A Pattern Matching Algorithm
Using
Deterministic Finite Automata
with Infixes Checking

Jung-Hua Hsu
A Pattern Matching Algorithm
Using
Deterministic Finite Automata
with
Infixes Checking

Student: Jung-Hua Hsu  Advisor: Dr. S.S. Yu

Department of Computer Science
National Chung Hsing University

Abstract

This thesis presents two string matching algorithms. The first algorithm constructs a string matching finite automaton (called mc-DFA) for multi-class of keystings with infix checking and then applies this mc-DFA to process the matching procedure for a given text in a single pass. This algorithm can output the occurrences of multiple keystings in a text concurrently. An algorithm used to minimize mc-DFA is also given and studied. The second algorithm applies the failure function to detect the occurrence of a prefix square when a string is on-line inputted. Some properties concerning these two algorithms are studied to show the time complexity of these two string matching algorithms.
利用自動機處理含有字中關係的多族關鍵字比對演算法

研究生: 竇a華 指導教授: 喻石生

國立中興大學資訊科學研究所

摘要

這篇論文中提出了2種用於作字串比對的演算法，第一種演算法我們建構了一個可以比對含有字中關係的多族關鍵字的自動機，此自動機在輸入一段文字後，可以判斷出在這段文字中的哪個位置比對到關鍵字。

在此篇論文的第五章，提出了一個演算法，可以判斷出一段文字中是否有重複的字首發生。在此論文中，我們也提出了一些性質，並對我們的演算法的時間複雜度的分析。
Contents

1 Introduction 1

2 The K.M.P algorithm and the Aho-Corasick algorithm 12
   2.1 The K.M.P algorithm 12
   2.2 The Aho-Corasick algorithm 16
   2.3 The problem of the Aho-Corasick algorithm 21

3 Basic notations and properties of the strings 23
   3.1 Relations between strings 23
   3.2 Definition of mc-DFA 25

4 The algorithm of mc-DFA 28
   4.1 The Multi-Final-State-Class DFA Constructing Algorithm 29
   4.2 Minimization 36
   4.3 Minimized Concurrent Matching DFA Generating Algorithm 40

5 A prefix square check algorithm 42
   5.1 Prefix Square Detection Algorithm 42
   5.2 Properties of Algorithm 5.1.1 44
   5.3 The Time Complexity of Algorithm 5.1.1 46

6 Concluding Remarks 52
List of Figures

1.1 before shift. .................................................. 8
1.2 after shift. ................................................... 8
1.3 Good suffix rule. .............................................. 9

2.1 Pattern string. ................................................ 15
2.2 The next table. ............................................... 15
2.3 The goto function of the pattern set \{ACT, TG\} ........... 20
2.4 The output function .......................................... 20
2.5 The pattern matching machine with a failure link. ......... 20
2.6 The goto function. ............................................. 22

3.1 ................................................................. 27
3.2 ................................................................. 27

4.1 ................................................................. 31
4.2 DFA ............................................................ 32

5.1 \(n_j\) table ..................................................... 44
5.2 \(n_j\) and \(4_j\) table ........................................... 46
Chapter 1

Introduction

This research concerns pattern matching problems. An algorithm solves the multiple keystrings (patterns) matching problems with infix checking is proposed in this study. Pattern matching is used to check the similarities of strings. To solve the pattern matching problem it is necessary to find an algorithm which can locate the similarities of strings, that is, the longest common segments or the smallest difference of strings. In many fields, such as: computer science, computer engineering, bio-science, lexical analysis, database query, search engines in WWW, and so on, pattern matching processings must be applied frequently. A good pattern matching algorithm can help the user to find appropriate results quickly. There are many ways to define the similarity of strings. Roughly speaking, there are two main criteria of the similarities of strings: the longest common consecutive segments and the longest common inconsective segments. The related pattern matching problems are exact pat-
tern matching problems and inexact pattern matching problems, respectively.

One way to find the longest common inconsecutive segments and the smallest
difference of strings is to insert spaces between strings so that the obtained
strings have the same lengths and have the longest common inconsecutive seg-
ments in the corresponding positions with the least insertions. One may assign
a positive value to each matched position and a negative value to each mis-
matched position. Then the largest sum is defined as the degree of similarity
of the strings. This is one of the inexact pattern matching methods. Inexact
pattern matching problems are not involved in this thesis. In this research, we
focus on exact pattern matching problems.

For exact pattern matching problems, there are also two main topics: finding
the longest consecutive common segments of strings and checking whether
some given keystings are substrings of a text. The former is studied by apply-
ing the failure function to solve the prefix square checking problem in Chapter
5. This research mainly focuses on the latter. That is, a pattern matching
procedure is an algorithm which compares a set of keystings with a text to
check whether any of the given keystings is a substring of the text, and then
outputs locations when a keysting occurs in the text or a mismatched signal
when no keysting occurs in the text.

Exact pattern matching algorithms have very extensive applications. For
example, a compilation consists of analyzing the source program (source code),
called linear analysis (or lexical analysis, scanning), and transferring it to a
target program (or machine code). During the linear analysis step, exact pat-
tern matching algorithms are applied to identify reserved words (keystrings),
variables (or operands), and operators from the source program. After this is
performed, the compiler makes up the source program from left-to-right and
transfers it into tokens.

Exact pattern matching algorithms are also applied in the field of bioin-
formation. DNA strings are composed of 4 characters: \{A, G, C, T\}. The
DNA string of a human being is a string of about three billion of such char-
acters. Huge databases for DNA strings are established for the study of life.
The *Genbank* (US), *Embl* (Europe), *Human* (GDB), *ESTs* (expressed string
tags), etc ... are all such large databases. Huge sizes and exponentially grow-
ing amount of data in biologic databases, particularly in genome and protein
databases, must be analyzed and processed. To study what characteristics a
given DNA string has, one may choose some shorter DNA strings with some
special characteristics to compare with the given DNA string. Hence, pattern
matching becomes a basic operation in computational biology. Since usually
DNA databases are much larger and grow faster, to find an algorithm that can
efficiently match the query string from a DNA database is very imperative.
There are many papers which have presented algorithms to deal with pattern matching problems which include numerical methods and comparison-based methods. The Karp-Rabin algorithm [5] uses the "seminumerical" method to determine whether a matching occurs or not. It transfers the pattern string and the text string to decimal numbers $H(P)$ and $H(T)$, respectively. It is defined that a match occurs if and only if $H(P) = H(T)$. The K.R algorithm compares these two numbers $H(P)$ and $H(T)$ rather than compare the characters between text and pattern directly. While the length of the pattern is large, to compute $H(P)$ and $H(T)$ becomes inefficient. The Karp-Rabin algorithm applies the modulo operation to reduce the number of bits that $H(P)$ and $H(T)$ might use. To compute $H(T)$ and $H(P)$ with fewer number of bits will be more efficient. But the so called false match problem arises when the modulo operation is used. For example, $(0011)_2 \mod 7 = 3 = (1010)_2 \mod 7$, but $(0011)$ doesn’t match $(1010)$. The case that $H(P) = H(T)$ whereas $P$ does not match $T$ is called a false match between $P$ and $T$. In these findings [5], it has been proved that the probability of a false match is insignificant. However, the false match can not be avoided after all by using the modulo operation.

Now we focus our attention on comparison-based methods. A comparison-based method compares characters of pattern and text realistically. These meth-
ods of course can prevent the occurrence of false matches.

For convenience, we give two notations to denote the keystring (or called the pattern) and the text. Let $\Sigma$ be the set of letters used. $T[i]$ denotes the $i$th character of the string of a text, and $T[1, \ldots, n]$ (or $T$) denotes the whole text. We use $P[j]$ to denote the $j$th character of the string of a pattern, and $P[1, \ldots, m]$ (or $P$) to denote the whole pattern. A match occurs at the position $i$ of the text if and only if $T[i] = P[1], T[i+1] = P[2], \ldots, T[i+m-1] = P[m]$. For example, if $T = abcd$ and $P = cd$, then the position of the matched occurrence is 3, that is, $T[3] = P[1] = c, T[4] = P[2] = d$. Meanwhile, the above mentioned example indicates that the main issue of pattern matching is to either locate the position of the text if a pattern matching occurs or declare a mismatch. We now introduce a simple comparison-based algorithm, known as the naive algorithm (see [3]), to solve the exact pattern matching problem.

Naive algorithm is the most intuitional way to find the position of an occurrence of a pattern in a text. For a given text with length $n$ and a pattern with length $m$, the naive algorithm first compares $T[1]$ and $P[1]$. If $T[1] = P[1]$, then compares $T[2]$ and $P[2]$, and so forth. If $T[i] = P[i]$ for each $1 \leq i \leq m$, then outputs a match signal; otherwise, the naive algorithm shifts the pattern one position to the right. That is, the naive algorithm starts to compare $T[2]$ and $P[1]$. If $T[2] = P[1]$, then the algorithm compares $T[3]$
and $P[2]$, and so on. If $T[j + 1] \neq P[j]$ for some $1 \leq j \leq m$, then the algorithm shifts the pattern one position to the right. This motion is repeated until the text ends or a full match occurs. If the text ends and the pattern is not fully matched, then output a mismatch signal. When the pattern is matched, that is, $T[j + k] = P[j]$ for each $j$ with $1 \leq j \leq m$ and for some $k$ with $m \leq m + k \leq n$, the naive algorithm outputs a message to indicate the occurrence of a match at the position $k + 1$ of the text. A run is an executable cycle which compares $P[1, \ldots, m]$ and $T[i, \ldots, m + i]$ for $1 \leq i \leq n - m$.

