition define indeterminate letters '4', '5', '6' and '7' so that $\Sigma' = \{1, 2, 3, 4, 5, 6, 7\}$ and during letter comparison: '4' matches any letter from set {1,2}, '5' from {2,3}, '6' from {1,3} and finally '7' from {1,2,3}.

We give the formal definition of indeterminate letters as follows: Let $\Sigma = \{1, 2, ..., k\}$ be an integer alphabet as described in the previous section; we define an ***extended alphabet***

$$\Sigma' = \{1, 2, ..., k, k+1, ...K\},$$

where for every $j \in k+1..K$, we associate a unique nonempty subset of Σ with j. We call this subset Σ_j; so that in letter comparison j matches any letter from Σ_j; we also require that $|\Sigma_j| \geq 2$. There are 2^n subsets of the set Σ, among them, k subsets have only one element, corresponding to determinate letters $1..k$, and one is the empty set. So $K - k \in 0..2^k - k - 1$, which means for an alphabet Σ we can have at most $2^k - k - 1$ extra indeterminate letters. If $K - k = 0$, then $\Sigma = \Sigma'$, meaning that there's no extra indeterminate letter defined. If only the "don't care" letter is defined, then $K - k = 1$ and $\Sigma_K = \Sigma$. Thus a "don't care" letter can be seen as a special case of indeterminate letters.

Having the definition above, we are now ready to define ***matching of indeterminate letters***. We say two indeterminate letters $i, j \in \Sigma'$ ***match*** iff $\Sigma_i \cap \Sigma_j \neq \emptyset$.

Considering both the case of determinate letters and indeterminate letters, we redefine matching of two letters as follows: given two letters $i, j \in \Sigma'$

- If $(i \leq k) \wedge (j \leq k)$ (both i, j are determinate letters), then $i \approx j \iff i = j$.
- If $i \leq k < j$ (exactly one of i and j is an indeterminate letter), then $i \approx j \iff i \in \Sigma_j$, similarly for $j \leq k < i$.
- If $(i > k) \wedge (j > k)$ (both i, j are indeterminate letters), then $i \approx j \iff \Sigma_i \cap \Sigma_j \neq \emptyset$.

In practice, it is convenient to use a simplified definition as follows: we regard determinate letters as a special form of indeterminate letters. In other words, we define a Σ_j for every determinate letter ($j \in 1..k$) similar to the way we define Σ_j for indeterminate letters ($j \in k+1..K$), while the subset Σ_j we associate to $j \in 1..k$ is just j itself. In other words, if $j \in 1..k$, $\Sigma_j = \{j\}$. Now we can consolidate the definition of letter matching for determinate letters and indeterminate letters by defining

$$i \approx j \iff \Sigma_i \cap \Sigma_j \neq \emptyset$$

As mentioned earlier , the matching relation '$\approx$' is not transitive for an alphabet containing "don't care" letters. In general, it is not transitive for any alphabet containing indeterminate letters. For example, if $5 = \{1, 2\}$, then $1 \approx 5$ and $2 \approx 5$, but $1 \not\approx 2$.

The idea of "indeterminate" can be extended to strings. A string containing indeterminate letters is called an ***indeterminate string***. For comparison purposes, we call a string without any indeterminate letter a ***determinate string***. The definition of ***matching*** (also denoted by $\approx$) between two indeterminate strings is not as straightforward as the determinate one. It may or may not be confined by some ***constraints*** as described below:

No constraint If there are no additional constraints imposed on the matching between strings u and v, then $u[1..n] \approx v[1..n] \Leftrightarrow \forall i \in 1..n,\ u[i] \approx v[i]$. This implies that there may exist an indeterminate letter j that matches simultaneously more than one letter from Σ_j during the match. For example if '5' is an indeterminate letter, '1' and '2' are basic determinate letters, and $\Sigma_5 = \{1, 2\}$, then string "515" matches "112". We observe that letter '5' matches both '1' and '2' during one single substring comparison procedure. In some cases this kind of matching may be acceptable, while in other cases it may not. For example, in DNA sequences, indeterminate pattern-matching result with no constraint may not be preferred due to biological reasons.

Local constraint If we require that the assignment of all indeterminate letters has to be consistent during one single substring comparison procedure — namely, in one comparison we do not allow an indeterminate letter to be assigned to a determinate letter in one position and be assigned to a different determinate

letter somewhere else, as in the previous example — then we call it ***matching with local constraint***. Put more formally, we say during the matching of strings u and v, for all indeterminate letter $j \in \Sigma'$, let $k_1, k_2, ..., k_t \in \Sigma'$ be all the letters that align with letter j (j may appear at many positions), we require

$$\Sigma_j \cap \Sigma_{k_1} \cap \Sigma_{k_2} \cap \Sigma_{k_3}, ..., \cap \Sigma_{k_t} \neq \emptyset.$$

In other words, Σ_j should have a common subset with the alphabets of all letters that it aligns with during a single substring matching subroutine. For example, if $u = 515, v = 611$, '5' ,'6' are indeterminate letters, '1', '2' and '3' are determinate letters, $\Sigma_5 = \{1, 2\}$ and $\Sigma_6 = \{2, 3\}$ then for position 1, $\Sigma_5 \cap \Sigma_6 = \{2\}$, for position 3, $\Sigma_5 \cap \Sigma_1 = \{1\}$, and $\{1\} \cap \{2\} = \emptyset$. So by the definition above u does not match v with local constraint.

Global constraint Local constraint as defined above requires the assignment of indeterminate letters to be consistent during a single substring comparison. We may also have some cases that require the assignment of indeterminate strings to be consistent throughout the whole pattern-matching procedure. For example, in pattern matching, if $\Sigma_5 = \{1, 2\}$ and we want to find the occurrence of pattern "512" in text "112212212". Then with local constraint we can find occurrences in both position 1, 4 and 7, but with global constraint we will face two options, either we choose to assign indeterminate letter '5' to determinate letter '1' throughout the pattern-matching procedure and find occurrence only

in position 1 or assign it to '2' and find occurrences in position 4 and 7. How to make the choice can be passed to the user to decide, but in practice the particular assignment of all the indeterminate letters that achieve the maximum number of occurrences will probably be most desirable, and that is an interesting optimization problem which will be left for future research.

Now we are ready to give the definition of ***matching of indeterminate strings***. We say that string $u[1..n]$ matches string $v[1..n]$ with constraint C (denoted by $u \approx v$ with C) if and only if $\forall i \in 1..n, u[i] \approx v[i]$ and constraint C is satisfied.

Let x be an indeterminate string on $\Sigma' = \{1, ..., k, ..., K\}$, a string $\hat{x}$ where $1 \leq \hat{x}[i] \leq k$ for all $i \in 1..|x|$ (thus $\hat{x}$ is a determinate string) is an ***assignment*** of x with constraint C if and only if $\hat{x} \approx x$ while constraint C is satisfied.

Indeterminate strings have some properties that are different from traditional determinate strings. One of them relates to the border. In traditional determinate strings a border b of x is any proper prefix of x that equals a suffix of x. We could extend this definition to indeterminate strings by saying: a border b of an indeterminate string x is any proper prefix of x that matches ($\approx$) a suffix of x. One problem with this definition is that it may cause some indeterminate letters in certain positions to be assigned to more than one value simultaneously. Consider the following example: for $x = a**b$, the longest border defined by the above will be 3 since "$a** \approx **b$". However, we observe that $x[2] = *$ will be assigned to 'a' and 'b' simultaneously. This

kind of border was defined as a ***quantum border*** [HS03]. A quantum border of string $x = x[1..n]$ on Σ' is a proper prefix $x[1..b]$ such that $x[1..b] \approx x[n-b+1..n]$ with no constraint. Note that there may be no assignment of x that can satisfy a quantum border just as the previous example shows. On the other hand, a ***deterministic border*** of string $x = x[1..n]$ on Σ' is a proper prefix $x[1..b]$ such that $x[1..b] \approx x[n-b+1..n]$ with local constraint.

It is mentioned in [HS03] that another special case of an indeterminate letter is the ***empty letter*** ε. If we allow alphabet Σ to contains the empty letter ε, — then correspondingly Σ_j may or may not contain the empty letter. If λ_j is assigned to ε, then it equals a "delete" operation similar to that defined in ***string alignment***. For example, if $5 = \{\varepsilon, 1, 2\}$, then "153" matches both "13" and "113". Note that if two strings u, v contain the empty letter, u can match v even if $|u| \neq |v|$. However this case is beyond the scope of this thesis, we would like to leave it for future work.

Chapter 3

Fundamental Algorithms

In order to better understand the new algorithms that we are going to develop let us first discuss a bunch of fundamental algorithms in this chapter.

3.1 Border Array

As discussed in chapter 2, a border of a string x is a substring y of x such that y is both a proper prefix and a proper suffix of x, therefore $y = x[1..|y|] = x[|x| - |y| + 1..|x|]$. Also note that $\forall i \in 1..n$, $x[1..i]$ is a substring of x. Now we define a ***border array*** β of string x to be a string such that:

- $|\beta| = |x|$
- $\forall i \in 1..n$, $\beta[i]$ gives the length of the longest border of string $x[1..i]$.

Table 3.1 gives an example of the border array of a string x:

The border array has many properties, we state some of them below as lemmas, since they will be used in our later discussion.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	4	5

Table 3.1: Example of a border array

Lemma 3.1.1. $\beta[1] = 0$

Proof. Since the border length is strictly less than the string length, and since every string has an empty border, therefore $x[1..1]$ has a longest border of length 0.

□

Lemma 3.1.2. *If $x[1..i]$ is a border of $x[1..n]$, then string $x[1..i+1]$ is a border of string $x[1..n]x[i+1]$.*

Proof. This lemma is trivial according to the definition of the border.

□

Lemma 3.1.3. *If $x[1..i]$ is the longest border of $x[1..n]$, then string $x[1..i+1]$ is the longest border of string $x[1..n]x[i+1]$.*

Proof. We can use contradiction to prove the correctness of this lemma. If there is another border say $x[1..i']$ that is a border of $x[1..n] \cdot x[i+1]$ and it is longer than $x[1..i+1]$ (thus $i' > i+1$), then $x[1..i'-1]$ is a border of $x[1..n]$ and it is longer than $x[1..i]$ (because $i'-1 > i$), therefore contradicting the assumption that $x[1..i]$ is the longest border of $x[1..n]$.

□

Lemma 3.1.4. *Given a string $x[1..n]$ and $i \in 1..n-1$, if $x[1..i]$ has a longest border $x[1..b]$ and $x[b+1] = x[i+1]$, then $x[1..b+1]$ is the longest border of $x[1..i+1]$.*

Proof. This lemma can be directly deduced from the previous two lemmas.

□

Theorem 3.1.5. *If b is a border of x, and b' is a border of b, then b' is a border of x. (A border of a border of x is a border of x.)*

Proof. We prove this theorem as follows: if b is a border of x, then $b = x[1..|b|] = x[|x|-|b|+1..|x|]]$. b' is a border of b so $b' = b[1..|b'|] = x[1..|b'|] = b[|b|-|b'|+1..|b|]] = x[|x|-|b'|+1..|x|]]$. So b' is also a border of x.

□

Theorem 3.1.6. *If b and b' are both borders of a string x, and $|b| < |b'|$, then b is a border of b'.*

Proof. We prove this theorem as follows: $b' = x[1..|b'|] = x[|x| - |b'| + 1..|x|]$, $b = x[1..|b|] = b'[1..|b|] = x[|x| - |b| + 1..|x|]] = x[|b'| - |b| + 1] = b'[|b'| - |b| + 1]$, so b is a border of b'.

□

Theorem 3.1.7. *The jth-longest border of $x[i]$ is a border of the $(j-1)$th-longest border.*

Proof. This theorem follows directly from the previous theorem.

□

Given a string x, let $b^j[i], j = 1, 2, ..., k$ represent the jth-longest border of $x[1..i]$, and let $\beta^j[i]$ represents their corresponding length (so $\beta^1[i] = \beta[i]$ and $\beta^k[i] = 0$). We have an important theorem as follows:

Theorem 3.1.8. *The jth-longest border of $x[i]$ is the **longest** border of the $(j-1)$th-longest border. Or put it in symbols:*

$$\beta^j[i] = \beta[\beta^{j-1}[i]] \quad e.g.\ \beta^2[i] = \beta[\beta[i]]$$

Proof. We prove this by contradiction: From the previous theorem we know that b^j is a border of b^{j-1}, so if b^j is not the longest border of b^{j-1}, then there must exist another string b' such that b' is a border of b^{j-1} and $|b'| > \beta^j$. According to theorem 3.1.8 b' is also a border of $x[i]$ which has length less than b^{j-1} but greater than b^j. Thus b^j cannot be the j-th longest border of $x[i]$, hence a contradiction exists.

□

This theorem leads to an efficient algorithm to compute the border array of a given string x. But first let us look at a straightforward algorithm as follows.

3.1.1 The Brute Force Border Array Calculation Algorithm

The border array of a string $x[1..n]$ can be calculated in an obvious way. If $i = 1$ then $\beta[i] = 0$; for each $i \in 2..n$, we can calculate the longest border of $x[1..i]$ using the following strategy: we first compare string $x[1..i-1]$ with $x[2..i]$, if match, then $x[1..i-1]$ is the longest border. If not, we compare $x[1..i-2]$ with $x[3..i]$. We reduce the length of comparison each time and repeat this process until a prefix of x matches a suffix of x of the same length, otherwise no such a prefix is found so the length of longest border is 0. We store the length of the longest border of $x[1..i]$ in $\beta[i]$ for every i, and the array β is the border array we want.

The Brute Force border array calculation algorithm is obviously correct but not very efficient. The time required by this algorithm is $O(n^2)$.

3.1.2 Classical Border Array Calculation Algorithm

Border array can be calculated much more efficiently using a classical algorithm [AHU74] described in Figure 3.1. This algorithm is a consequence of Theorem 3.1.8: because the j-th longest border is the longest border of the $(j-1)$-th longest border of $x[i]$, so we can start from $\beta[x[i]]$, "travel down" in array β, thus all the borders of $x[i]$ can be obtained. The longest border of $\beta[x[i+1]]$ must be one of these borders concatenated with the following letter.