The running time (time complexity) of the naive algorithm is the sum of comparisons in each run.

The naive algorithm is simple but its time complexity is great. The worst case of time complexity of the naive algorithm is $O(nm)$. For example, make the DNA string $T = A^nC$ be the text which is longer than the DNA string $P = A^mC$ (i.e., $m < n$). Since $P[1] = P[2] = \cdots = P[m] = A$, $P[m + 1] = C$ and the first $n$ characters of the text are all $A$ where $n > m$, a mismatch occurs at position $m + 1$ of the text in the first run. The number of comparisons in the first run is $m + 1$. When a mismatch occurs, the pattern is shifted one position to the right and the following run is processed. This is then clear that a mismatch constantly arises at the position $i + m$ of the text during the $i$th run $(1 \leq i \leq n - m + 1)$. In this case, there are $n - m + 1$ runs and $m + 1$ comparisons
in each run, so that the total of comparisons is \((m + 1) \times (n - m + 1)\). If the number \(n\) is 1000 and the number \(m\) is 10 (i.e., \(n \gg m\)), the total amount of comparisons is almost equal to \(n \times m = 1000 \times 10 = 10000\). Some of the comparisons made by the naive algorithm are not necessary and can be avoided. In the following chapter, we will introduce some algorithms which can improve the naive algorithm to reduce the unnecessary comparisons and speed up the pattern matching process.

The Aho-Corasick algorithm [1], the Knuth-Morris-Pratt algorithm [6] (K.M.P algorithm), and the Boyer-Moore algorithm [2] (B.M algorithm) are some other comparison-based algorithms. We will introduce the Aho-Corasick algorithm and the K.M.P algorithm in Chapter 2. The B.M algorithm accomplishes the pattern matching procedure under three new rules. According to the first rule, the B.M algorithm scans characters of pattern and text from right to left. This rule is different from the naive algorithm. For example, if the pattern is \(P[1\ldots5] = a b a b c\) and the text is \(T[1\ldots10] = a b a d c a b a b c\).

The B.M pattern matching algorithm compares \(P[5]\) and \(T[5]\) first. As \(P[5] = T[5] = c\), the algorithm compares \(P[4]\) and \(T[4]\), and so forth (i.e., the B.M algorithm compares characters from the rightmost of the pattern to the leftmost). The second rule of the B.M algorithm is called the bad character rule. For example, when \(P[4]\) and \(T[4]\) are compared, a mismatch signal appears,
since \( P[4] = b \) and \( T[4] = d \). In this case, the naive algorithm should shift the pattern exactly one position to the right and then repeat scanning the characters of the pattern and the text from right to left. This causes the redundant comparison and wastes time. Consider \( T[4] = d \), the character "d" does not occur in the pattern string \( P \). So that if one shifts merely one position and repeats pattern matching algorithm to compare the pattern and the text from right to left, a match would not occur. Since character "d" never matched with any character in \( P \). So that, the bad character rule of B.M algorithm shifts the pattern \( P \) completely past the mismatched point in \( T \). See Figure 1.1 and Figure 1.2. Figure 1.1 denotes the alignment of the text and the pattern

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
T: & a & b & a & d & c & a & b & a & b & c \\
P: & a & b & a & b & c \\
\end{array}
\]

Figure 1.1: before shift.

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
T: & a & b & a & d & c & a & b & a & b & c \\
P: & & a & b & a & b & c \\
\end{array}
\]

Figure 1.2: after shift.

before shift and Figure 1.2 denotes the alignment after shift when we find a mismatch at the position \( T[4] \). This property is known as the bad character rule. When a character in the text \( T \) does not occur in the pattern \( P \), by the
bad character rule, we may save many needless comparisons. The third rule of the B.M algorithm is called the good suffix rule. See Figure 1.3. Figure 1.3 denotes when the substring $s$ of $T$ matches the suffix of the pattern, but a mismatch occurs in the next character to the left (that is, $x \neq z$). If there is a substring $s'$ not a suffix of the pattern $P$ matches the suffix of $P$ and the left character is different then shift the pattern to right such that the substring $s'$ of pattern is aligned to the substring $s$ of the text. If there no such substring $s'$ exists, then shift the left-end of $P$ past to the left-end $s$ in $T$. If no such shift is possible (i.e., the prefix of $P$ does not match the proper suffix of $s$), then shift $P$ by $m$ places ($m$ be the length of the pattern), that is to say, shift $P$ past $s$ in $T$. Because, any substring of $P$ before $y$ never matches the substring $s$ of text $T$. Through rule 2 and 3, unnecessary comparisons will be avoided and the time complexity to determine which position of text matches the pattern is linear relative to the length of the text. The bad character rule
and the good suffix rule is called the preprocessing of the pattern. The good suffix rule is similar to the next table of the K.M.P algorithm, and the next table will be specified in Chapter 2.

Some definitions of deterministic finite automaton and some relation between strings are stated in Chapter 3. In Chapter 4, this thesis brings up a new algorithm which can handle multiple keystrokes concurrently by constructing an multi-class-deterministic-finite-automaton (shortly, mc-DFA). The mc-DFA constructed in Chapter 4 not only can recognize the multiple keystrokes but also can identify the multiple classes of keystrokes such as the following:

\[ F_1 = \{CAGC, AGT, CGT, CCTC, TGCA\}, \]
\[ F_2 = \{GGCG, GGCC, TAC, CCG\}, \]
\[ F_3 = \{GCGC, CGGC, CCGC\}. \]

(The recognition strings are referred in the Restriction Enzyme Database [10].) 

\( F_1, F_2, \) and \( F_3 \) are three classes of keystrokes, and each class contains one or more keystrokes. The keystrokes sketched above are sometimes called the patterns, and the text is sometimes called subject string. It’s awkward to implement the B.M algorithm and K.M.P algorithm to handle the multiple classes or multiple keystrokes synchronously. But sometimes, to compare multiple keystrokes concurrently and efficiently is required. We mentioned such problems in detail in Chapter 4 by using a finite state machine to solve such
a problem.

In Chapter 5, a pattern matching algorithm is proposed which can check the prefix square in an inputted string. The algorithm in Chapter 5 applies to the next function in K.M.P algorithm and can output the prefix square string if the inputted string has prefix squares. Finally, the conclusion is presented in Chapter 6.
Chapter 2

The K.M.P algorithm and the Aho-Corasick algorithm

In this chapter, we introduce two algorithms which are most well-known in the field of pattern matching. The two algorithms are the K.M.P algorithm [6] and the Aho-Corasick algorithm [1], respectively. In Chapter 4, we introduce an algorithm which improves the Aho-Corasick algorithm and the pattern matching process, the mc-DFA does not require a next-table in [6] or any failure function in [1].

2.1 The K.M.P algorithm

In this section, we roughly introduce the Knuth-Morris-Pratt algorithm. The principle of the K.M.P algorithm is to compute the next table first (i.e. preprocessing of pattern). And then, when a mismatch occurs, we look up
the next table to shift the pattern appropriately until the pattern matches the
text exactly (fully) or the text ends.

Algorithm 2.1.1

Pattern matching algorithm:

\[
j := k := 1;
\]
\[
\text{while } j \leq m \text{ and } k \leq n \text{ do}
\]
\[
\quad \text{begin}
\]
\[
\quad \quad \text{while } j > 0 \text{ and } text[k] \neq pattern[j]
\]
\[
\quad \quad \quad \text{do } j := \text{next}[j];
\]
\[
\quad \quad \quad k := k + 1; j := j + 1;
\]
\[
\quad \text{end;}
\]

Algorithm 2.1.1: The Knuth-Morris-Pratt pattern matching algorithm.

Algorithm 2.1.1 shows the process of the pattern matching algorithm, \( k \)
denotes the \( k \)th character of the text, and \( j \) denotes the \( j \)th character of the
pattern. First, we need two arrays to store the text string and the pattern
string and then we proceed with comparing the characters of pattern and text
from the left of the array to right. It is easier if we imagine placing the pattern
array under the text array and move it to the right in a certain way. Compare
the characters of the text and the pattern, while \( text[k] \neq pattern[j] \), means
that a mismatch occurs, then we seek the next table, and shift the pointer
of the pattern \( \text{next}[j] \) positions (i.e., shift the pattern \( j - \text{next}[j] \) places, \( j \)
be the position where a mismatch occurs). And then, continue the procedure
until the condition of the first loop of Algorithm 2.1.1 no longer holds. \( k > n \)
Algorithm 2.1.2
Compute the next table:

\[
\begin{align*}
j & := 1; \ t := 0; \ next[1] := 0; \ f[1] = 0 \\
\text{while} \ j < m \ & \text{do} \\
& \begin{align*}
\text{begin comment } t &= f[j]; \\
& \text{while } t > 0 \text{ and } \text{pattern}[j] \neq \text{pattern}[t] \\
& \quad \text{do } t := next[t]; \\
& \quad t := t + 1; \ j := j + 1; \\
& \quad \text{if } \text{pattern}[j] = \text{pattern}[t] \\
& \quad \quad \text{then } next[j] := next[t] \\
& \quad \quad \text{else } next[j] := t; \\
& \text{end.}
\end{align*}
\end{align*}
\]

Algorithm 2.1.2: Compute the next table.

Algorithm 2.1.2 calculates the next table. The next table plays the main role of the K.M.P algorithm and it is of great interest. It leads us to know how far we may shift the pointer of the pattern when a mismatch occurs. The principle of the next table is to preprocess the pattern string. Algorithm 2.1.2 computes the longest length of the prefix of the pattern string such that the prefix of the string matches the suffix of the string. Assume the pattern string
is $ACDAG$ shown in Figure 2.1, and the next table is shown in Figure 2.2.

The number $f(j)$ denotes the largest $i$ less than $j$ such that $P[1] \ldots P[i-1] =

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
P: & A & C & D & A & C & D & A & G \\
\end{array}
\]


\[
next[j] = \begin{cases} 
  f[j], & \text{if } pattern[j] \neq pattern[f[j]], \\
  next[f[j]], & \text{otherwise.} 
\end{cases}
\]
The advantage of the next table is that it avoids unnecessary comparisons and decreases the time complexity. The time complexity of the pattern matching algorithm using next table is $O(n)$, $n$ being the length of the text. But, it is difficult to implement if we want to compare two or more patterns with the text concurrently.

The next section, we mention another important algorithm which can operate the multiple patterns or keystrings and compare with the text in a single pass concurrently.

### 2.2 The Aho-Corasick algorithm

The K.M.P algorithm can solve the pattern matching problem and outputs whether a text matches a pattern or not. But at times, there may not only be one pattern to be compared. In many situations, we hope to deal with two or more keystings concurrently and efficiently. In this section, we introduce the Aho-Corasick algorithm which constructs a matching machine (goto function) and compare multiple keystings with the text in a single pass.

#### Algorithm 2.2.1

**Pattern matching machine:**

**Input:** A text string $x = a_1a_2\cdots a_n$ where each $a_i$ is an input symbol and a pattern matching machine $M$ with goto function $g$, failure function $f$, and output function output, as described above.

**Output:** Location at which keystings occur in $x$.

**Method:**
Algorithm 2.2.1: Pattern matching machine.

Algorithm 2.2.1 is the pattern matching algorithm using the goto function constructed in Algorithm 2.2.2 and the failure function constructed in Algorithm 2.2.3. First, according to the set of patterns, we decide on a set of alphabets \( \Sigma \). And then, we construct the goto function, sometimes known as a trie, too.

**Algorithm 2.2.2**

Construction of the goto function:

*Input:* Set of keystrings \( K = \{y_1, y_2, \ldots, y_k\} \)

*Output:* Goto function \( g \) and a partially computed output function \( \text{output} \).

*Method:* We assume \( \text{output}(s) \) is empty when state \( s \) is first created, and \( g(s, a) = \text{fail} \) if \( a \) is undefined or if \( g(s, a) \) has not yet been defined. The procedure \( \text{enter}(y) \) inserts into the goto graph a path that spells out \( y \).

*begin*

\[ \text{newstate} \leftarrow 0 \]

*for* \( i \leftarrow 1 \) *until* \( n \) *do*

*begin*

\[ \text{while} \ g(\text{state}, a_i) = \text{fail} \text{ do } \text{state} \leftarrow f(\text{state}) \]

\[ \text{state} \leftarrow g(\text{state}, a_i) \]

if \( \text{output(\text{state})} \neq \text{empty} \) then

*begin*

print \( i \)

print \( \text{output(\text{state})} \)

*end*

*end*

*end*
\begin{algorithm}
\begin{algorithmic}
\State $state \leftarrow 0; j \leftarrow 1$
\While{$g(state, a_j) \neq \text{fail}$}
\Begin
\State $state \leftarrow g(state, a_j)$
\State $j \leftarrow j + 1$
\End
\For{$p \leftarrow j$ \Until{$m$}}
\Begin
\State $\text{newstate} \leftarrow \text{newstate} + 1$
\State $g(state, a_p) \leftarrow \text{newstate}$
\State $state \leftarrow \text{newstate}$
\End
\State $\text{output}(state) \leftarrow \{a_1a_2 \cdots a_m\}$
\End
\end{algorithmic}
\end{algorithm}

\textit{Algorithm 2.2.2: Constructing the goto function}

The manner of constructing the \textit{goto} function is similar to the finite state automaton defined in Chapter 3. For example, assume the set of patterns is \{\textit{ACT, TG}\}. Thus, we have the alphabet $\Sigma = \{A, C, T, G\}$. The initial state is $S_0$, and from the pattern \textit{ACT}, we have the \textit{goto} function (transition rule): $(S_0, A) = S_1$, $(S_1, C) = S_2$, $(S_2, T) = S_3$, and then add $S_3 = \{\textit{ACT}\}$ to the \textit{output} function. From the pattern \textit{TG}, we have the \textit{goto} function (transition rule): $(S_0, T) = S_4$, $(S_4, G) = S_5$ and then add $S_5$ to the \textit{output} function.

After the \textit{goto} function is constructed, we have the set of states which is $Q = \{S_0, S_1, S_2, S_3, S_4, S_5\}$, and \textit{output} function is $\{S_3 : \textit{ACT}, S_5 : \textit{TG}\}$. From the previous example, we can see that the \textit{goto} function is constructed for each pattern step by step.
Algorithm 2.2.3
Construction of the failure function

Input: Goto function \( g \) and output function \( \text{output} \) from Algorithm 2.
Output: Failure function \( f \) and output function \( \text{output} \).
Method:

begin
    queue ← empty
    for each \( a \) such that \( g(0, a) = s \neq 0 \) do
        begin
            queue ← queue ∪ \{s\}
            \( f(s) \) ← 0
        end
    while queue ≠ empty do
        begin
            let \( r \) be the next state in queue
            queue ← queue − \{r\}
            for each \( a \) such that \( g(r, a) = s \neq \text{fail} \) do
                begin
                    queue ← queue ∪ \{s\}
                    state ← \( f(r) \)
                    while \( g(state, a) = \text{fail} \) do state ← \( f(state) \)
                    \( f(s) \) ← \( g(state, a) \)
                    output(s) ← output(s) ∪ output(f(s))
                end
            end
        end
    end
end

Algorithm 2.2.3: Constructing the failure function

So that we can sketch the goto function as shown in Figure 2.3. According to the goto function shown in Figure 2.3, Algorithm 2.2.3 constructs the failure link (failure function) of each node of the goto function.

The principle of finding a failure function of a node \( s \) is that finding node \( r \)
such that the path from root node to \( r \) is the longest path which matches the path from a node \( s' \) to \( s \), for some \( s' \) on the path from root to \( s \). See Figure 2.5, the failure link of the node \( S_3 \) points to \( S_4 \). Because the path from \( S_0 \) to
$S_4$ is the longest path that matches the suffix of the path from $S_0$ to $S_3$.

Now assume the text is $ACTG$, one can use Algorithm 2.2.1 to process the text. $g(S_0, A) = S_1$, $g(S_1, C) = S_2$, $g(S_2, T) = S_3$, and $S_3$ is in the output function. So that, Algorithm 2.2.1 outputs the position of the text and the matching keystore {ACT}. When $T[4] = T$ is inputted, the goto function of $S_3$ is undefined. According to the failure function, the node $S_3$ translates into $S_4$ and $g(S_4, G) = S_5$. $S_5$ is in the output function, so that a match signals output, too.

### 2.3 The problem of the Aho-Corasick algorithm

Since the Aho-Corasick algorithm can compare multiple keystrogs with text concurrently, sometimes it cannot get an exact result, and the backtracking of the failure link cannot always be avoided. For example, the set of multiple patterns is \{ACT, TG, CT\}. The goto function of the pattern set is shown in Figure 2.6.

Suppose the text now is $ACTC$. According to Algorithm 2.2.1, $g(S_0, A) = S_1$, $g(S_1, C) = S_2$, $g(S_2, T) = S_3$, and $S_3$ is in the output function then Algorithm 2.2.1 outputs the keystring ACT. When the letter $C$ is inputted, $g(S_3, C)$ undefined. According to the failure function, the state $S_3$ translates
Figure 2.6: The goto function.

into $S_4$. And $g(S_4, C)$ is also undefined, the state translates into $S_0$. $g(S_0, C) = S_6$ and $S_6$ is in the output function. The Algorithm 2.2.1 outputs the keystring $C$. In this example, the failure function is executed twice when the state is $S_3$ and the input letter is $C$. This is unsatisfactory since the failure function may execute over and over until locating an exact state. And when the state is $S_1$, the input letter is $C$. The state translates into $S_2$ and $S_2$ is not in the output function. But actually, $T[2] = C$ matches the keystring $C$.

In Chapter 4, we propose an algorithm which constructs an mc-DFA. The mc-DFA does not require a next table or failure link (i.e., no backtracking). And also, the mc-DFA can process the text string to compare with multiple keystring concurrently in a single pass.
Chapter 3

Basic notations and properties of the strings

This chapter explains the relations and properties between strings which could influence the matching result and a definition of the mc-DFA is stated.

3.1 Relations between strings

Definition 3.1.1 For any words $z, u, v, w \in \Sigma^*$, if $z = uvw$ then $u$ (resp. $v, w$) is a prefix (resp. infix, suffix) of $z$, denoted by $u \leq_p z$ (resp. $v \leq_i z$, $w \leq_s z$). If $u \neq z$ (resp. $v \neq z$ or $w \neq z$) then $u$ (resp. $v, w$) is a proper subword of $z$, denoted by $u <_p z$ (resp. $v <_i z$, $w <_s z$).