Algorithm described in Figure 3.1 correctly computes the border array β of string

Algorithm of border array calculation

Input: *String* $x[1..n]$
Output: *String* $\beta[1..n]$

```
Array : β[1..n]
Integer : b, i
β[1] ← 0
for i ← 1 to n − 1 do
    b ← β[i]
    while b > 0 and x[i + 1] ≠ x[b + 1] do
        b ← β[b]
    if x[i + 1] = x[b + 1] then
        β[i + 1] ← b + 1
    else
        β[i + 1] ← 0
return β
```

Figure 3.1: Algorithm of border array calculation

x and has a time complexity of $O(n)$. Interested readers can refer to page 14-17 of [Smy03] for proofs of its correctness and time complexity.

3.2 Determinate Pattern-Matching Algorithms

We now turn to several determinate pattern-matching algorithms.

3.2.1 The Brute Force Pattern-Matching Algorithm

First to get started we look at a simple pattern-matching algorithm called Brute Force algorithm. The basic strategy of Brute Force algorithm is to use the so-called "sliding window" method. That is, it assumes there is an imaginary moving window which has the same length as p. Initially the window starts from the left end of the

text string; in a substring-comparison subroutine we compare p with the substring of x in the current "window", and shift the window to the right according to the result of substring-comparison. How far to the right we shift varies from algorithm to algorithm, but all these algorithms usually have a similar substring-comparison subroutine. In the following parts of this thesis this subroutine will be used many times so we describe it more precisely as in Figure 3.2:

Function `StringComparison`

Input: *String* $u[1..n]$, $v[1..n]$
Output: *Integer* k which is the smallest integer such that $u[k] \neq v[k]$ or $k = m + 1$ in case two strings match each other completely.

```
1    Integer: i
2    i ← 1
3    while (u[i] = v[i] and i ≤ m) do
4        i ← i + 1
5    output i
```

Figure 3.2: Function `StringComparison`

Later in this thesis we will see that for indeterminate algorithms the function `StringComparison` may be a different computational procedure than the one in Figure 3.2. But for now let us get back to Brute Force algorithm: it effectively calls `StringComparison`$(x[i..i + m - 1], p)$ to test if these two strings match, outputs an occurrence in case of match, then shifts the current window to the right by just one position, and performs the string comparison again (see Figure 3.3 for illustration).

The Brute Force algorithm is described in Figure 3.4. It is obvious that the Brute

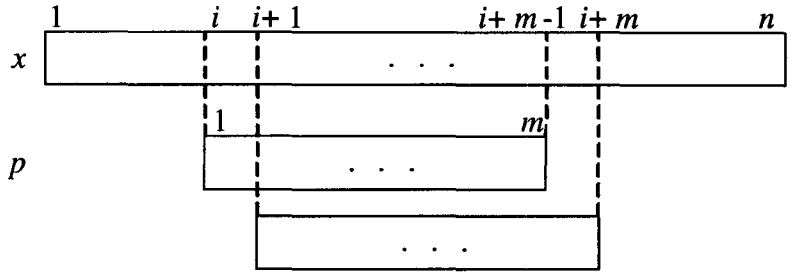

Figure 3.3: Brute Force shifts

Algorithm Brute Force

Input: *String* $x[1..n]$, $p[1..m]$
Output: Positions of occurrences of p in x

```
Integer : i
i ← 1
while i ≤ n − m + 1 do
   if StringComparison(x[i..i + m − 1], p) = m + 1 then
      output i
   i ← i + 1
```

Figure 3.4: Algorithm Brute Force pattern-matching

Force algorithm correctly outputs every occurrence of p in x and it has a worst case $\Theta(mn)$ running time, achieved by inputs such as $x = a^n, p = a^m$.

3.2.2 The Knuth-Morris-Pratt Algorithm

The KMP algorithm [KMP77] is probably the most well known pattern-matching algorithm in early years. It improves the Brute Force algorithm by skipping unnecessary letter comparisons, therefore increasing the length of the shift after every

`StringComparison` function. We give detailed description as follows:

Imagine we are performing pattern-matching using the same sliding window strategy as the Brute Force algorithm. That is, we compare p with the substring of x in the current window, if match we output an occurrence, if not we slide the window to the right. And imagine we are in a situation as demonstrated by Figure 3.5. The variables i and j indicate the current positions of letters being compared in strings x and p respectively. Imagine we also know the longest border of string $p[1..j-1]$, say string B. Then if $x[i] \neq p[j]$, it is secure to shift p to the right to a new position so that the prefix B of $p[1..j-1]$ overlaps with the suffix B of $p[1..j-1]$ in the old position. So now we compare $x[i]$ with $p[\beta[j-1]+1]$. Why is it secure to shift to this position knowing that there cannot be any occurrence between the new position and the old position? Because if there is one, for example the dashed position in Figure 3.5, then $B1$ is a border of $p[1..j-1]$ and it is longer than B. Therefore it contradicts the assumption we made in the beginning that B is the longest border of $p[1..j-1]$.

The shifting strategy described above was adopted first by the MP algorithm in [MP70]. In the MP algorithm an auxiliary array MP_Next is pre-computed using the following formula:

$$MP\_Next[j] = \begin{cases} 0, & \text{if } j = 1; \\ \beta[j-1]+1, & \text{if } j \in 2..n+1 \end{cases}$$

During pattern-matching, when the pattern is going to be shifted to the right, the algorithm looks for the number in $MP\_Next[j]$, and shifts to the right according to

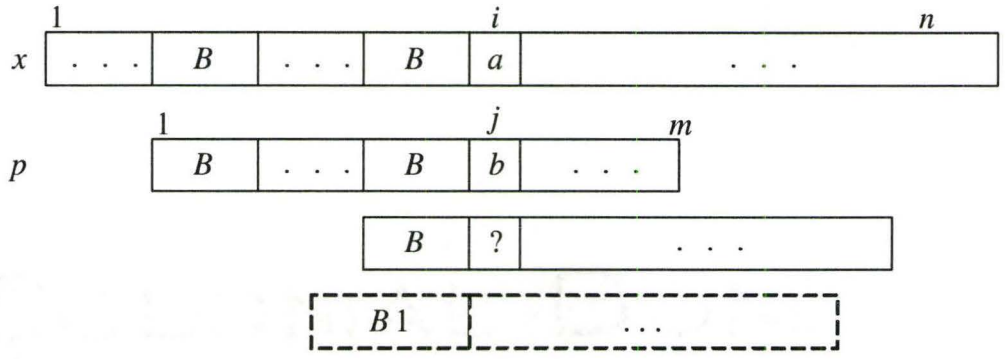

Figure 3.5: KMP Shifts

this amount (effectively by assigning $MP\_Next[j]$ to j.)

Knuth made a point that if we know in advance $p[\beta[j-1]+1] = p[j]$ — in other words, if we know in advance that after shifting, the letter we want to compare is identical to the previous one — then it must be different to $x[i]$. In this case we can shift p further to the right iteratively until it is shifted to a new position where $p[j'] \neq p[j]$. So the KMP [KMP77] algorithm actually precomputes a different array called KMP_Next, and shifts the pattern to the right according to this array.

Given a string x and its border array β, the KMP_Next array can be computed as Figure 3.6 shows.

Figure 3.7 describes the KMP algorithm.

Now let's analyze the time complexity of the KMP algorithm. Note that during the entire procedure of KMP, when we compare $x[i]$ and $p[j]$, if two letters match, then both i and j increase. If two letters do not match, then i stays the same, and j

Algorithm of KMP_Next array calculation

Input: *String* $x[1..n], \beta[1..n]$
Output: *String* $KMP\_Next[1..n+1]$

```
Array : KMP_Next[1..n + 1]
Integer : b, i
KMP_Next[1] ← 0
for i ← 1 to n do
    b ← β[i]
    while b > 0 and x[i + 1] = x[b + 1] do
        b ← β[b]
    if x[i + 1] ≠ x[b + 1] then
        KMP_Next[i + 1] ← b + 1
    else
        KMP_Next[i + 1] ← 0
returnKMP_Next
```

Figure 3.6: Algorithm of KMP_Next array calculation

decreases. We observe that if j decreases, then the pattern shifts to the right. So in the first case ($x[i] = p[j]$) i increases, at most n times; in the second case ($x[i] \neq p[j]$) pattern shifts to the right, at most $n - m$ times. So the maximum number of overall letter comparisons is $2n - m$.

So KMP correctly outputs the occurrences of p in x and has a worst case running time of $O(n)$.

3.2.3 The Boyer-Moore-Sunday Algorithm

Although the KMP algorithm has an excellent worst case time complexity of $O(n)$, it is known that the KMP algorithm is not a very fast algorithm in terms of practical speed. On the contrary, we now look at the BMS algorithm [Sun90] which has

Algorithm KMP

Input: *String* $x[1..n]$, $p[1..m]$
Output: Positions of occurrences of p in x

```
Integer : i, j
i ← 1; j ← 1
while i ≤ n do
   while j > 0 and p[j] ≠ x[i] do
      j ← KMP_Next[j]
   i ← i + 1
   j ← j + 1
   if j > m then
      output i − j + 1
```

Figure 3.7: Algorithm KMP

$O(mn)$ worst case running time but actually is one of the fastest determinate pattern-matching algorithms in practice. Unlike the KMP algorithm, the computation of the BMS algorithm doesn't depend on the construction of a border array.

Figure 3.8 demonstrates the basic shifting strategy of the BMS algorithm: after a mismatch or substring match, the algorithm shifts the pattern to the right so that $x[i]$ aligns with the rightmost occurrence of $x[i]$ in p. There can be no occurrence of p between the new position and the old one because there is no occurrence of $x[i]$ between these two positions.

In the preprocessing phase, algorithm BMS computes a Δ array as follows:

> For every $j \in \Sigma$, $\Delta[j] = m - l + 1$, where l is the rightmost position in p where j occurs; if j doesn't occur in p, then $\Delta[j] = m + 1$.

Thus the correct shift is ensured by computing $i + \Delta[x[i+1]]$ as the new value of

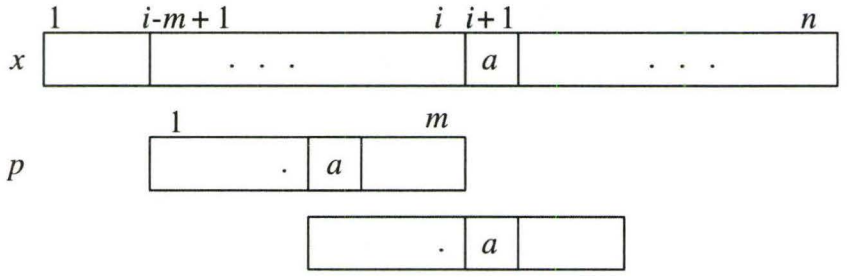

Figure 3.8: BMS Shifts

i. We describe this algorithm in Figure 3.9.

Algorithm BMS-Preprocessing

Input: *String* $p[1..m]$
Output: *Array* $\Delta[1..|\Sigma|]$

```
Integer: l
Array: Δ[1..|Σ|]
for l ← 1 to |Σ| do
    Δ[l] ← m + 1
for l ← 1 to m do
    Δ[p[l]] ← m − l + 1
```

Figure 3.9: Algorithm BMS-Preprocessing

The main procedure of BMS algorithm is described in Figure 3.10. Algorithm BMS correctly computes all the occurrences of pattern p and has a time complexity of $O(mn)$. The worst case running time is achieved by pathological inputs such as $x = a^n, p = a^m$. For further discussion, see [Smy03].

Algorithm BMS

Input: **String** $x[1..n]$, $p[1..m]$
Output: Positions of occurrences of p in x

```
Integer: i, k
i ← m; j ← 1;
while i ≤ n do
    k ← StringComparison(x[i − m + 1..i], p)
    if k = m + 1 then
        output i − m + 1;
    else
        i ← i + Δ[x[i + 1]]
```

Figure 3.10: Algorithm BMS

3.2.4 The Shift-Or Algorithm

The Shift-Or algorithm [Döm68] [BYG92] [WM92] is yet another exact pattern-matching algorithm. It has been widely used in pattern-matching programs such as Unix `agrep` [AGR]. Moreover, it has a special property that after slight modification, it will be able to perform pattern-matching on a pattern string in which each position may be a set of characters instead of just one character. We will discuss this feature later in this thesis, but for now we describe the standard Shift-Or algorithm as follows:

Preprocessing

For each j in the alphabet Σ, the Shift-Or algorithm computes a bit array S_j of size m, such that for $\forall i \in 1..m$, $S_j[i] = 0$ iff $p[i] = j$, otherwise $S_j[i] = 1$. For example, if alphabet $\Sigma = \{A, C, G, T\}$ and $m = AATCG$, Table 3.2 illustrates the bit arrays

preprocessed by ShiftOr. For instance the bit array S_A for letter A is 00111. Note that in general the length of such a bit array is $\lceil m/w \rceil$ (w is the system word length) .

$m\backslash\Sigma$	A	C	G	T
A	0	1	1	1
A	0	1	1	1
T	1	1	1	0
C	1	0	1	1
G	1	1	0	1

Table 3.2: Preprocessing of S in ShiftOr

Computation

The main procedure of the ShiftOr algorithm is shown in Figure 3.11.

Algorithm Shift-Or

Input: *String* $x[1..n]$, $p[1..m]$
Output: Positions of occurrences of p in x

```
Preprocessing();
BitArray: R[1..n]
R[1] ← S_{x[1]}
for i = 2 to n do
    R[i] ← Shift(R[i − 1]) ∨ S_{x[i]};
    if R_m[i] = 0 then output i − m + 1;
```

Figure 3.11: Algorithm Shift-Or

We illustrate an example in Table 3.3, where $x = ACAATCGT$ and $m = AATCG$. Note that for every $x[i]$ ShiftOr construct a bit array $R[i]$, which is represented by a column in Table 3.3. $R[i]$ is calculated from $\texttt{Shift}(R[i-1]) \vee S_{x[i]}$,

where both `Shift` and $\vee$ are ordinary bitwise operations supported by many computer systems. The shift is performed from the least to the most significant bit, and the new spaces are filled with 0. For instance, the second column (11111) is calculated first by performing `Shift`(11100) which yields (11000), and then by performing logical disjunction with S_C in Table 3.2, which is (10111), and finally yields (11111). If $R_m[i] = 0$ then it indicates that there is an occurrence in position $i - m + 1$, otherwise it indicates a mismatch.