Definition 3.1.2 For any $u, v \in \Sigma^+$, $u$ is a non-prefix infix (shortly, np-infix) of $v$ if $v = xuy$ for some $x \in \Sigma^+$ and $y \in \Sigma^*$. 
Example 3.1.1 Suppose the set of strings is \( \{K_1, K_2, K_3\} \) over the alphabet \( \Sigma = \{A, C, T, G\} \), where \( K_1 = AC \), \( K_2 = ACT \), \( K_3 = ACACT \). According to Definition 3.1.1 and Definition 3.1.2, \( K_1 \) is a proper prefix of \( K_2 \) and \( K_3 \), denoted by \( K_1 <_p K_2 \) and \( K_1 <_p K_3 \), respectively. \( K_2 \) is a proper suffix of \( K_3 \) and \( K_1 \) is a proper infix of \( K_3 \), denoted by \( K_2 <_s K_3 \) and \( K_1 <_i K_3 \), respectively. Also see that, \( K_2 \) is a non-prefix infix (shortly, np-infix) of \( K_3 \).

Definition 3.1.3 We say that there is a prefix square of a string if for a string \( u \), there is a subword (substring) \( v \) such that \( u = vw^2w \), for some \( w \in \Sigma^* \).

Example 3.1.2 From Example 3.1.1, the string \( K_3 \) has a prefix square since \( K_3 = ACACT = (AC)^2 T \).

Definition 3.1.4 A word \( v \) is called a bifix of a word \( u \) if \( v \) is both a prefix of \( u \) and a suffix of \( u \).

Example 3.1.3 Suppose \( u = ACGTACG \), \( v = ACG \) are two strings over the alphabet \( \{A, C, T, G\} \), one can observe that \( v <_p u \) and \( v <_s u \), so that \( v \) is a bifix of \( u \).

In Chapter 5, an algorithm is proposed to detect the occurrence of a prefix square of a string. The concept of the algorithm followed from the next-function of the K.M.P algorithm. It is an on-line algorithm which can check
whether an input string has a prefix repetition and the time complexity is not only linear but also sublinear.

3.2 Definition of mc-DFA

Definition 3.2.1 A deterministic finite automaton (shortly, DFA) $M$ is a quintuple $(Q, \Sigma, \delta, s, F)$, where $Q$ is the finite set of states, $\Sigma$ is the input alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function, $s \in Q$ is the start state, and $F \subseteq Q$ is the set of final states.

Definition 3.2.2 Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA and $\epsilon$ the empty string. For $m \geq 1$, the extended transition function $\delta^* : Q \times \Sigma^* \rightarrow Q$ is defined recursively as: $\delta^*(s_i, \epsilon) = s_i$ and $\delta^*(s_i, wa) = \delta(\delta^*(s_i, w), a)$ for all $s_i \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$. In the sequel, $\delta$ is used as $\delta^*$ without special mention when it is not confusing.

Definition 3.2.3 A DFA $M = (Q, \Sigma, \delta, s, F)$ is a complete DFA if for any $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ is defined.

Definition 3.2.4 A multi-final-state-class DFA (shortly, mc-DFA) $M_{mc}$ is a quintuple

$$M_{mc} = (Q, \Sigma, \delta, s, \bigcup_{i=1}^{n} F_i),$$

where
$Q$ is an alphabet of state symbols;

$\Sigma$ is an alphabet of input symbols;

$\delta : Q \times \Sigma \rightarrow Q$ is a transition function ($\delta$ will also be used as $\delta^*$ in the sequel);

$s \in Q$ is the start state;

$\bigcup_{i=1}^{n} F_i \subseteq Q$ is the disjoint union of $n$ classes of final states.

**Example 3.2.1** For the alphabet $\Sigma = \{A, C, G, T\}$, Figure 3.1 shows a DFA which can recognize the string with prefix $ACG$. And Figure 3.2 reveals the complete DFA of Figure 3.1.

For properties of deterministic finite automata (or finite state machine) one can see in [9]. Next chapter, an algorithm is proposed to construct an mc-DFA which can compare multiple classes of keystrings with text concurrently.
Chapter 4

The algorithm of mc-DFA

In this chapter, we propose an algorithm which can construct a deterministic finite state machine (automaton) and this deterministic finite automaton can recognize multiple classes of keystrings. The algorithm presented in this chapter improves the A.C algorithm and the K.M.P algorithm since it never used the failure function or the next table (i.e. no backtracking). If we want to match multiple classes of keystrings by Knuth algorithm, we must deal with the next table for each keystring. And process the text to compare with each keystring of the classes of the keystrings. It is hard to compare multiple keystrings concurrently. In Aho-Corasick algorithm, although it can compare multiple keystrings concurrently, the infix problem and the backtracking of the failure link can not be avoided. And the state of the goto function can not be minimized. The number of states would almost be equal to $\sum_{i=1}^{n} \log(K_i)$ (lg($K_i$) be the length of each keystring $K_i$). To adapt such problems, the mc-DFA has
been constructed. Every transition rule would be defined exactly after executing the complete algorithm, and every final state would be labeled clearly. The pattern matching process begins from the start (initial) state and ends if there is no more input characters of the text. And the algorithm outputs the keystrings when passing through the final state.

4.1 The Multi-Final-State-Class DFA Constructing Algorithm

This section presents how to construct a mc-DFA with multiple keywords. Make $\Sigma$ denote the set of alphabets. $K = \{K_1, K_2, \ldots, K_n\}$ be the finite set of keystrings and $T$ be the text which will be compared with $K$.

Algorithm 4.1.1 (mc-DFA Constructing)

Input: $n$ distinct nonempty keystrings $K_1, K_2, \ldots, K_n, n \geq 1$.
Some Variables:
- $FS_i$: the set of final states for the keystring $K_i$, $1 \leq i \leq n$.
- $D_S$: a binary relation denoting the production rules from $S$, i.e., $(a, S') \in D_S$ iff $\delta(S, a) = S'$.
- $\delta(S^{(0)}, a_1a_2\cdots a_i) = S^{(i)}$ if $(a_j, S^{(j)}) \in D_{S^{(j-1)}}$ for $1 \leq j \leq i$.
- $C(S)$: the word such that $\delta(S_0, C(S)) = S$ according to $\delta$ established in Step 1.
- $I(S)$: the set of numbers $i$ which indicate the identifications of keystrings $K_i$ when the state $S$ is reached.

Output: A complete mc-DFA $M_{mc} = (Q, \Sigma, \delta, S_0, \bigcup_{i=1}^{n} FS_i)$

Step 0: Initiation:
- 0.1 Construct the set $\Sigma$ of letters used by $K_i$ for $1 \leq i \leq n$.
- 0.2 Let $FS_i = \emptyset$, $1 \leq i \leq n$.
- 0.3 Suppose $K_i = p_{i1}p_{i2}\cdots p_{ik_i}$, $1 \leq i \leq n$, where $p_{ij} \in \Sigma$.
- 0.4 Let $k_i = \log(K_i)$ for $1 \leq i \leq n$.

Step 1: Establish states and state transitions:
- 1.1 Let $Q = \{S_0\}$, $D_{S_0} = \emptyset$, $S_{i0} = S_0$ for $1 \leq i \leq n$, $C(S_0) = \epsilon$, $I(S_0) = \emptyset$, $q = 1$.
- 1.2 For $i = 1$ to $n$ do
- 1.3 For $j = 1$ to $k_i$ do
1.4 If $p_{ij} \in \pi_1(D_{S_{i(j-1)}})$,
1.5 Then Let $S_{ij} = S'$ where $(p_{ij}, S') \in D_{S_{i(j-1)}}$.
1.6 Else Let $Q = Q \cup \{S_q\}$, $S_{ij} = S_q$, $D_{S_q} = \emptyset$, $D_{S_{i(j-1)}} = D_{S_{i(j-1)}} \cup \{(p_{ij}, S_q)\}$,
$C(S_q) = p_1p_2\cdots p_{ij}$, $I(S_q) = \emptyset$ and $q = q + 1$.
1.7 End If
1.8 Next $j$.
1.9 $FS_i = FS_i \cup \{S_{ik}\}$ and $I(S_{ik}) = I(S_{ik}) \cup \{i\}$.
1.10 Next $i$.

**Step 2: Complete the state transitions:**

2.1 For any $b \in \Sigma \setminus \pi_1(D_{S_0})$, let $D_{S_0} = D_{S_0} \cup \{(b, S_0)\}$.
2.2 For $i = 1$ to $n$ do
2.3 While $\Sigma \setminus \pi_1(D_{S_{i+1}}) \neq \emptyset$ do
2.4 Choose $b \in \Sigma \setminus \pi_1(D_{S_{i+1}})$, let $D_{S_{i+1}} = D_{S_{i+1}} \cup \{(b, \delta(S_0, b))\}$.
2.5 End While
2.6 Next $i$.
2.7 Let $M = \max\{k_i \mid i = 1, 2, \ldots, n\}$. (Note that $k_i = \lg(K_i)$ for $1 \leq i \leq n$.)
2.8 While $M \geq 2$ do
2.9 For $j = 2$ to $M$ do
2.10 For $i = 1$ to $n$ do
2.11 If $j \leq k_i$, then
2.12 Let $v = \lambda(C(S_{ij}))$, $S = \delta(S_0, v)$, $I(S_{ij}) = I(S_{ij}) \cup I(S)$
and $FS_k = FS_k \cup \{S_{ij}\}$ for $k \in I(S)$.
2.13 While $\Sigma \setminus \pi_1(D_{S_{ij}}) \neq \emptyset$ do
2.14 Choose $b \in \Sigma \setminus \pi_1(D_{S_{ij}})$, let $D_{S_{ij}} = D_{S_{ij}} \cup \{(b, \delta(S_0, vb))\}$.
2.15 End While
2.16 End If
2.17 Next $i$
2.18 Next $j$
2.19 $M = 0$.
2.20 End While

---

*Algorithm 4.1.1: Constructing $mc$-$DFA$*

Algorithm 4.1.1 describes an algorithm which constructs the complete $mc$-$DFA$ and can be applied in pattern matching algorithm to compare with multiple classes of keystrengs concurrently. The following example given shows how the algorithm works.
**Remark 4.1.1** After Step 1 of algorithm 4.1.1, the automaton established is a rooted directed tree.

**Example 4.1.1** Let $K_1 = TAC$, $K_2 = AC$ be the keystrings over the alphabet $\Sigma = \{A, C, G, T\}$. After Step 1, we have $D_{S_0} = \{(A, S_4), (T, S_1)\}$, $D_{S_1} = \{(A, S_2)\}$, $D_{S_2} = \{(C, S_3)\}$, $D_{S_4} = \{(C, S_5)\}$, $FS_1 = \{S_3\}$, $FS_2 = \{S_3\}$ and the variables $C(S)$ and $I(S)$ is:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>$\epsilon$</td>
<td>$T$</td>
<td>$TA$</td>
<td>$TAC$</td>
<td>$A$</td>
<td>$AC$</td>
</tr>
<tr>
<td>$I$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>${2}$</td>
</tr>
</tbody>
</table>

The tree diagram is:

![Tree Diagram](image-url)

The main purpose of Step 2 is completing the automaton which is constructed...
in Step 1 such that each state \( q \) of \( Q \) according to each alphabet \( a \) of \( \Sigma \), \( \delta(q,a) \) must be defined. After Step 2, we have

\[ D_{S_0} = \{(A, S_4), (C, S_0), (T, S_1), (G, S_0)\}, \]

\[ D_{S_1} = \{(A, S_2), (C, S_0), (T, S_1), (G, S_0)\}, \]

\[ D_{S_2} = \{(A, S_4), (C, S_3), (T, S_1), (G, S_0)\}, \]

\[ D_{S_3} = \{(A, S_4), (C, S_0), (T, S_1), (G, S_0)\}, \]

\[ D_{S_4} = \{(A, S_4), (C, S_5), (T, S_1), (G, S_0)\}, \]

\[ D_{S_5} = \{(A, S_4), (C, S_0), (T, S_1), (G, S_0)\}. \]

\[ \delta_{S_0} \]

\[ \delta_{S_1} \]

\[ \delta_{S_2} \]

\[ \delta_{S_3} \]

\[ \delta_{S_4} \]

\[ \delta_{S_5} \]

and \( FS_1 = \{S_3\}, FS_2 = \{S_3, S_5\} \). The \( I \) value in Step 1 is changed as:

\[ I 0 1 2 3 4 5 \]

\[ 0 0 0 \{1,2\} 0 \{2\} \]

The complete mc-DFA is:

![DFA Diagram](image-url)

Figure 4.2: DFA
Next, some properties of the Algorithm 4.1.1 are given.

**Remark 4.1.2** For the mc-DFA obtained in Algorithm 4.1.1, if $\delta(S_0, w) = S$ and $v$ is the shortest word such that $\delta(S_0, v) = S$, denoted by $C(S)$ in Algorithm 4.1.1, then $v$ is the longest suffix of $w$ such that $v \leq_p K$ for some keystring $K$.

**Proposition 4.1.1** Let $M = (Q, \Sigma, \delta, S_0, \bigcup_{i=1}^n F S_i)$ be a complete mc-DFA constructed in Algorithm 4.1.1. For any $w \in \Sigma^+$, if $\delta(S_0, w) \neq S_0$ then there is a unique nonempty suffix of $w$ such that $\delta(S_0, v) = \delta(S_0, w)$ and $v \leq_p K$ for some keystring $K$. Note that such $v$ is the longest nonempty suffix of $w$ such that $v \leq_p K$ for some keystring $K$.

**Proof.** The set of states, $Q$, is initiated in Step 1.1 and extended in Step 1.6. In Step 2, nothing in $Q$ has changed. By virtue of Step 1, for each $S \in Q \setminus \{S_0\}$, there is a unique word $v \in \Sigma^+$ such that $\delta(S_0, S) = S$ and $v \leq_p K$ for some keystring $K$. Therefore, for any $w \in \Sigma^+$, if $\delta(S_0, w) \neq S_0$, then there is a unique word $v \in \Sigma^+$ such that $\delta(S_0, v) = \delta(S_0, w)$ and $v \leq_p K$ for some keystring $K$. Now, we are going to show that $v \leq_s w$. Let $w = a_1 a_2 \cdots a_k$ for some $k \geq 1$ and let $S_i = \delta(S_{i-1}, a_i)$ for $1 \leq i \leq k$. If for every $1 \leq i \leq k$, $(a_i, S_i) \in D_{S_{i-1}}$ is established in Step 1, then $w = v$. Clearly, $v \leq_s w$ and $v$ is the longest such suffix of $w$. Suppose there is $1 \leq i \leq k$ such that...
\((a_i, S_i) \in D_{S_{i-1}}\) is established in Step 2. Let \(i\) be the largest such number. Then by Step 2.1, 2.4 and 2.14, \(C(S_i) \leq a_1 a_2 \cdots a_i\) and \(\delta(S_0, C(S_i)) = \delta(S_0, a_1 \cdots a_i) = S_i\). Then by virtue of the constructions of \(\delta(S_0, K)\) and \(C(S_i)\), \(v = C(S_i) a_{i+1} \cdots a_k\). If \(i = k\), then \(v = C(S_i) \leq a_1 a_2 \cdots a_i = w\). If \(i \neq k\), as each \((a_r, S_r) \in D_{S_{r-1}}, i + 1 \leq r \leq k\), is established in Step 1, it follows from Step 1 that \(v = C(S_i) a_{i+1} \cdots a_k = C(\delta(S_0, w))\). Thus \(v = C(S_i) a_{i+1} \cdots a_k \leq a_1 a_2 \cdots a_i a_{i+1} \cdots a_k = w\). As \(\delta(S_0, w) = \delta(S_0, C(S_i) a_{i+1} \cdots a_k) = \delta(S_0, v)\) and \(v = C(\delta(S_0, w))\), from Remark 4.1.2, it follows that \(v\) is the longest such suffix of \(w\).

**Proposition 4.1.2** Let keystings \(K_1, K_2, \ldots, K_n\) have the property that a keystring is a proper infix of any other keystring if it is a proper prefix of that keystring. Let \(M = (Q, \Sigma, \delta, S_0, \bigcup^n_{i=1} F S_i)\) be the complete mc-DFA constructed in Algorithm 4.1.1 for inputs \(K_1, K_2, \ldots, K_n\). Then for any keystring \(K_i\) and any \(x \in \Sigma^*\), \(\delta(S_0, xK_i) = \delta(S_0, K_i)\).

**Proof.** As \(K_i \neq \epsilon\), if \(\delta(S_0, xK_i) = S_0\) then \(\delta(S_0, sK_i) = \delta(S_0, b)\) for \(b \in \Sigma \setminus D_{S_0}\) (in Step 2.1) and there is no nonempty suffix of \(xK_i\) which is a prefix of a keystring. This contradicts the fact that \(K_i\) is a nonempty suffix of \(xK_i\). Thus \(\delta(S_0, xK_i) \neq S_0\). By Proposition 4.1.1, there is a longest suffix \(v\) of \(xK_i\) such that \(\delta(S_0, xK_i) = \delta(S_0, v)\) and \(v \leq_p K\) for some keystring \(K\). Clearly, \(\log(v) \geq \log(K_i)\). If \(\log(v) > \log(K_i)\) then \(K_i\) is a proper infix but not a prefix of \(K\) which
contradicts the condition of keystrings $K_1, K_2, \ldots, K_n$. Thus $\log(v) = \log(K_i)$, i.e., $v = K_i$.

**Proposition 4.1.3** Let $K_1, K_2, \ldots, K_n$ be $n$ distinct nonempty keystrings which have the property that a keystring is a proper infix of another keystring if it is a proper prefix of that keystring. Then the mc-DFA constructed in Algorithm 4.1.1 can detect the occurrences of these keystrings and has the property that each $FS_i$ is a singleton.

**Proof.** By virtue of Step 1, when Algorithm 4.1.1 is used to process a keystring $K_i$, only one state $S_{ik_i}$ is added into $FS_i$. Thus each $FS_i$ is a singleton. Suppose keystrings $K_1, K_2, \ldots, K_n$ have the property that a keystring is a proper infix of any other keystring if it is a proper prefix of that keystring. Let $w$ be a word such that $w = xK_iy$ for some $x, y \in \Sigma^*$ and a keystring $K_i$. By Proposition 4.1.2, $\delta(S_0, xK_i) = \delta(S_0, K_i) = S_{ik_i} \in FS_i$. Thus substring $K_i$ can be detected correctly.

**Proposition 4.1.4** The mc-DFA constructed in Algorithm 4.1.1 can detect the occurrences of nonempty keystrings $K_1, K_2, \ldots, K_n$ in a text string (even when keystrings have np-infix relations).