$m\backslash x$	A	C	A	A	T	C	G	T
A	0	1	0	0	1	1	1	1
A	0	1	1	0	1	1	1	1
T	1	1	1	1	0	1	1	1
C	1	1	1	1	1	0	1	1
G	1	1	1	1	1	1	0	1

Table 3.3: The bit arrays R in ShiftOr

Algorithm Shift-Or correctly computes all the occurrences of pattern string p and has a worst case time complexity of $O(nm/w)$ where w is the machine word length. For proofs, see [Smy03].

3.2.5 The Franek-Jennings-Smyth Algorithm

As we have discussed previously, the KMP algorithm has an excellent worst case running time but in practice is not very fast; on the contrary, the BMS algorithm

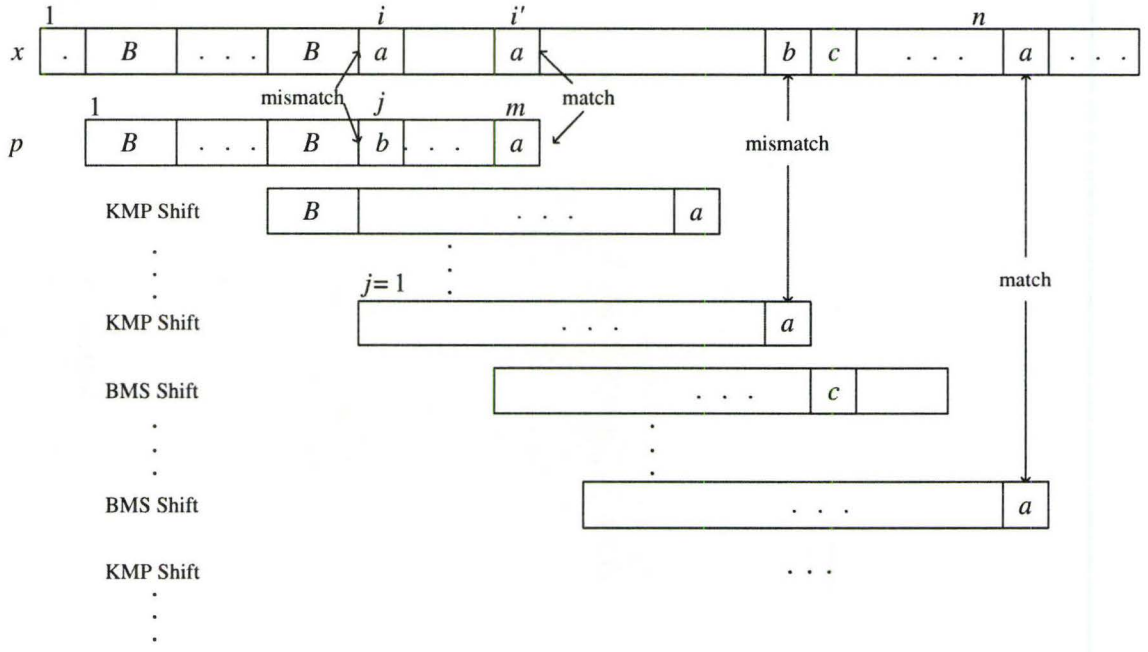

Figure 3.12: FJS Shifts

performs well on average but in the worst case is not very efficient. The recent FJS algorithm [Jen02, FJS05b, FJS05a] is an attempt to combine the merits of these two. It has a practical running time faster than that of the BMS algorithm while the worst case time complexity is almost as good as KMP.

The FJS algorithm has two kinds of shifts.

- **BMS Shift**. Similar to BMS, FJS first compares $p[m]$ with the corresponding $x[i']$. If a mismatch occurs, a "BMS shift" is implemented, moving p along x until $p[m] = x[i']$.

- **KMP Shift**. If $p[m] = x[i']$, KMP pattern-matching begins, starting from the

Algorithm FJS

Input: **string** $x[1..n], p[1..m]$
Output: Positions of occurrences of p in x

```
Integer : i, j, i', m'

— Initialization
if m < 1 then return
j ← 1; i ← 1; m' ← m − 1; i' ← m;

while i' ≤ n do

  if j ≤ 1 then

    — If no partial match with p, perform Sunday shifts,
    — Returning other position i' such that x[i'] = p[m]
      SUNDAY-SHIFT(i')

    — Reset invariant for KMP matching of p[m − 1]
      j ← 1; i ← i' − m';

      KMP-MATCH(m'; j, i)

  else

    — Continue KMP matching of p[1..m]
      KMP-MATCH(m; j, i)

    — Restore invariant for next entry SUNDAY or KMP
      j ← β'[j]; i' ← i + m − j
```

Figure 3.13: Algorithm FJS.

left-hand end $p[1]$ of the pattern and, if no mismatch occurs, extending as far as $p[m-1]$. Then, whether or not a match for p is found, KMP shifts are iteratively performed until $j = 1$, followed by a return to a BMS Shift.

We illustrate the above procedure in Figure 3.12 and describe the algorithm in Figures 3.13 and 3.14.

In [FJS05a] it is shown that the maximum number of letter comparisons of FJS

KMP-MATCH$(m'; j, i)$

while $j < m$ **and** $x[i] = p[j]$ **do**
  $i \leftarrow i + 1; j \leftarrow j + 1;$
if $j = m$ **then**
  $i \leftarrow i + 1; j \leftarrow j + 1;$
  output $i - m;$

KMP-MATCH$(m; j, i)$

while $j \leq m$ **and** $x[i] = p[j]$ **do**
  $i \leftarrow i + 1; j \leftarrow j + 1;$
if $j > m$ **then**
  output $i - m;$

SUNDAY-SHIFT(i')

while $x[i'] \neq p[m]$ **do**
  $i' \leftarrow i' + \Delta[x[i' + 1]]$
if $j' > n$ **then return**

Figure 3.14: Subroutines of FJS

is $O(3n - 2m)$, a bound obtained by $p = aba, x = a^n$.

Chapter 4

Indeterminate Pattern Matching algorithms

An indeterminate string is a relatively new concept, there haven't been many algorithms published yet that deal with indeterminate-string operations including pattern-matching. In this chapter we discuss three ways of performing indeterminate pattern-matching. The first is a group of algorithms based on the BMS algorithm that we introduced in chapter 3. This group of algorithms is also detailed in [HSW05a, HSW05b]. The second one is a group of algorithms modified from Shift-Or algorithm, which is the underlying algorithm of the well-known Unix `agrep` utility. Finally we discuss the problems and difficulties that arise when the FJS algorithm is used as a basis to perform pattern-matching on indeterminate letters, and we give details in section 3 of this chapter.

There are many models that can be used to characterize different versions of indeterminate pattern-matching. We identify three models as follows:

- Types of indeterminate letters. We can allow only the don't care letter $*$ ($*$-version) or general indeterminate letters (i-version).

- Existence of indeterminate letters. We may require or have the knowledge that indeterminate letters only occur in the pattern string (p-version) or both pattern and text (b-version).

- Constraints of indeterminate matching. We may perform indeterminate matching with no constraint (q-version for "quantum"), or with local constraint (d-version for "determinate").

Note that these three categories are orthogonal and can be used as specifications of our algorithms. For example, $*pq$ means letter '$*$' is the only indeterminate letter, it only appear in pattern and there is no constraint posed on matching. On the other hand, *ibd* means we allow general indeterminate letters, they appear in both text and pattern, and we require local constraint posed on matching.

In our following description of indeterminate algorithms we describe them in order of increasing sophistication, that is, we start from the simplest model, then move on to more sophisticated ones.

4.1 Indeterminate BMS algorithms

4.1.1 $*pq$BMS

First let's consider the simplest case. In addition to an alphabet Σ of normal determinate letters, we also define an extended alphabet Σ' that contains only the don't care letter $*$. In other words, $\Sigma' = \Sigma \cup \{*\}$. We don't require any indeterminate constraints on pattern-matching; and all the indeterminate letters, in this case just the symbol $*$, occur only in the pattern string. Let's call this version of indeterminate BMS algorithm $*pq$BMS (in accordance with the notation we gave previously).

By slightly modifying the preprocessing phase and function `StringComparison` in Figure 3.2, determinate BMS can be easily transformed to $*pq$BMS. The main procedure is the same as the one described in Figure 3.10. We just discuss the preprocessing and function `StirngComparison` as follows:

Preprocessing Δ

First let's see how the Δ array in $*pq$BMS is computed. We start by examining the purpose of the Δ array in the determinate BMS algorithm: shift pattern p to the right to a position so that there won't be any occurrences "skipped" by that shift. The BMS algorithm stores the rightmost occurrence of j for each $j \in \Sigma$ so that when a mismatch occurs it will align this position with $x[i+1]$ with no occurrence "skipped". In $*pq$BMS, the Δ array is computed as follows:

Let ℓ_* be the position of the rightmost occurrence of letter $*$ in p. $\forall j \in 1..k$, let ℓ_j be the position of the rightmost occurrence of j in p. If j doesn't appear in p, then $\ell_j = 0$. If $\ell_j > \ell_*$ then $\Delta[j] = m - l + 1$, otherwise $\Delta[j] = m - l_* + 1$.

And this can be implemented as follows:

Algorithm $*pq$BMS-Preprocessing

Input: *String* $p[1..m]$
Output: *Array* $\Delta[K]$

```
Integer: ℓ
Array: Δ[K]
for j ← 1 to K do
    Δ[j] ← 0
for ℓ ← 1 to m do
        Δ[p[ℓ]] ← ℓ
for j ← 1 to k do
    if Δ[j] > Δ[K] then
        Δ[j] ← m − Δ[j] + 1
    else
        Δ[j] ← m − Δ[K] + 1
```

Figure 4.1: Algorithm $*pq$BMS-Preprocessing

Notice here we use $\Delta[K]$ to store the value of ℓ_*, and because we assume in this model that the indeterminate letter '$*$' never occurs in the text, $\Delta[K]$ will not be accessed during the matching. From our algorithm we can see preprocessing can be performed in $O(2k + m)$ time.

Function `StringComparison`

It is easy to see that function `StringComparison` for $*pq$BMS should be changed as follows:

Function $*pq$`BMS-StringComparison`

Input: *String* $u[1..n]$, $v[1..n]$
Output: *Integer* k which is the smallest integer such that $u[k] \neq v[k]$ or $k = m + 1$ in case of two strings match each other completely.

Integer: i
$i \leftarrow 1$
while $((x[i] = p[i]$ **or** $p[i] =$ '$*$' $)$ **and** $i \leq m)$ **do**
  $i \leftarrow i + 1$
output i

Figure 4.2: Function $*pq$`BMS-StringComparison`

4.1.2 $*bq$BMS

Preprocessing Δ

In $*bq$BMS, don't care letters are allowed in both text and pattern. As a consequence, $\Delta[K]$ will possibly be accessed and the proper value should be 1. So in the preprocessing phase, in addition to the commands in Figure 4.1, the following line should also be added at the end:

$$\Delta[K] \leftarrow 1$$

That means if $x[i + 1] =$ '$*$' then we should shift just 1 position to the right.

Function `StringComparison`

Correspondingly `StringComparison` should be changed as follows:

Function $*bq$`BMS-StringComparison`

Input: *String* $u[1..n]$, $v[1..n]$
Output: *Integer* k which is the smallest integer such that $u[k] \neq v[k]$ or $k = m + 1$ in case of two strings match each other completely.

Integer: i
$i \leftarrow 1$
while $((x[i] = p[i]$ **or** $p[i] =$ '$*$' **or** $x[i] =$ '$*$')
  $i \leftarrow i + 1$
output i

Figure 4.3: Function $*bq$`BMS-StringComparison`

4.1.3 *ipq*BMS

Now let's have a look at a model that allows general indeterminate letters, but only in pattern p. Note that this model corresponds exactly to the `agrep` utility in the Unix system that we have already mentioned.

In section 2.3 we gave the definition of matching of indeterminate letters for three cases. That is, the case in which both i, j are determinate letters, the case in which exactly one of i, j is an indeterminate letter and finally the case in which both i, j are indeterminate letters. Clearly in this model the second case applies. So if $i \leq k, j > k$, then $i \approx j \iff i \in \Sigma_j$. Note that up to this point we have only been dealing with the determinate alphabet Σ. Now for *ipq*BMS we have to be clear what exactly do

Σ_j and $i \in \Sigma_j$ mean.

First we need to know how a regular alphabet Σ is represented in a computer system. For example, an ASCII alphabet is actually stored in the computer system in the form of a lookup table. Each letter in the alphabet is associated with a number so that we can use it as an index to refer to a specific entry of an array. Essentially inside a computer system, letters in the alphabet are always represented in the form of numbers. It will only be transformed to a graphical symbol if there is human-computer interaction involved. The transformation between the symbol of a letter and its associated number is automatically performed by the compiler. For instance, when a compiler takes some source code that contains $\Delta[p[j]]$, first it transforms ASCII letter j into a number, uses this number as an index to get the entry in array p, then again transforms the entry into another number, then gets the entry in array Δ using this new number. It is also possible that it will transform the result back to a graphical symbol if it needs to send the result to an output device such as a monitor. So we should keep in mind that every compiler uses a coding system (ASCII in most cases) which defines an alphabet. It uses lookup tables to transform between graphical representation of a letter in the alphabet and its associated number.

The alphabet defined by a coding system is ***implicit*** in some algorithms and ***explicit*** in others. For example, in the KMP algorithm, the alphabet of the system is implicit. We don't need to know how many letters exist in the alphabet and what

they are, the only thing we require is that given $\sigma_1, \sigma_2 \in \Sigma$, it is possible to decide if $\sigma_1 = \sigma_2$ in constant time. In other words, the KMP algorithm requires a general alphabet, which we gave definition in Chapter 2, and it is implicit.

However, in the BMS algorithm, the alphabet of the system is explicit. There is a copy of all the letters in the system alphabet stored in array Δ. Moreover, we require that every $\sigma \in \Delta$ can be used as an index in an array. So the BMS algorithm actually uses an indexed alphabet, and it is explicit.

For algorithms dealing with indeterminate letters, we also need to consider how an indeterminate alphabet should be properly represented in a computer system. Without loss of generality, we assume all determinate letters are defined by the system using its default alphabet and each indeterminate letter $j > k$ are given by users using linked lists as input. So for every indeterminate letter j, there's a linked list Σ_j such that j is the head of the list and every $j' \in \Sigma_j$ is a node in the list.

Preprocessing Δ

Now having defined how an indeterminate alphabet is represented in the system, we are ready to look at the algorithmic solution for *ipq*BMS. First let us discuss the preprocessing in *ipq*BMS. We present two methods as follows.

The first approach is a natural modification of the preprocessing of exact BMS in Figure 3.9. As we have known, during exact BMS matching, if $x[i+1] = a$ (see Figure 3.8), $\Delta[a]$ stores the length that the pattern should shift to the right. It is guaranteed

that this length is the largest length where there is no occurrence of letter a between the original position and the new position after shifting. So in exact BMS we only have to go through the pattern p from left to right, increasing ℓ by one in every step, and store $m - \ell + 1$ in $\Delta[p[l]]$ (See Figure 3.10). Note that if a letter appears in the pattern more than once, only the rightmost one is stored in Δ.

In *ipq*BMS however, there are indeterminate letters in the pattern, which makes things a little bit more complicated. If an indeterminate letter appears in position $p[\ell]$, we have to update Δ for every $j \in \Sigma_{p[\ell]}$. The reason is obvious: during the matching, suppose $x[i+1] = j$, and we have an indeterminate letter in position $p[\ell]$, where $j \in \Sigma_{p[\ell]}$, then the pattern can be shifted to the right at most by aligning $p[\ell]$ with $x[i+1]$! Therefore the preprocessing should be as Figure 4.4 shows.

Algorithm *ipq*BMS-Preprocessing1

Input: *String* $p[1..m]$
Output: *Array* $\Delta[K]$

```
Integer: ℓ
Array: Δ[K]
for j ← 1 to k do
    Δ[j] ← 0
for ℓ ← 1 to m do
    if p[ℓ] ≤ k then
        Δ[p[ℓ]] ← ℓ
    else
        ∀j ∈ Σ_p[ℓ]
            Δ[j] ← ℓ
for j ← 1 to k do
    Δ[j] ← m − Δ[j] + 1
```

Figure 4.4: Algorithm *ipq*BMS-Preprocessing1

The time complexity for the above preprocessing approach is

$$O(2k + \sum_{\ell=1}^{m} |\Sigma_{p[\ell]}|).$$

In the worst case, the time complexity is $O(2k + mk)$.

An alternative preprocessing approach is shown in Figure 4.5. In this approach, as usual we first go through p from left to right once, but indeterminate letters in the pattern are first handled in the same manner as determinate letters (namely, $\Delta[p[\ell]] \leftarrow \ell$ for very position ℓ). After that, all indeterminate letters in the alphabet are gone through once again, and all associated determinate letters are updated. This approach and the previous one both correctly compute Δ, they differ only in the ordering of processing indeterminate letters.

Algorithm *ipq*BMS-Preprocessing2

Input: *String* $p[1..m]$
Output: *Array* $\Delta[K]$

```
Integer: ℓ
Array: Δ[K]
for j ← 1 to K do
    Δ[j] ← 0
for ℓ ← 1 to m do
    Δ[p[ℓ]] ← ℓ
for j' ← k to K do
    ∀j ∈ Σ_{j'}
        if Δ[j] < Δ[j'] then
            Δ[j] ← Δ[j']
for j ← 1 to k do
    Δ[j] ← m − Δ[j] + 1
```

Figure 4.5: Algorithm *ipq*BMS-Preprocessing2

The time complexity for this approach is $O(K+k+m+\sum_{j'=k}^{K}|\Sigma_{j'}|)$. In the worst case, the time complexity is $O(K+k+m+(K-k)k) = O(2k+(K-k)+m+(K-k)k)$. From the analysis of the time complexity we can see that preprocessing1 will be faster if $m \ll (K-k)$.

After preprocessing Δ, for the rest of the solution we present two approaches that are based on two different data-structures, which are bit array and matrix respectively.

Solution 1 - Bit Array Solution

In this solution we use bit arrays as our data structure to decide whether two letters match each other or not. We construct a bit array $b_j[1..k]$ for each $j \in 1..K$ such that:

- for $j \leq k, b_j[j'] = 1 \Longleftrightarrow j' = j$;
- for $j > k, b_j[j'] = 1 \Longleftrightarrow j' \in \Sigma_j$.

The time we need to construct this bit array is

$$O(k + \sum_{j=k}^{K} |\Sigma_j|).$$

Function `StringComparison`

The function StringComparison should be changed as follows:

$i \leftarrow 1$
while $(b_{x[i]} \wedge b_{p[i]}$ **and** $i \leq n)$ **do**
  $i \leftarrow i + 1$

output i

Therefore, to compare two indeterminate letters, the time cost will be $O(k/w)$, where w is the system word size. However, to avoid unnecessary use of bit array, which is slow in terms of speed, especially when $w \ll k$, the following code may be more desirable:

Function *ipq*`BMS1-StringComparison`

Input: *String* $u[1..n]$, $v[1..n]$
Output: *Integer* k which is the smallest integer such that $u[k] \neq v[k]$ or $k = m + 1$ in case of two strings match each other completely.

```
Integer: i
Boolean : match
i ← 0
do
    i ← i + 1
    if (p[i] ≤ k) then
        match ← (x[i] = p[i])
    else
        match ← (b_{x[i]} ∧ b_{p[i]})
while (match = true and i ≤ n)
output i
```

Figure 4.6: Function *ipq*`BMS1-StringComparison`

That means, we use bit array to decide whether two letters match or not only when the pattern letter is an indeterminate letter.

Note that another option is to access the individual bit $b_{p[i]}[x[i]]$ instead of performing a bit operation $b_{x[i]} \wedge b_{p[i]}$ to decide whether two indeterminate letters match or not. However, although efficient in theory, random bit accessing can be expensive in practice and difficult to express in some languages such as C [WIK].

Solution 2 - Matrix Solution

Solution 1 uses a bit array in order to determine whether two indeterminate letters match each other or not. But if the number of indeterminate letters in the alphabet is small, and these indeterminate letters occur frequently in the pattern, then it may be more desirable to precompute the results of letter comparison testing in solution 1 and store it in a matrix so that we can access the result in constant time.

Precompute Matrix

We construct a $(K-k) \times K$ matrix M such that:

- for $j \leq k, M[i][j] = 1 \Longleftrightarrow j \in \Sigma_i$.
- for $j > k, M[i][j] = 1 \Longleftrightarrow \Sigma_i \cap \Sigma_j \neq \emptyset$.

Function `StringComparison`

Similarly, for function `StringComparison`, we can have a process as Figure 4.7 shows.

We can see that solution 2 uses more space and spends more time in the pre-processing phase (because we have to compute $\Sigma_i \cap \Sigma_j$ for all $i, j \in k+1..K$), but in the case that the number of indeterminate letters $(K-k)$ in the alphabet is small, solution 2 is expected to be faster than solution 1.

Function *ipq*`BMS2-StringComparison`

Input: *String* $u[1..n]$, $v[1..n]$
Output: *Integer* k which is the smallest integer such that $u[k] \neq v[k]$ or $k = m + 1$ in case of two strings match each other completely.

```
Integer: i
Boolean : match
i ← 0
do
   i ← i + 1
   if (p[i] ≤ k) then
      match ← (x[i] = p[i])
   else
      match ← (M[p[i]][x[i]] = 1)
while (match = true and i ≤ n)
output i
```

Figure 4.7: Function *ipq*`BMS2-StringComparison`

4.1.4 *ibq*BMS

In this model general indeterminate letters can occur in both pattern and text.

Preprocessing Δ

Similar to *ibq*BMS we present two approaches to preprocessing Δ as follows.

First a natural approach is described in Figure 4.8. Like all previous discussed BMS-derived algorithms, the aim of preprocessing Δ is to store the proper length that the pattern should shift to the right when a mismatch occurs. So like *ipq*BMS-Preprocessing1 we first go through m from left to right once, for every position ℓ we update the corresponding determinate letters by assigning $\Delta[j] \leftarrow \ell$ for every $j \in \Sigma_{p[\ell]}$. Because indeterminate letters can also appear in text, for every position ℓ

we need to update all the indeterminate entries in Δ as well. If $b_j \wedge b_{p[\ell]} \neq \emptyset$ while $j \in k+1..K$, then it means entry $\Delta[j]$ should be updated by ℓ. For all j, using the bit array encoding b_j of Σ_j, the time requirement is $O(2K + m(k + k(K-k)/w))$.

Algorithm *ibq*BMS-Preprocessing1

Input: *String* $p[1..m]$
Output: *Array* $\Delta[K]$

```
Integer: j, ℓ
Array: Δ[K]
for j ← 1 to K do
  Δ[j] ← 0
for ℓ ← 1 to m do
  ∀ j ∈ Σ_{p[ℓ]} do
    Δ[j] ← ℓ
  for j ← k+1 to K do
    if b_j ∧ b_{p[ℓ]} ≠ ∅ then
      Δ[j] ← ℓ
for j ← 1 to K do
  Δ[j] ← m − Δ[j] + 1
```

Figure 4.8: Algorithm *ibq*BMS-Preprocessing1

For m large with respect to k, an alternative and more complicated preprocessing may be preferable. Similar to algorithm *ipq*BMS-preprocessing2, in this preprocessing approach we first go through p without further processing indeterminate letters. Then after that, we go through every indeterminate letter in the alphabet, updating every letter that matches with them. Note that it is necessary to use an auxiliary array Δ_0 that stores the "original" rightmost position of every letter in p, because of the non-transitivity of indeterminate letters ($(a \approx b) \wedge (b \approx c) \not\Rightarrow (a \approx c)$). The details of the algorithms are shown in in Figure 4.9. The time requirement is now $O(2K +$

Algorithm *ibq*BMS-Preprocessing2

Input: *String* $p[1..m]$
Output: *Array* $\Delta[K]$

```
Integer: j, ℓ
Array: Δ0[K], Δ[K]
for j ← 1 to K do
  Δ[j] ← 0
for ℓ ← 1 to m do
  Δ[p[ℓ]] ← ℓ
for j' ← 1 to K do
  Δ0[j'] ← Δ[j']
for j' ← k+1 to K do
  for j ← 1 to K do
    if b_j ∧ b_j' ≠ ∅ then
      if Δ[j] < Δ0[j'] then
        Δ[j] ← Δ0[j']
      elsif Δ0[j] > Δ[j'] then
        Δ[j'] ← Δ0[j]
for j ← 1 to K do
  Δ[j] ← m − Δ[j] + 1
```

Figure 4.9: Algorithm *ibq*BMS-Preprocessing2

$m+Kk(K-k)/w)$.

Function `StringComparison`

The function `StringComparison` is the same as that of *ipq*BMS.

4.1.5 *ipd*BMS

In this model, general indeterminate letters appear only in the pattern, and we require local constraints applied to the matching. *ipd*BMS and *ibd*BMS are very similar, so we skip details of *ipd*BMS and move on to next section, which describe *ibd*BMS — the most general model of all we have discussed.

4.1.6 *ibd*BMS

In this model we require that the assignment of an indeterminate string should be consistent. In other words, we require the local constraint to be imposed on the matching. The definition of local constraint is given in Chapter 2, which requires

$$\Sigma_j \cap \Sigma_{k_1} \cap \Sigma_{k_2} \cap \Sigma_{k_3}, ..., \cap \Sigma_{k_t} \neq \emptyset,$$

where j is any indeterminate letter, and $k_1, k_2, ..., k_t$ are all the letters j aligns with (either in the text or the pattern). We present our solution as follows.

Preprocessing Δ

The preprocessing of the Δ array is the same as that of *ibq*BMS.

Function `StringComparison`

In order to assure that the assignment of an indeterminate string is consistent with bitarray b we introduce a new auxiliary bitset $current[k+1..K]$, and an integer array $right[k+1..K]$ for every indeterminate letter j.

Bitarray b stores the same content as previously described algorithms, namely,

- for $j \leq k, b_j[j'] = 1 \iff j' = j$;
- for $j > k, b_j[j'] = 1 \iff j' \in \Sigma_j$.

When a `StringComparison` function proceeds from left to right along the pattern p, bitarray $current_j$ stores the intermediate result of $\Sigma_j \cap \Sigma_{k_1} \cap \Sigma_{k_2} \cap \Sigma_{k_3}, ..., \cap \Sigma_{k_t}$.

Function *ibd*`BMS-StringComparison`

Input: *String* $u[1..n]$, $v[1..n]$, p_0
Output: *Integer* k which is the smallest integer such that $u[k] \neq v[k]$ or $k = m + 1$ in case of two strings match each other completely.

```
Integer: i, j
Boolean : match
Array : right[K]
Bitarray : b, current, vector

i, j ← 1
match ← true
while (match = true and i ≤ m)
    j ← x[i]; j' ← p[i]
    if (j ≤ k and j' ≤ k) then
      match ← (j = j')
    else
      if j > k and right[j] = p0 then
        B1 ← current[j]
      else
        B1 ← bj
      if j' > k and right[j'] = p0 then
        B2 ← current[j']
      else
        B2 ← bj'
      vector ← B1 ∧ B2
      if vector ≠ 0 then
        match ← true
        if j > k then
          right[j] ← p0; current[j] ← vector
        if j' > k then
          right[j'] ← p0; current[j'] ← vector
      else match ← false
  i ← i + 1
  output i
```

Figure 4.10: Function *ibd*`BMS-StringComparison`

If at anytime $current[j] = 0$ then it means a mismatch occurs or the local constraint has been violated. At the beginning (or end) of every function `StringComparison` $current[j]$ should be reset to b_j. However, this approach is not efficient (a bit array may span several system words) and can be replaced by using an array *right*.