**Proof.** Let $w = xK_iy$ for some $x, y \in \Sigma^*$ and a keystring $K_i$. If $K_i$ is not an np-infix of any other keystring, then by virtue of Proposition 4.1.3, $K_i$ can
be detected correctly. If $x = \epsilon$, then by the construction of the rooted tree constructed in Step 1 of Algorithm 4.1.1, $\delta(S_0, K_i) \in F S_i$. The occurrence of $K_i$ can be detected correctly. Now, suppose that $x \neq \epsilon$ and $K_i$ is an np-infix of some keystrings. Consider the following two cases:

(1) There is a keystring $K_j$ such that $K_j = x_1K_iy_1$ for some $x_1 \in \Sigma^+$ and $y_1 \in \Sigma^*$ with $x = x_2x_1$ and $y = y_1y_2$ for some $x_2, y_2 \in \Sigma^*$. Let $K_j$ be such a keystring with the longest $x_1$. Then by virtue of Proposition 4.1.2, $\delta(S_0, x_2K_j) = \delta(S_0, K_j)$. That is, $\delta(S_0, x_2x_1K_i) = \delta(S_0, x_1K_i)$. Since $\delta(S_0, K_i) \in F S_i$, $i \in I(\delta(S_0, K_i))$. By Step 2.12, $\delta(S_0, x_1K_i) \in F S_i$. Thus $\delta(S_0, x_2x_1K_i) = \delta(S_0, x_1K_i) \in F S_i$. This yields that $K_i$ can be identified correctly.

(2) There is no keystring $K_j$ such that $K_j = x_1K_iy_1$ for some $x_1 \in \Sigma^+$ and $y_1 \in \Sigma^*$ with $x = x_2x_1$ and $y = y_1y_2$ for some $x_2, y_2 \in \Sigma^*$. Then by virtue of Proposition 4.1.2, $\delta(S_0, xK_i) = \delta(S_0, K_i)$. Thus the occurrence of $K_i$ can be detected correctly.

### 4.2 Minimization

Regarding this section, an algorithm is proposed to reduce the number of states of an mc-DFA such that the number of states are minimal and the
minimized mc-DFA never changes the properties of the original mc-DFA.

We say that a state $S$ is *unreachable* if there is no word $w \in \Sigma^*$ such that $\delta(S_0, w) = S$, where $S_0$ is the initial state. A state $S$ is *useless* if it is unreachable or there is no word $w \in \Sigma^*$ such that $\delta(S, w) \in F$. None of the states established in Algorithm 4.1.1 are useless. So that the issue of the minimization is considering which states must be merged.

**Definition 4.2.1** Let $M_{mc} = (Q, \Sigma, \delta, s, \bigcup_{i=1}^{n} F_i)$ be an mc-DFA. Two distinct states $p, q \in Q$ are mc-distinguishable if there exists a word $x \in \Sigma^*$ such that $\delta(p, x) \in F_i$ for some $F_i$ and $\delta(q, x) \in Q \setminus F_i$, or vice versa. Then the word $x$ mc-distinguishes $p$ from $q$.

If there is no such word $x \in \Sigma^*$, then the two states $p$ and $q$ are mc-undistinguishable. In other words, two state $p$ and $q$ are mc-undistinguishable whenever $\delta(p, x) \in F_i \iff \delta(q, x) \in F_i$ for any $x \in \Sigma^*$ and $F_i$, $1 \leq i \leq n$.

**Definition 4.2.2** Let $M_{mc} = (Q, \Sigma, \delta, s, \bigcup_{i=1}^{n} F_i)$ be an mc-DFA and $p, q$ two distinct states in $Q$. If $p$ is mc-distinguished from $q$ by a word $x \in \Sigma^*$ with length $k$ for some $k \geq 0$, then $p$ and $q$ are said to be $k$-mc-distinguishable. Otherwise, $p$ and $q$ are called $k$-mc-undistinguishable, denoted by $p \equiv_{mc}^k q$.

By Definition 4.2.2, we say that two states $p$ and $q$ are $k$-mc-distinguishable for some $k \geq 0$ if and only if there exists a word $x \in \Sigma^*$ with length $k$ such
that $\delta(p, x) \not\equiv^{0}_{mc} \delta(q, x)$.

**Remark 4.2.1** Given an mc-DFA $M_{mc} = (Q, \Sigma, \delta, s, \bigcup_{i=1}^{n} F_i)$. Let $F_{n+1} = Q \setminus \bigcup_{i=1}^{n} F_i$. Then for any $p, q \in Q$, $p \not\equiv^{0}_{mc} q$ if and only if $p \in F_i$ and $q \in F_j$ for some $i \neq j$.

**Proposition 4.2.1** Given an mc-DFA $M_{mc} = (Q, \Sigma, \delta, s, \bigcup_{i=1}^{n} F_i)$. For any $p, q \in Q$, $p \not\equiv^{k}_{mc} q$ for some $k \geq 1$ if and only if there is $a \in \Sigma$ such that $\delta(p, a) \not\equiv^{k-1}_{mc} \delta(q, a)$.

From Remark 4.2.1 and Proposition 4.2.1, an mc-DFA minimization algorithm is proposed to minimize the number of the states of the mc-DFA $M_{mc}$.

We claim that any states can be merged if they are mc-undistinguished.

**Algorithm 4.2.1** *(mc-DFA Minimization)*

**Input:** An mc-DFA $M_{mc} = (Q, \Sigma, \delta, s, \bigcup_{i=1}^{n} F_i)$

**Output:** A minimized mc-DFA $M'_{mc} = ((Q_1, Q_2, \ldots, Q_N), \Sigma, \delta', s', \bigcup_{i=1}^{n} F'_i)$.

**Step 1:** Let $Q_j = F_j$ for $1 \leq j \leq n$, $Q_{n+1} = Q \setminus \bigcup_{i=1}^{n} F_i$, $I = n$, $N = n + 1$ and $Flag = 1$.

**Step 2:** *(Distinguish splitting)*

2.1 While $Flag = 1$ do (Flag = 0 means that no more set is split.)

2.2 $Flag = 0$

2.3 For $a \in \Sigma$ do

2.4 For $i = 1$ to $N$ do

2.5 Let $R = Q_i$, $R_{i+1} = Q_i$, $r = 0$.

2.6 While $R \neq 0$ do

2.7 $r = r + 1$, $I = I + 1$.

2.8 Choose any $p \in R$, let $Q' = R \setminus \{p\}$ and $R_{i+1} = \emptyset$.

2.9 While $Q' \neq \emptyset$ do

2.10 Choose $q \in Q'$, let $Q' = Q' \setminus \{q\}$.

2.11 If $\delta(p, a) \in Q_j$ and $\delta(q, a) \in Q_k$ for some $Q_j \neq Q_k$,

then $R_{i+1} = R_{i+1} \cup \{q\}$, $R_i = R_i \setminus \{q\}$ and $Flag = 1$.

38
2.12 End While
2.13 Let $R = R_{i+1}$
2.14 End While
2.15 Next $i$
2.16 Let $N = I$, rename \{\(R_{11}, R_{12}, \ldots, R_{1s}, R_{21}, \ldots, R_{2s}, \ldots, R_{n1}, \ldots, R_{ns}\)\} as \{\(Q_1, Q_2, \ldots, Q_N\)\}.
2.17 Next $a$
2.18 End While

Step 3: Output the mc-DFA $M'_{mc} = (\{Q_1, Q_2, \ldots, Q_N\}, \Sigma, \delta', S', \bigcup_{i=1}^{n} F'_i)$, where $S' = Q_j$ if $s \in Q_j$, $F'_i = \{Q_r | Q_r \subseteq F_i, 1 \leq r \leq N\}$, $1 \leq i \leq n$, and $\delta'(Q_i, a) = Q_r$ for any $a \in \Sigma$ if there exist $p \in Q_i$ and $q \in Q_r$ such that $\delta(p, a) = q$.

Remark 4.2.2 Let $(Q, \Sigma, \delta, s, \bigcup_{i=1}^{n} F_S)$ be a complete mc-DFA constructed in Algorithm 4.1.1. Then for any keystring $K_i$ and any word $v \in \Sigma^* \setminus \{K_i\}$ with $\lg(v) \leq \lg(K_i)$, $\delta(s, v) \notin F_S$.

Proposition 4.2.2 The complete mc-DFA $(Q, \Sigma, \delta, s, \bigcup_{i=1}^{n} F_S)$ obtained in Algorithm 4.1.1 is a minimal DFA.

Proof. Assume that there are two distinct states $p$ and $q$ such that $p \equiv_{mc} q$. Clearly, $C(p) \neq C(q)$. From Algorithm 4.1.1, it follows that there are $x_1, y_1 \in \Sigma^*$ such that $C(P)x_1 = K_i$ and $C(q)y_1 = K_j$ for some keystrings $K_i$ and $K_j$. If $\lg(C(p)) < \lg(C(q))$, then by Remark 4.2.2, $\delta(s, C(p)y_1) \notin F_S$ whereas $\delta(s, C(q)y_1) = \delta(s, K_j) \in F_S$. This contradicts the assumption that $p \equiv_{mc} q$. Similarly, $\lg(C(q)) < \lg(C(p))$ is impossible. Thus $\lg(C(p)) = \lg(C(q))$. Suppose $\lg(x_1) < \lg(y_1)$. This together with $C(p) \neq C(q)$, by Remark 4.2.2, yields $\delta(s, C(q)x_1) \notin F_S$ whereas $\delta(s, C(p)x_1) = \delta(s, K_i) \in F_S$. Therefore, $p \equiv_{mc} q$.
This contradicts the assumption that $p \equiv_{mc} q$. Similarly, $\lg(y_1) \leq \lg(x_1)$ is impossible. Therefore, $p \not\equiv_{mc} q$ for any two distinct states $p$ and $q$.