Entry $right[j]$ records the rightmost position where indeterminate letter j matches with other letters. So we don't need to reset $current_j$ in every `StringComparison`. Instead we need only to check if $right[j] = i - m + 1$ when comparing two letters. If $right[j] = i - m + 1$ then it means we should use b_j because j hasn't been compared so far in this position and $current_j$ stores the intermediate result from previous matching, which is irrelevant to the current one. Otherwise we should use $current_j$ because it means j has been compared with other letters in the current position, and $current_j$ stores the correct intermediate result.

If p_0 represents the "current" position $i - m + 1$ during the pattern-matching, the function `StringComparison` is as Figure 4.10 shows.

4.2 Indeterminate Shift-Or Algorithms

In [BYG92] and [WM92] it is shown that the ShiftOr algorithm is also capable of handling matching of a set of characters, by slightly modifying the construction of the S array in the preprocessing phase. Suppose we are looking for pattern $p[1..m]$ in text x, where each element $p[j], j \in 1..m$ represents "a set of" letters [WM92]. Letter $x[i]$

matches $p[j]$ iff $x[i] \in \Sigma_{p[j]}$. Note that the concept of "a set of" letters correspond to our definition of indeterminate letters, and the standard ShiftOr corresponds exactly to the pq model we have defined at the beginning of this chapter. Two cases arise, one for the don't care case ($*pq$) and another for general indeterminate letters (ipq). We discuss them respectively as follows:

4.2.1 $*pq$ShiftOr

First we discuss a simple model $*pq$ShiftOr. We can see there is a simple solution by slightly modifying the preprocessing of original ShiftOr as follows.

> For each $j \in \Sigma$, construct a bit array S_j of size m, such that for $i \in 1..m$, $S_j[i] = 0$ iff $j = p[i]$ or $p[i] =$ '$*$', otherwise $S_j[i] = 1$.

4.2.2 ipqShiftOr

In order to compute matching on a set of characters, the preprocessing phase should be modified as follows:

> For each $j \in \Sigma$, construct a bit array S_j of size m, such that for $i \in 1..m$, $S_j[i] = 0$ iff $j \in \Sigma_{p[i]}$, otherwise $S_j[i] = 1$.

The rest of the algorithm is identical to the regular one.

4.3 Indeterminate FJS Algorithms

Because of the non-transitivity of indeterminate letters, the border array of an indeterminate string cannot be used directly to perform pattern-matching. We give two examples as follows and analyze the reason why difficulty arises when we try to perform indeterminate pattern-matching using border arrays.

First we consider an example illustrated in Table 4.1. In this example letter '$*$' is the only indeterminate letter and it only appears in the pattern.

Index		1	2	3	4	5	6	7	
x	.....	a	a	b	b	a	b	b	.....
p		a	$*$	$*$	b	a	$*$	a	.....
$1st$ Shift				a	$*$	$*$	b	a	.....
$2nd$ Shift						a	$*$	$*$	.....
$3rd$ Shift							a	$*$	.....

Table 4.1: First example of the non-transitivity effect

We can see that we have a partial match in position 1..6, and a mismatch in position 7. According to the definition, the longest border of $p[1..6]$ is $a**b$, the second-longest border is $a*$ and the third is a. Like the determinate case when we perform shifts we consider borders in order of decreasing length. That is, we first consider the longest border, if mismatch still occurs, then we move on to the second-longest border, and so on. The shifted positions are listed in the table. However, we observe that if we perform a shift according to the longest border, which aligns $p[1..4]$ with $x[3..6]$, then we will have letter a aligned with b in position 3. That means

shifting the pattern to the right according to its longest border cannot guarantee a resulting partial match, which is not the case in determinate strings. Moreover, we can see that between the $1st$ shift and the $2nd$ shift a border of length 3 which is the longest border of substring $a**b$ is missed. This reveals another property that only occurs in indeterminate strings: a border of a border of x is not necessarily a border of x. In other words, Theorem 3.1.5 does not hold for indeterminate strings.

It is easy to see that these two properties also hold when indeterminate letters appear both in text and pattern, and hold for general indeterminate letters as well.

Now let us look at another example in Table 4.2, in which case indeterminate letters appear only in text.

Index		1	2	3	4	5	6	7	
x	.....	a	b	a	$*$	a	$*$	a	.....
p		a	b	a	a	a	b	b	.....
Wrong Shift						a	b	a	.....
Correct Shift				a	b	a	a	a	.....

Table 4.2: Second example of the non-transitivity effect

We can see that the length of the longest border of substring $p[1..6]$ is 2. But if we shift the pattern p to the right according to its longest border by 6-2=4, then we will miss a possible occurrence in position 3. This is also due to the non-transitivity of indeterminate letters.

Like the previous example, it is easy to see that this property also holds when

indeterminate letters appear both in text and pattern, and holds for general indeterminate letters as well.

From the above two examples, we claim the following three properties for indeterminate pattern-matching based on border arrays:

- Shifting the pattern to the right according to the longest border cannot guarantee a partial match (from Example 1).
- A border of a border of x is not necessarily a border of x (from Example 1).
- Shifting the pattern to the right according to the longest border can miss some occurrences in between. (from Example 2).

So difficulties arise when we try to directly utilize indeterminate border array as an effective tool to perform indeterminate pattern-matching. In fact, it is unclear whether it is possible to do so. This will be an interesting topic that we would like to leave for future research. However, there is a simple solution which does not have the same time bound as FJS but will be an alternative solution to avoid using an indeterminate border array. We state it as follows.

Take Table 4.2 for example. We have a partial match in $p[1..6]$ and a mismatch in $p[7]$. According to the reason we discussed previously, we cannot shift the pattern to the right according to the longest border of $p[1..6]$. However, we can shift the pattern to the right according to the longest border of $p[1..3]$, because 3 is the rightmost

position within the border $p[1..6]$ where no indeterminate letter occurs prior to it in both pattern and text. So we can maintain a variable that stores the index of such a rightmost position and update it after each shift.

We state our algorithm informally as follows:

- In preprocessing, we construct the border array for $p[1..u]$, where u is the rightmost position in p such that no indeterminate letters occur prior to it.

- We perform FJS in the determinate case, while substituting letter comparison by bitarray operation when necessary. That is, we call `SUNDAY-SHIFT` and `KMP-MATCH` as we do in FJS. In procedure `KMP-MATCH`, assume that we have a partial match between $p[1..j]$ and $x[i..i+j-1]$ and a mismatch in position $j+1$.

- The border array shifting part is a little bit complicated. We maintain a variable v such that it is the rightmost position in $1..j$ where no indeterminate letters occur in $x[i..v]$. Then it is secure to shift the pattern to the right according to the length of the longest border of $p[1..min(u,v)]$, because there is no indeterminate letters occur in $p[1..min(u,v)]$ both in text and pattern.

- After a KMP-shift, u is unchanged while v is updated by $v \leftarrow v - w$ (w is the length of the shifting).

In summary, in this chapter we have investigated three groups of algorithms that

can potentially perform pattern-matching tasks on indeterminate strings. We developed algorithms based on BMS and showed how ShiftOr can be modified to perform some of the equivalent tasks (See Table 4.3). We also pointed out the difficulty when border-array-related algorithms such as FJS is used as a base to perform indeterminate pattern-matching. We proposed a solution for an indeterminate version of FJS and would like to leave the details for future research.

Models	**pq*	**bq*	*ipq*	*ibq*	*ipd*	*ibd*
Indeterminate BMS	√	√	√	√	√	√
Indeterminate ShiftOr	√		√			

Table 4.3: Model Table

Chapter 5

Experiments

As observed in the introduction, the bit-mapping algorithm `agrep` (what we have called ShiftOr in Chapter 4) is the only practical algorithm currently available for indeterminate pattern-matching. `agrep` also possesses more general capabilities, but viewed in this context, it corresponds exactly to our *pq* model: it does not handle local constraint matching, and it does not allow for the occurrence of indeterminate letters in the text string x.

Thus to compare like with like, we need to compare `agrep` with its corresponding equivalent. A strong point of `agrep` is that for large values of n (long text strings), it depends only slightly on pattern length and alphabet size: `agrep`'s preprocessing computes a two-dimensional bit array of size $K \times m$, but normally $Km \ll n$. On the other hand, `agrep` needs to process every position of x, something that in most cases indeterminate versions of BMS would be able to avoid.

Some factors likely to be of interest in experiments on the comparative running

times of `agrep` and indeterminate variants of BMS are as follows:

* pattern and text length;
* frequency of occurrence of pattern in text (0.1 – 1%);
* nature of the text (for example, DNA, natural language files);
* alphabet size (96 – 256);
* ratio $(K-k)/k$ of indeterminate letters to alphabet size (0 – 50%);
* frequency of indeterminate letters in p and x.

In our tests we try to determine the effects of specific factors on the behaviour of the algorithms, recognizing that in some cases there may be unexpected interactions among them.

5.1 Testing Details

5.1.1 Versions

We compare the six versions of indeterminate BMS, plus the original BMS, with ShiftOr and two of its variants that correspond to the $*pq$ and ipq models.

Thierry Lecroq's website *Exact String Matching Algorithms* [Lec] presents a collection of several dozen well-tested exact string matching algorithms. Our implementation of exact BMS and ShiftOr are adapted from this source. However, necessary modification are made.

Lecroq's implementation of ShiftOr is a very simple version that does not serve for general purposes. It does not handle pattern length greater than the system word size. The preprocessing is separate from the main function which we believe will incur function call overheads. So it can not be used directly for testing. Our implementation on the other hand handles the case that the pattern length can be greater than the system word size. For each word we use a "carry bit" so that shift operation can propagate from the least significant bit to the highest or vice versa. We realize that careless implementation will probably result in behavior change such as speed slow down so we have been trying to avoid that. When pattern size is smaller than system word size, our implementation has a very close performance to that of Lecroq's ShiftOr.

Our implementation of BMS is basically adopted from Lecroq's website (where it is called Quick Search). However, it has the same function call overhead problem. In our implementation all functions are moved inline.

In our tests all variants of indeterminate BMS and ShfitOr are implemented as described in their respective sections. We presented several solutions for Algorithm *ipq*BMS in Chapter 4. In our implementation we use solution 1 – Bitset Solution, where the preprocessing and function `StringComparison` are implemented using *ipq*BMS-Preprocessing1 in Figure 4.4 and *ipq*BMS1-StringComparison in Figure 4.6 respectively. We also tried several other preprocessing methods, but the results show

that preprocessing has very little effect on the total running time.

5.1.2 Environment

We performed tests on an application server machine (Penguin) with 4 Intel Xenon 2.4GHz CPUs running GNU/Linux and 4GB memory in total. When executing the testing program one CPU is used. We also performed tests on other hardware platforms such as desktop PC and operating systems such as Microsoft Windows: the results across these platforms are consistent.

5.1.3 Timing

We use the C++ standard library function `Clock` to time the consumption of processor time of different algorithms. Each test was repeated 20 times and the minimum was taken as the final result. We choose minimum instead of average running time because we believe minimum running time reflects the actual running time better than the average running time. In computers the measured running time can sometimes be increased by cache missed, running other background programs etc, while there is no factor that can decrease the running time. So the minimum running time reaches closely to the real running time, and that was in fact also confirmed by our experiments.

All preprocessing time was included in the final result. In order to eliminate the effect of function call overhead, all function calls were moved inline or declared as

inline functions.

5.1.4 Testing Data

The main corpus of this project was taken from Project Gutenberg [Har], which has a collection of more than 15,000 eBooks produced by hundreds of volunteers. Ten of them in different lengths were selected at random as the testing text. Another corpus of data was taken from The Human Genome Project [HGP] as an auxiliary corpus. The DNA files downloaded from [GB] contain header information and they were filtered out so that only pure DNA sequence information remained.

5.2 Test Results

There are many factors that could affect the execution time of the algorithms. In this section we present our test results and analyze these factors separately.

Text Length

All these algorithms should work well on normal texts without indeterminate letters. So first we tested all the algorithms against text length on such texts. We used English texts (ASCII alphabet with 256 symbols) obtained from [Har] and the high-frequency pattern set from [FJS05b]. The pattern set consisted of 7 patterns of length 6:

_of_th *of_the* *f_the_* *_that_* *,_and_* *_this_* *n_the_*

The results are shown in Figure 5.1. All algorithms are linear in text length. BMS has the best performance, followed by the $*pq$BMS and $*bq$BMS algorithms.

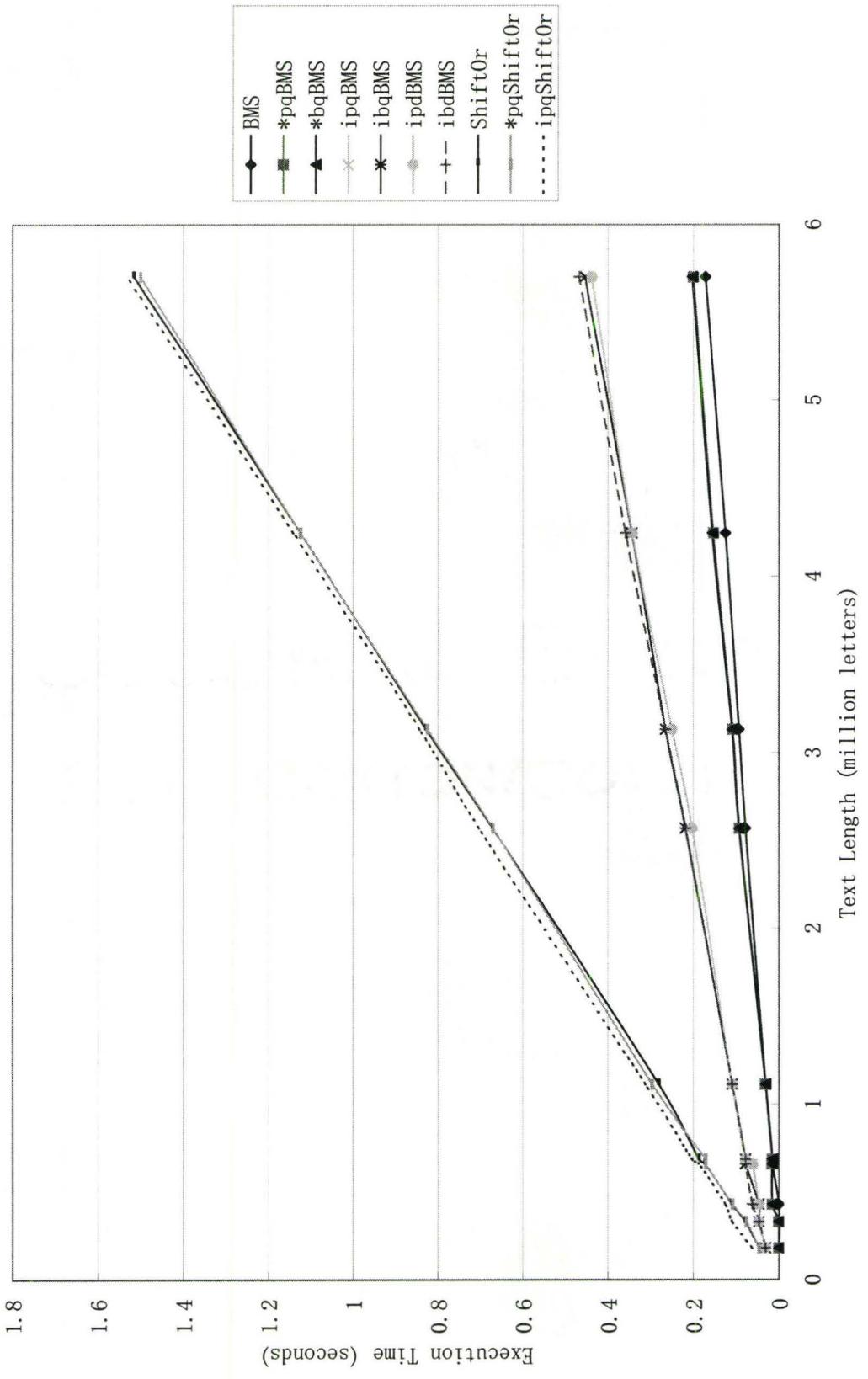

Figure 5.1: Execution time against text length on high-frequency pattern set

Next, slightly slower, come the other four indeterminate BMS algorithms, all with very similar performance. ShiftOr and its derived algorithms are by far the slowest.

Frequency of Occurrence

In this test we replace the high-frequency pattern set with a moderate-frequency pattern set (also from [FJS05b]), and run it on the same texts as in the previous test. The pattern set used in this test is as follows:

better enough govern public someth system though

The frequency of occurrence (Occurrences/Text Length) of the moderate frequency pattern set (0.05%–0.09%) is about 1/10 of the occurrence rate of the high-frequency pattern set (0.62%–0.92%).

The results are shown in Figure 5.2. The relative performances of the algorithms in this test are identical to those in the previous test (Figure 5.1). BMS-derived algorithms actually perform slightly better on the moderate-frequency pattern set, because the letters in these patterns are usually rarer than those in high-frequency patterns; thus longer shifts tend to occur.

Pattern Length

We tested using pattern sets of different lengths, consisting of ten groups of 9 patterns each. Patterns in each group are of the same size, ranging from 3 to 500 letters.

The results in Figure 5.3 demonstrate that when pattern length is less than the system word size (32 bits), execution time of all algorithms tends to stay unchanged,

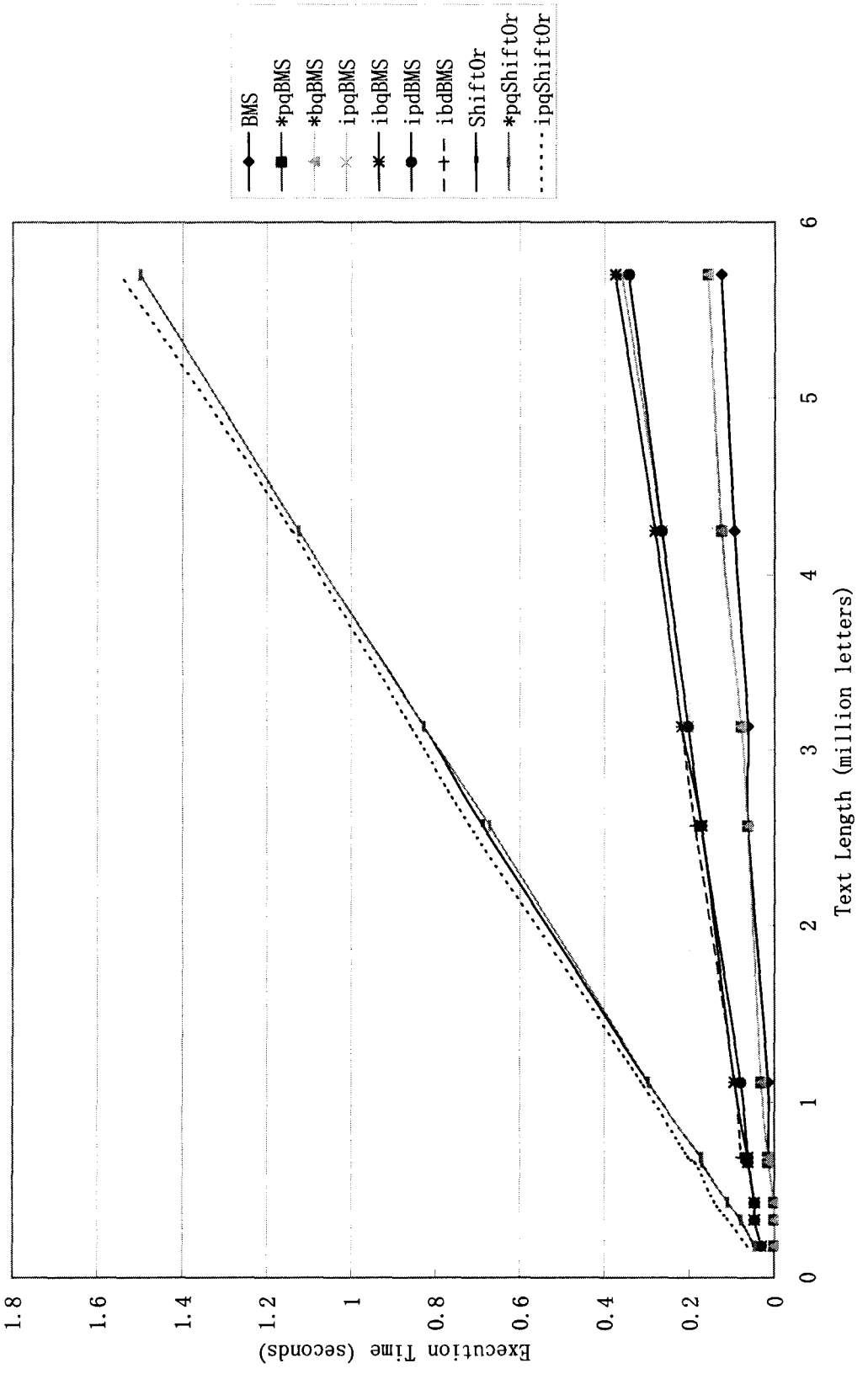

Figure 5.2: Execution time against text length on moderate-frequency pattern set

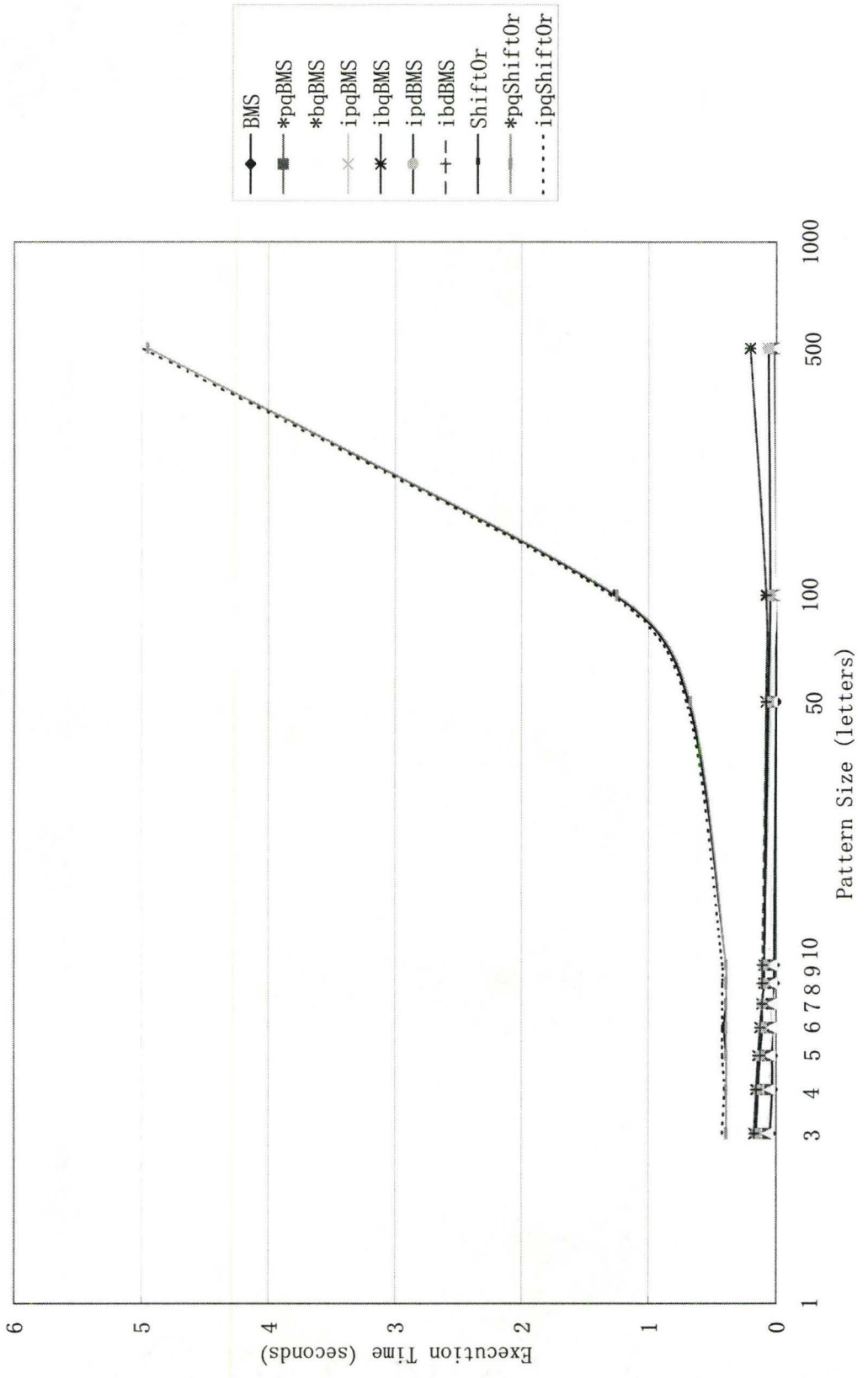

Figure 5.3: Execution time against pattern length

with BMS-derived algorithms even dropping slightly. This is because the average shift when a mismatch occurs for BMS-derived algorithms is greater when the pattern length is large. So the longer the pattern, the longer the shift when a mismatch occurs, thus the faster the algorithm.

When pattern length is greater than the system word size, there is no significant change in the running time of BMS-derived algorithms, while the running time of ShiftOr-derived algorithms increases dramatically — this reflects the fact that ShiftOr execution time $O(mn/w)$ is actually linear in pattern length.

Pathological Case

The BMS algorithm is known to have a worst case $O(mn)$ execution time when input text and pattern take the form a^n and a^m. We tested the algorithms in this pathological case, as shown in Figure 5.4. The result shows that execution time of all BMS-derived algorithms is linear in pattern size while execution time of the ShiftOr-derived algorithms remains approximately constant.

Alphabet Size

We tested the effect of alphabet size on all algorithms. We gradually increased the alphabet size, while keeping all other parameters (ie. frequency of occurrence, text and pattern length) unchanged. The results are shown in Figure 5.5. We see that four algorithms (*ipq*BMS, *ipd*IBMS, *ibq*BMS and *ibd*BMS) are affected by the increase in alphabet size, because for every letter comparison, these algorithms need to compute

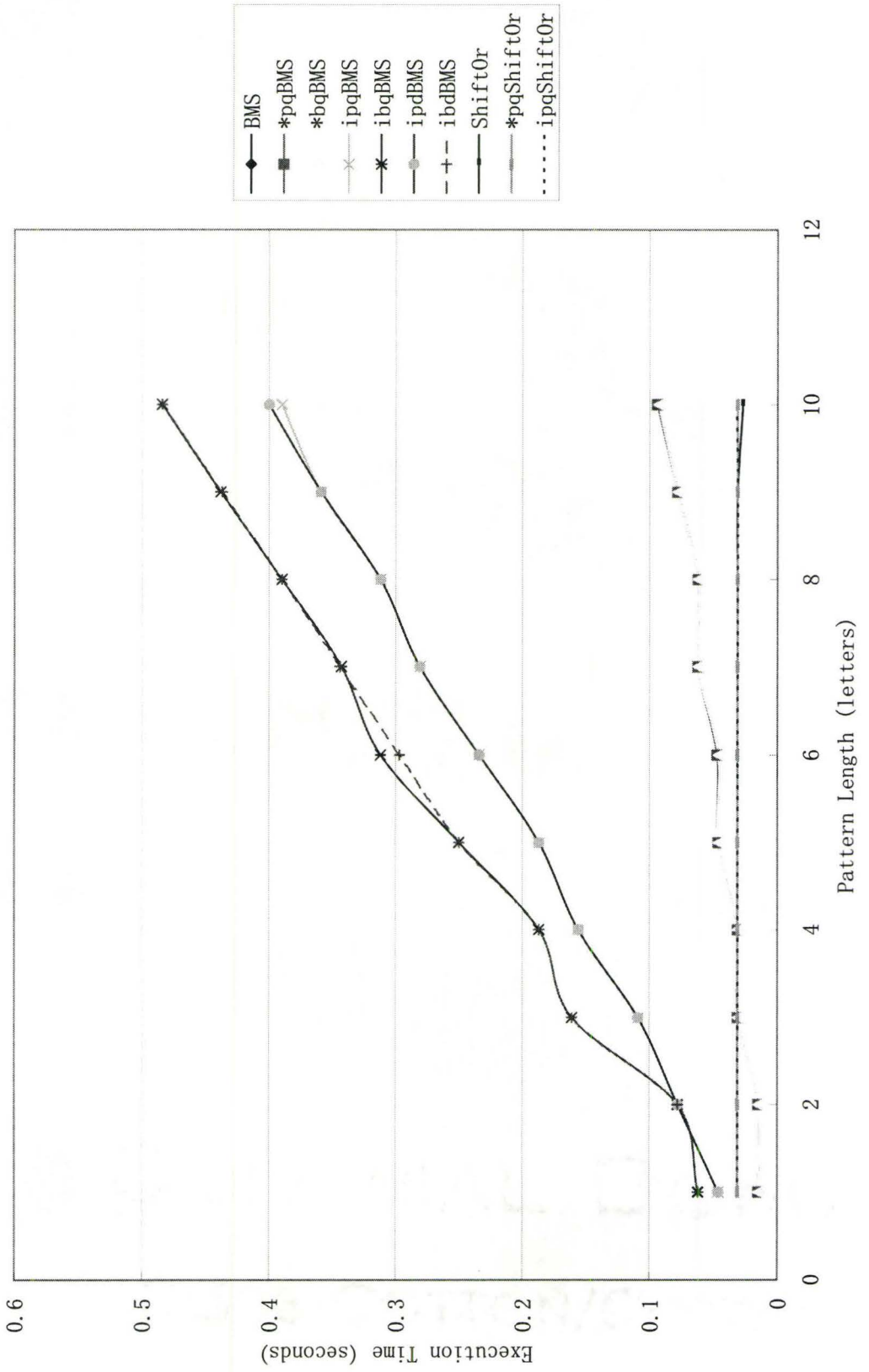

Figure 5.4: Execution time against pattern length on Text = $a^{1000000}$, Pattern = a^m