### 4.3 Minimized Concurrent Matching DFA Generating Algorithm

**Algorithm 4.3.1** *(mc-DFA for multi-class of keystrings)*

**Input:** $m$ distinct classes $C_1, C_2, \ldots, C_m$ of nonempty keystrings with $m \geq 1$.

**Output:** An mc-DFA $M_{mc} = (Q, \Sigma, \delta, S, \bigcup_{i=1}^{m} F_i)$.

**Step 1:**

1.1 Let $r_0 = 0$, $C_i = \{K_{r_i-1+1}, \ldots, K_{r_i}\}$ for $1 \leq i \leq m$.

1.2 Apply Algorithm 4.1.1 to process $K_1, \ldots, K_m$ and construct a complete mc-DFA $M' = (Q, \Sigma, \delta, S_0, \bigcup_{i=1}^{m} F_{S_i})$.

**Step 2:** Output the mc-DFA $M_{mc} = (Q, \Sigma, \delta, S_0, \bigcup_{i=1}^{m} F_{S_i})$ where $F_i = \bigcup_{j=r_i-1+1}^{r_i} F_{S_j}$.

**Definition 4.3.1** Let $M_{mc} = (Q, \Sigma, \delta, \bigcup_{i=1}^{m} F_i)$ be an mc-DFA. The language accepted by each DFA $(Q, \Sigma, \delta, s, F_i)$, where $1 \leq i \leq n$, is $L_i(M_{mc}) = \{x \in \Sigma^* \mid \delta(s, x) \in F_i\}$.

**Algorithm 4.3.2** The minimized matching mc-DFA generating algorithm

**Input:** Classes of keystrings $C_i$, $i = 1, 2, \ldots, m$.

**Output:** An mc-DFA $M'_{mc}$ with the minimal number of states and $m$ disjoint classes of final states such that $L_i(M'_{mc}) = \Sigma^* C_i$ for $i = 1, 2, \ldots, m$.

**Step 1:** Apply Algorithm 4.3.1 to construct an mc-DFA $M_{mc} = (Q, \Sigma, \delta, \bigcup_{i=1}^{m} F_i)$ for multiple classes of keystrings, i.e., $F_i$ denotes the occurrences of keystrings in $C_i$.

**Step 2:** Apply Algorithm 4.2.1 to construct a minimized mc-DFA $M'_{mc} = (Q', \Sigma, \delta', \bigcup_{i=1}^{m} F'_i)$.

**Step 3:** Set the output function for this $M'_{mc}$.
For the string matching, the mc-DFA obtained in Algorithm 4.3.2 can detect the occurrences of multiple keystring classes in a text string concurrently.

It is not difficult to show that the constructing time complexity of Algorithm 4.3.2 is $O(n)$ (here, $n$ denotes the sum of the lengths of keystrians).
Chapter 5

A prefix square check algorithm

In this chapter, an exact pattern matching algorithm is proposed to check
the prefix square (defined in Chapter 3) in a string. We improved the next
function algorithm of the K.M.P algorithm [6] and present an algorithm that
not only can on-line check the inputted string but also can detect the prefix
square of the inputted string.

5.1 Prefix Square Detection Algorithm

In this section, a pattern matching algorithm is proposed to check whether
there is a prefix square when a string is inputted. The next function algorithm
proposed in [6] was used to find the next position of the pattern to compare
with the text when a mismatch occurs. And the next function obtained by
comparing the prefix with the suffix of a pattern string. Consider that a prefix
square occurs if and only if the prefix of the string matches the suffix of the
string and the length of the prefix equals the length of the suffix. In other
words, a string \( v \) is a prefix square of a string \( u \) if and only if \( u = v^2w \),
for some \( w \in \Sigma^* \). Next, we present an algorithm which modifies the ‘next
function’ algorithm and can detect the prefix square. Suppose a string of
letters \( a_1, a_2, \ldots, a_n, \ldots \) is inputted.

**Algorithm 5.1.1**

**Input:** \( a_j, j = 1, 2, \ldots \)

**Variables:**
- \( n_k \): the length of the longest proper bifix of \( a_1, a_2, \ldots, a_k \);
- \( j \): an index denotes the letter \( a_j \) inputted;
- \( i + 1 \): an index denotes the letter \( a_{i+1} \) which is compared with the inputted letter \( a_j \).

**Output:** The repeated prefix \( a_1 \cdots a_{n_j} \) once it is detected.

1. \( n_1 := 0, j := 2, i := n_1 \).
2. **While** \( a_j \neq \text{Nil} \) **Do**
   3. **While** \( a_j \neq a_{i+1} \) and \( i \neq 0 \) **Do**
   4. \( i := n_i \).
   5. **End While**
   6. **If** \( i = 0 \)
   7. **Then** \( n_j := 0 \).
   8. **End If**
   9. **If** \( a_j = a_{i+1} \)
   10. **Then** \( n_j := i + 1, i := n_j \).
11. **End If**
12. **If** \( n_j = \frac{j}{2} \)
13. **Then** **Output:** A prefix \( a_1 \cdots a_{n_j} \) is repeated; **Exit**.
14. **End If**
15. \( j := j + 1 \)
16. **End While**
17. **Output:** There is no prefix square.

Algorithm 5.1.1 is an on-line prefix square algorithm, while the string let-
ters \( a_j, j = 1, 2 \cdots \) is inputted, the algorithm computes the length of the
longest proper bifix \( n_k \) of \( a_1 a_2 \cdots a_k \) for each \( k \). While the length of the longest
bifix of $a_1 a_2 \cdots a_k$ equals to $\frac{k}{2}$, the algorithm outputs that a prefix square occurs, and the repeated prefix is $a_1 a_2 \cdots a_{\frac{k}{2}}$. In other words, algorithm 5.1.1 can output $v = a_1 a_2 \cdots a_{\frac{k}{2}}$ if the inputted string $u = v^2$ and $\log(u) = 2 \log(v)$.

**Example 5.1.1** Consider the DNA string $u = ACGACGTACGACGTCA$, when the string $u$ is inputted, the related variables $j, a_j, n_j$ of Algorithm 5.1.1 are listed as in Figure 5.1. See Figure 5.1, while the letter $a_4$ is inputted, $n_4$ equals to 1 means that a proper prefix $A$ of the string $ACGA$ matches the proper suffix of that string. While the letter $a_6$ is inputted, $n_6 = 3 = \frac{6}{2}$, that is, a prefix square occurs, the output is $ACG$. The same reason, $n_{16} = 8 = \frac{16}{2}$, the repeated prefix is $a_1 a_2 \cdots a_8 = ACGACGTCA$.

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| $a_j$ | A | C | G | A | C | G | T | C | A | C | G | A | C | G | T | C |
| $n_j$  | 0 | 0 | 0 | 1 | 2 | 3 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Figure 5.1: $n_j$ table

### 5.2 Properties of Algorithm 5.1.1

In this part, we provide some properties of the variable $n_j$, and reveal the correctness of the Algorithm 5.1.1.
**Proposition 5.2.1** Let $j \geq 1$. The number $n_j$ of Algorithm 5.1.1 denotes the length of the longest proper prefix $v$ of $a_1a_2 \cdots a_j$ such that $v$ is also a suffix of $a_1a_2 \cdots a_j$.

**Proof.** Line 1 sets $n_1$ to 0 and $i$ to $n_1$. This means the only such proper prefix is the empty word $\lambda$. Suppose that there is $k \geq 1$ such that for any $1 \leq j \leq k$, $n_j$ denotes the length of the longest proper prefix $v$ of $a_1a_2 \cdots a_j$ such that $v$ is also a suffix of $a_1a_2 \cdots a_j$. Consider $n_{k+1}$. From Line 7 and Line 10, it is clear that $i = n_k$ when $j = k + 1$ and Line 3 is executed. If $a_{k+1} \neq a_{i+1}$ then $a_1a_2 \cdots a_{i+1} \neq a_{k-i+1}a_{k-i+2} \cdots a_{k+1}$. As $a_1a_2 \cdots a_i = a_{k-i+1}a_{k-i+2} \cdots a_k$ whenever $i \neq 0$, the longest proper prefix $v$ of $a_1a_2 \cdots a_i$ such that $v$ is a suffix of $a_k$ is the next proper prefix and suffix of $a_1a_2 \cdots a_k$, which must be considered. Moreover, $a_1a_2 \cdots a_i = a_{k-i+1}a_{k-i+2} \cdots a_k$ yields that $v$ is a suffix of $a_1a_2 \cdots a_i$ and $lg(v) = n_i$. Thus Line 4 sets $i$ to $n_i$. If $i = 0$, then we need only check whether $a_1 = a_{k+1}$. Therefore, one of Line 7 and Line 10 will set $n_{k+1}$ to the length of the longest proper prefix $v$ of $a_1a_2 \cdots a_{k+1}$ such that $v$ is also a suffix of $a_1a_2 \cdots a_{k+1}$. By mathematical induction on the index $j$, the assertion holds.

**Proposition 5.2.2** Algorithm 5.1.1 can find the first prefix repetition of a word.
Proposition 5.2.3 $n_j \leq \frac{j}{2}$ and $n_j < \frac{j}{2}$ except the last $n_j$. Hence, if $n_j \neq 0$ then $n_{n_j} < \frac{n_j}{2}$.

Proof. From Proposition 5.2.1 and the Line 12 of Algorithm 5.1.1, it follows that $n_j \leq \frac{j}{2}$. Since the algorithm ends when $n_j = \frac{j}{2}$ or $a_j = \text{Nil}$, the case $n_j = \frac{j}{2}$ can only hold for the last $n_j$. It is then clear that if $n_j \neq 0$ then $n_{n_j} < \frac{n_j}{2}$.