$b_i \cap b_j$, where the length of each bit array b is related to the alphabet size ($|b| = \lceil k/w \rceil$). Therefore the larger the alphabet, the longer the length of b, and the slower the speed. The remaining algorithms, none of which require a bit array for letter comparison, are independent of alphabet size. For alphabet sizes up to 256, all BMS-derived algorithms are at least twice as fast as their ShiftOr-derived equivalents.

Types Of Texts

We tested 256-letter ASCII English texts as well as 4-letter DNA texts from [HGP] and randomly generated strings. The results are shown in Figure 5.6. The algorithms show similar behavior across these different inputs, with the BMS-derived algorithms generally faster. However the gap between ShiftOr-derived algorithms and BMS-derived algorithms shrinks significantly for small-alphabet files, due to the fact that the average shift of BMS-derived algorithms in a small-alphabet (DNA) string is not as large as that in a large-alphabet (English) string.

Frequency of Indeterminate Letters

To test how frequency of indeterminate letters in input affects the speed of different algorithms, we conducted tests with increasing frequencies of indeterminate letters in pattern, text and alphabet respectively.

First we test all the "*i*" algorithms on patterns with variable frequency of indeterminate letters. We use the high-frequency pattern sets described earlier, modified

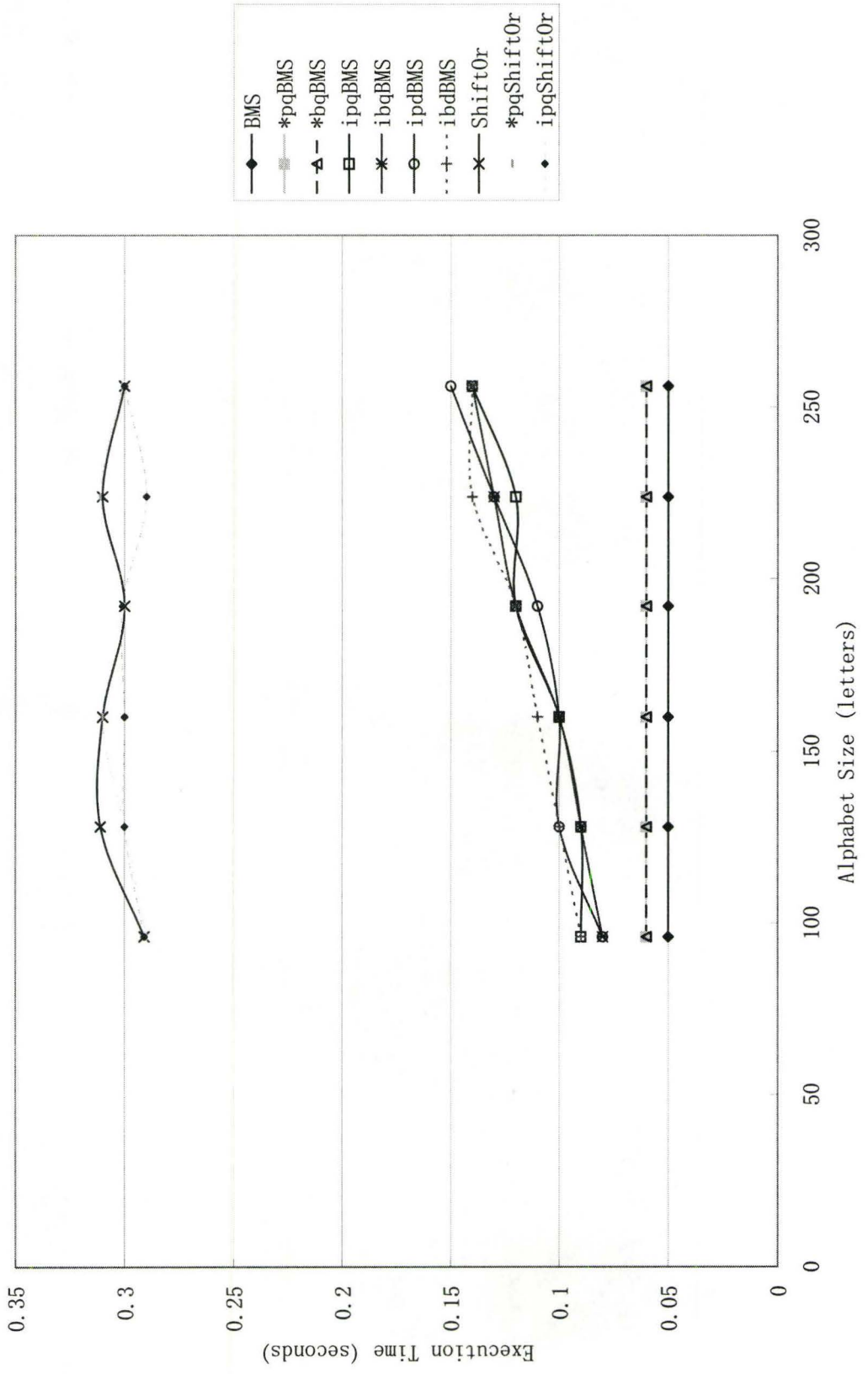

Figure 5.5: Execution time against alphabet size

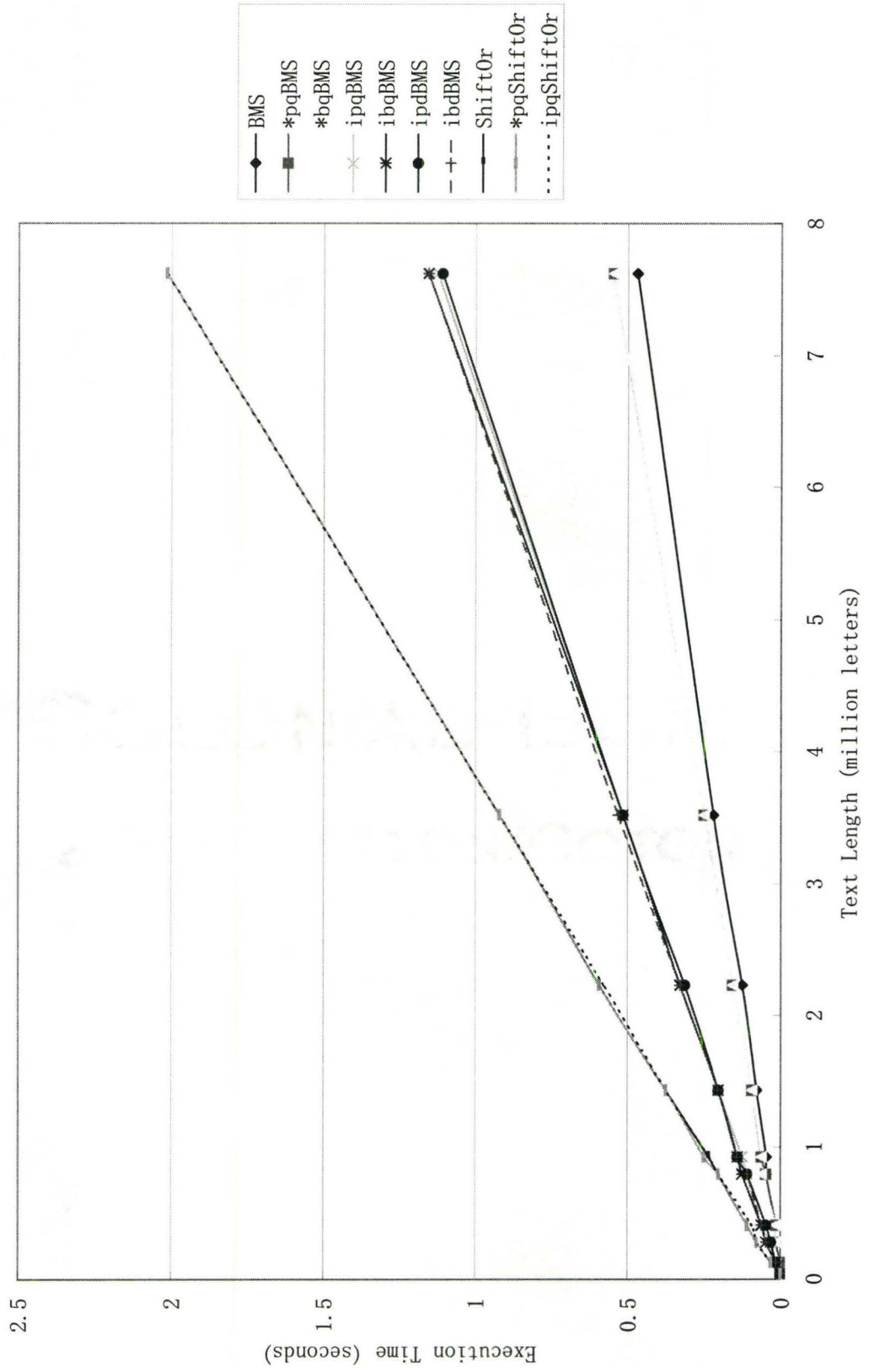

Figure 5.6: Execution time against text length on DNA files

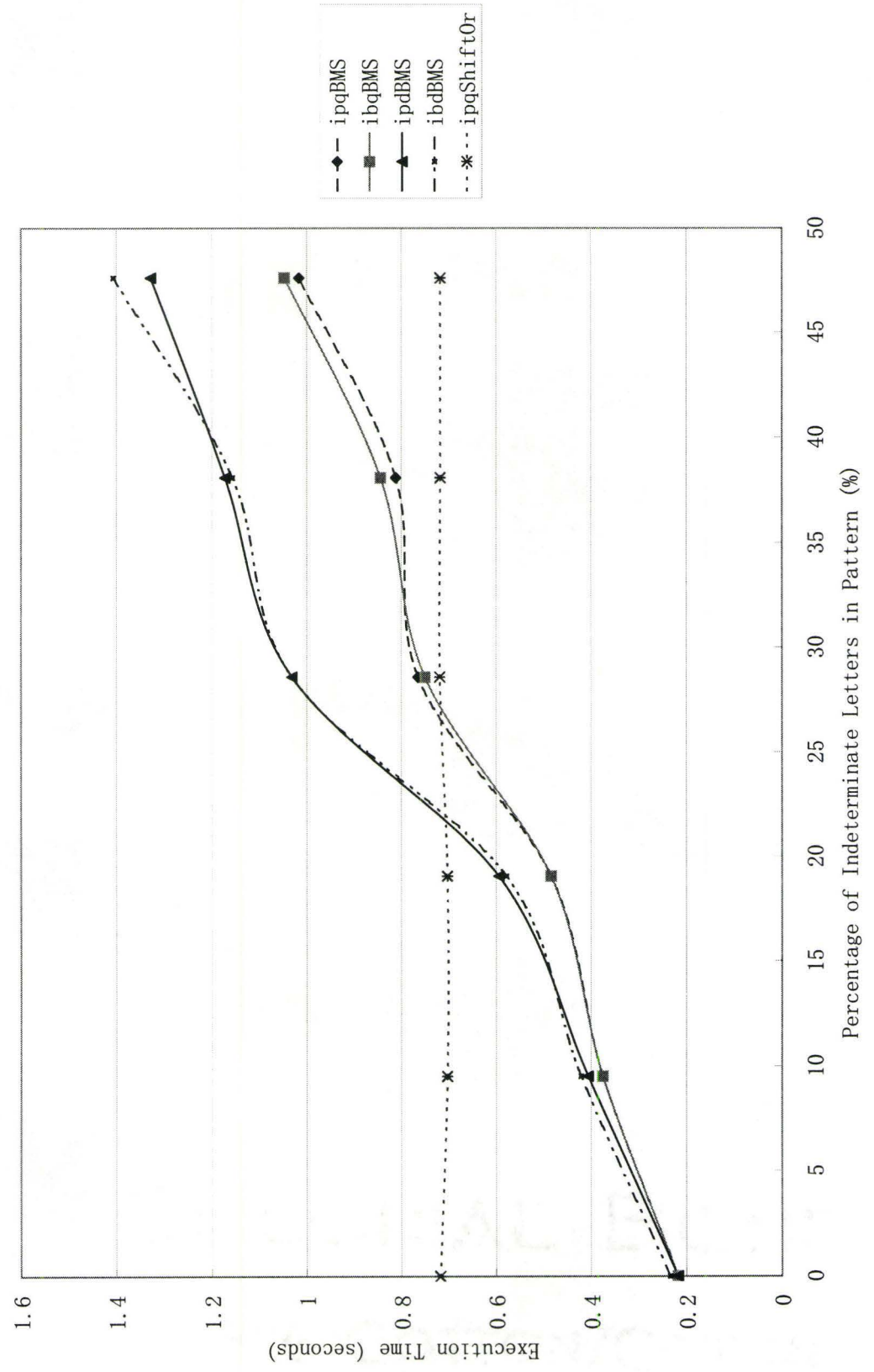

Figure 5.7: Execution time against percentage of indeterminate letters in pattern

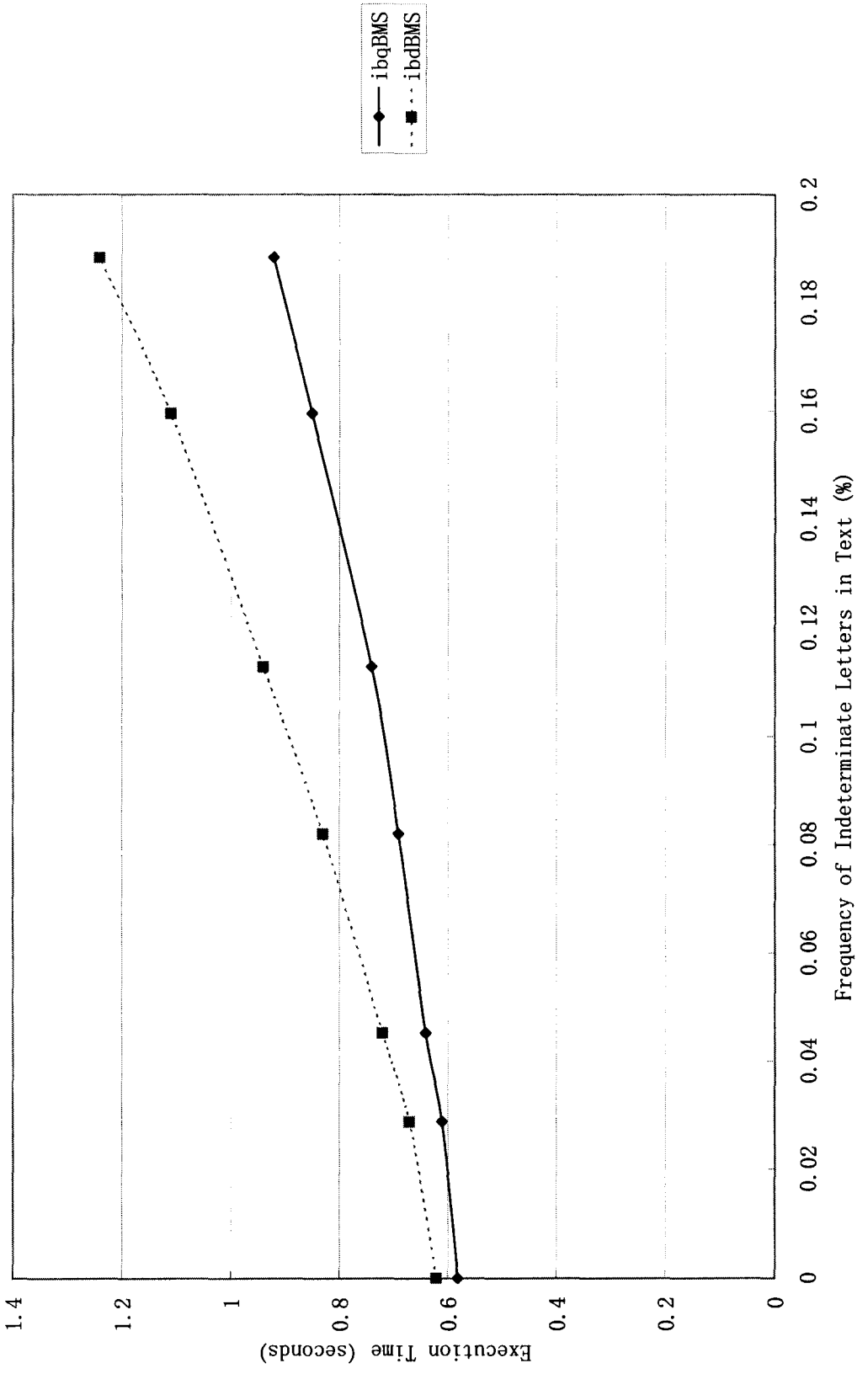

Figure 5.8: Execution time against percentage of indeterminate letters in text

by randomly substituting indeterminate letters for determinate ones in various positions. In this test we define the letter '∧' to be the only kind of indeterminate letter, matching all lower case English letters plus the space symbol. As shown in Figure 5.7, if up to about 25% of the letters in the pattern are indeterminate, *ipq*BMS is faster than *ipq*ShiftOr. As noted earlier, the execution time of *ipq*BMS can also be affected by the size of Σ_j and the locations (left or right) at which the indeterminate letter occurs in the pattern.

Next we tested *ibd*BMS and *ibq*BMS against frequency of indeterminate letters in the text. These are the two algorithms that have the capability of handling indeterminate letters in both text and pattern. Indeterminate letters in this test appear in the text only. The results are shown in Figure 5.8. As expected, execution time increases with increasing frequency, because more computation is required, both in preprocessing and in `StringComparison`.

Finally we tested *ipd*BMS, *ipq*BMS, *ibd*BMS, *ibq*BMS and *ipq*ShiftOr based on various frequencies of indeterminate letters in the alphabet. Because in general increasing the frequency of indeterminate letters in the alphabet will also cause an increase in the indeterminate letters in both text and pattern, which will have compound effects on the running time, we therefore conducted the tests in such a way that newly introduced indeterminate letters neither appear in the text nor in the

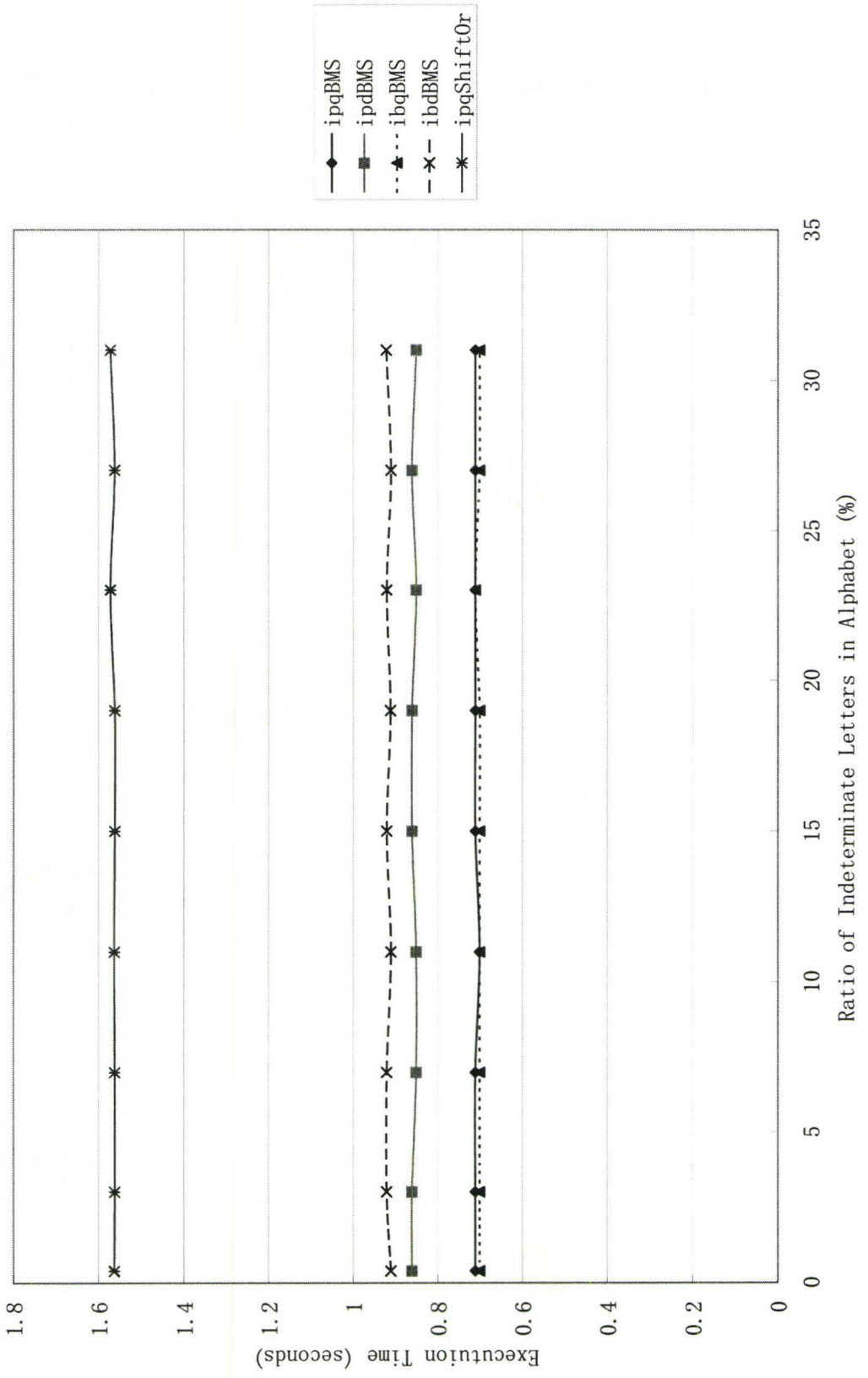

Figure 5.9: Execution time against percentage of indeterminate letters in alphabet

pattern. In other words, the frequency of indeterminate letters in both text and pattern is constant, only the size of the alphabet is changed. The results are shown in Figure 5.9. We see none of the algorithms are affected by increased frequency in the alphabet, an expected result, since the ratio $(K - k)/K$ only affects preprocessing time.

5.3 Conclusions from Experiments

We have tested six indeterminate algorithms based on BMS against equivalent ShiftOr-derived algorithms. On larger alphabets (more than 4 letters), the BMS-derived algorithms seem to offer a significant advantage in processing speed, even though the indeterminate versions of BMS are affected somewhat by very large alphabet size. Also, with realistic frequency levels for indeterminate letters in the pattern (up to 25–30%), the BMS variants have an advantage. Further, the performance of the BMS-derived algorithms, unlike the ShiftOr-derived equivalents, is unaffected by large pattern length.

Chapter 6

Conclusions and Future Work

In this thesis we have discussed efficient algorithms for pattern-matching on indeterminate strings, including the case of locally-constrained matching. We developed new algorithms on the basis of three known determinate pattern-matching algorithms respectively. They are Sunday's adaptation of the Boyer-Moore algorithm, the ShiftOr algorithm and the FJS algorithm, all of which have their own strengths and weaknesses. We conducted comprehensive experiments on selected algorithms, analyzed the results carefully and draw conclusions on which situations these algorithms perform best based on our results.

Also, we would like to further extend our algorithms in two ways: first by extending the functionality such as allowing an indeterminate letter to be empty; second by extending the pattern-matching of indeterminate letters to other determinate pattern-matching algorithms such as FJS, and we wonder whether faster approaches can be found.

At last, we would like to explore alternative approaches to border array calculation and its application to calculation of repetition and repeats.

Bibliography

[AGR] AGREP V3.37, Homepage V1.12, T. Grie. Available from:

http://www.tgries.de/agrep/.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffey D. Ullman. *The Design & Analysis of computer Algorithms*. Addison-Wesley, 1974.

[BM77] Robert S. Boyer and J. S Strother Moore. A fast string searching algorithm. *CACM*, 20(10):762–772, 1977.

[BYG92] R.A. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[Döm68] Bálint Dömölki. A universal computer system based on production rules. *BIT*, 8:262–275, 1968.

[FAS] Fasta format description, National Center for Biotechnology Information. Available from: http://www.ncbi.nlm.nih.gov/BLAST/fasta.shtml.

[FJS05a] Frantisek Franek, Christopher G. Jennings, and W. F. Smyth. A simple fast hybrid pattern-matching algorithm. *submitted for publication*, 2005.

[FJS05b] Frantisek Franek, Christopher G. Jennings, and W. F. Smyth. A simple fast hybrid pattern-matching algorithm (preliminary version). In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching*, LNCS 3537, pages 288–297. Springer-Verlag, 2005.

[GB] GenBank Database, National Center for Biotechnology Information. Available from: http://www.ncbi.nlm.nih.gov/Genbank/index.html.

[Har] Michael Hart. Project gutenberg, project gutenberg literary archive foundation (2004):.

[HGP] The Human Genome Project. Available from:

http://www.doegenomes.org/.

[HS03] Jan Holub and W. F. Smyth. Algorithms on indeterminate strings. In *Proc. 14th Australasian Workshop on Combinatorial Algorithms*, pages 36–45, 2003.

[HSW05a] Jan Holub, W. F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. In *Proc. 16th Australasian Workshop on Combinatorial Algorithms*, pages 415–428, 2005.

[HSW05b] Jan Holub, W. F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *Submitted for publication*, 2005.

[Jen02] Christopher G. Jennings. A linear-time algorithm for fast exact pattern matching in strings. Master's thesis, McMaster University, 2002.

[KMP77] D. E. Knuth, J. H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[Lec] Thierry Lecroq. Exact String Matching Algorithms. Available from: http://www-igm.univ-mlv.fr/ lecroq/.

[MP70] J. H. Morris and V.R. Pratt. A linear pattern-matching algorithm. Technical report 40, University of California, Berkeley, 1970.

[Smy03] Bill Smyth. *Computing Patterns in Strings.* Addison Wesley, 2003.

[Sun90] D.M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 1990.

[UNI] The Unicode Consortium. Available from: http://www.unicode.org/.

[WIK] Bit Array, Version 30744085, The Wikipedia Encyclopedia. Available from: http://en.wikipedia.org/w/index.php?title=Bit_array&oldid=30744085.

[WM92] S. Wu and U. Manber. Fast text searching with errors. *Communications of the ACM*, 35(10):83–91, 1992.