5.3 The Time Complexity of Algorithm 5.1.1

In this section, some properties are given to analyze the time complexity of the Algorithm 5.1.1 and a variable named $4_j$ is given to display the times of Line 4 of Algorithm 5.1.1 to be executed when $a_j$ is inputted. The following figure lists the value of the variable $4_j$ in Example 5.1.1.

<table>
<thead>
<tr>
<th>$j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_j$</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td>$n_j$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$4_j$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.2: $n_j$ and $4_j$ table

Lemma 5.3.1

(1) $4_1 = 0$ and $4_2 = 0$. 

46
(2) If \( a_j \) is processed for some \( j \geq 3 \), then \( n_2 = 0 \) and \( 4_3 = 0 \).

(3) If \( n_j = 0 \), then \( 4_{j+1} = 0 \) and \( n_{j+1} = 0 \) or 1.

(4) If \( n_j = 1 \), then \( 4_{j+1} = 0 \) or 1.

(5) If \( n_j = n_{j-1} + 1 \), then \( 4_j = 0 \).

(6) If \( n_j \neq 0 \) and \( 4_{j+1} = 0 \), then \( n_{j+1} = n_j + 1 \).

Proof. By virtue of Line 1 of Algorithm 5.1.1, it is clear that \( 4_1 = 0 \) and \( 4_2 = 0 \). If \( a_2 = a_1 \), then Algorithm 5.1.1 ends. Thus \( a_j \) being processed for an integer \( j \geq 3 \) means that \( a_2 \neq a_1 \) and \( n_2 = 0 \). This yields \( 4_3 = 0 \). From Line 3 of Algorithm 5.1.1, it is clear that if \( n_j = 0 \) then \( 4_{j+1} = 0 \); by Lines 7 and 10, \( n_{j+1} = 0 \) or 1. If \( n_j = 1 \) and \( a_{j+1} = a_{n_{j+1}} \), then \( 4_{j+1} = 0 \). Let \( i = n_j \). As \( n_1 = 0 \), if \( n_j = 1 \) and \( a_{j+1} \neq a_{n_{j+1}} = a_2 \), then by Line 4 of Algorithm 5.1.1, \( i = n_{n_j} = n_1 = 0 \). By Lines 7 and 10 of Algorithm 5.1.1, \( n_{j+1} = 0 \) or 1. Lines 9 and 10 of Algorithm 5.1.1 yield that \( n_j = n_{j-1} + 1 \) holds only if \( a_j = a_{n_{j-1} + 1} \). Thus \( n_j = n_{j-1} + 1 \) implies \( 4_j = 0 \). If \( n_j \neq 0 \) and \( 4_{j+1} = 0 \), then by Line 3 of Algorithm 5.1.1, \( a_{j+1} = a_{n_{j+1}} \). From Line 10 of Algorithm 5.1.1, it follows \( n_{j+1} = n_j + 1 \).

Proposition 5.3.1 Let \( 4_j = r \geq 1 \), \( i_1 = n_{j-1} \) and \( i_{s+1} = n_{i_s} \) for \( 1 \leq s \leq r \). Then \( n_j = n_{i_{r+1}} \) or \( n_j = 0 \).
Proof. In view of the while loop: Lines 3-5 of Algorithm 5.1.1, it follows that $i_s \neq 0$ and $a_{n_j} \neq a_{i_s+1}$ for $1 \leq s \leq r$. And each time the loop resets $i$ from $i_s$ to $i_{s+1}$, where $1 \leq s \leq r$. If $a_j = a_{i_s+1}$, then by Line 10 of Algorithm 5.1.1, $n_j = n_{n_{i_s}+1}$. The case $a_j \neq a_{i_s+1}$ can hold only when $n_{i_s} = 0$. In this case, by virtue of Line 7 of Algorithm 5.1.1, $n_j = 0$.

**Proposition 5.3.2** $4_j < \log_2 j$ for any $j \geq 1$.

Proof. By Lemma 5.3.1, $4_1 = 0$, $4_2 = 0$, and $4_3 = 0$ if they are counted. Thus $4_j < \log_2 j$ for $1 \leq j \leq 3$. For $j \geq 4$, clearly if $4_j = 0$ or $4_j = 1$, then $4_j < \log_2 j$. Now let $j \geq 4$ and $4_j = r \geq 2$. Let $i_1 = n_{j-1}$ and $i_{s+1} = n_{i_s}$ for $1 \leq s \leq r$. In view of the while loop: Lines 3-5 of Algorithm 5.1.1, it follows that $i_s \neq 0$ for $1 \leq s \leq r$. By virtue of Proposition 5.2.3, $i_{s+1} < i_s/2$ for $1 \leq s \leq r$. Since $i_1 = n_{j-1}$ is not the last one, by Proposition 5.2.3, $i_1 < \frac{j-1}{2}$. Thus $i_r < \frac{i_1}{2^{j-r}} < \frac{j}{2}$. As $i_r \neq 0$, $i_r \geq 1$. This yields that $1 < \frac{j}{2}$. Hence $0 < (\log_2 j) - r$, i.e., $r < \log_2 j$.

For the given string $w = a_1a_2\cdots a_n$ precessed in Algorithm 5.1.1, let

$$m_j = \begin{cases} 
4_j - 1 & \text{if } 4_j \neq 0 \smallskip 
0 & \text{if } 4_j = 0 
\end{cases}, \quad 
0_j = \begin{cases} 
0 & \text{if } 4_j \neq 0 
1 & \text{if } 4_j = 0 
\end{cases} \quad \text{and } \beta_j = \sum_{i=1}^{j} (0_j - m_j).
$$

The following proposition concerns the execution time of Line 4 which is compared with the number $n_j$. 

48
Proposition 5.3.3 \( \beta_j > n_j \) for each \( j \geq 1 \).

Proof. By Lemma 5.3.1, \( 4_1 = 0 \) and \( 4_2 = 0 \), and \( 4_3 = 0 \) if it is counted. Thus \( \beta_1 = 1, \beta_2 = 2 \) and \( \beta_3 = 3 \). It is clear that \( n_1 = 0 \) and \( n_2 \leq 1 \). And, \( n_2 = 0 \) when \( j \geq 3 \) is considered. Now, if \( n_3 = 0 \), then \( 4_4 = 0 \). Otherwise, \( n_3 = 1 \) and \( a_3 = a_1 \). In this case, if \( a_4 = a_2 \), then \( 4_4 = 0, n_4 = 2 \) and Algorithm 5.1.1 ends.

If \( a_4 \neq a_2 \), then \( 4_4 = 1 \) and \( n_4 = 0 \) or 1. These observations yield \( \beta_j > n_j \) for each \( 1 \leq j \leq 4 \). Assume that there is \( r \geq 4 \) such that for each \( 1 \leq j \leq r, \beta_j > n_j \). Now, consider \( \beta_{r+1} \) and \( n_{r+1} \). We have the following two cases:

(1) \( n_{r+1} = n_r + 1 \). Then by Lemma 5.3.1, \( 4_{r+1} = 0 \). This implies that \( \beta_{r+1} = \beta_r + 1 \). By assumption, \( \beta_{r+1} = \beta_r + 1 > n_r + 1 = n_{r+1} \).

(2) \( n_{r+1} \neq n_r + 1 \). By virtue of Lines 9-11, \( n_{r+1} \neq n_r + 1 \) only if \( a_{r+1} \neq a_{n_{r+1}} \).

Consider the following two subcases:

(2-1) \( n_r = 0 \). Then by Lines 6-8, \( n_{r+1} = 0 \). And, clearly, \( 4_{r+1} = 0 \).

This yields \( \beta_{r+1} = \beta_r > n_r = n_{r+1} \).

(2-2) \( n_r \neq 0 \). In view of Line 3, it follows that \( 4_{r+1} \geq 1 \). If \( 4_{r+1} = 1 \), then \( \beta_{r+1} = \beta_r \) and \( n_{r+1} = n_r + 1 \) or \( n_{r+1} = 0 \). By Proposition 5.2.3, \( n_{n_r} < \frac{n_r}{2} \). Thus \( n_{r+1} \leq n_r \). By assumption, \( \beta_{r+1} = \beta_r > n_r \geq n_{r+1} \).

Now, consider the case \( 4_{r+1} > 1 \). From the definition of \( \beta \), we have \( \beta_{r+1} = \beta_r - 4_{r+1} + 1 \). Let \( k = 4_{r+1}, i_1 = n_r \) and \( i_s = n_{i_{s-1}} \) for
2 \leq s \leq k. By Line 3, \( i_s \neq 0 \) and \( a_{r+1} \neq a_{i_s} \) for any \( 1 \leq s \leq k \). By virtue of Lines 7 and 10, \( n_{r+1} = 0 \) or \( n_{r+1} = n_{i_k} + 1 \). Proposition 5.2.3 implies that \( n_{i_s} \leq i_s - 1 \) for \( 1 \leq s \leq k \). By Algorithm 5.1.1, the condition \( k = 4_{r+1} > 1 \) derives that \( n_{r+1} = 0 \) or \( n_{r+1} = n_{i_k} + 1 \). Thus \( n_{r+1} \leq n_r - k + 1 \). By assumption, \( \beta_{r+1} = \beta_r - 4_{r+1} + 1 > n_r - k + 1 \geq n_{r+1} \).

By mathematical induction on \( j \), the assertion holds.

**Proposition 5.3.4** \( \sum_{i=1}^{j} m_i < \sum_{i=1}^{j} 0_i \).

Let \( \gamma_j = \begin{cases} 
1 & \text{if } 4_j \geq 1 \\
0 & \text{if } 4_j = 0 
\end{cases} \). By Proposition 5.3.4, \( \sum_{i=1}^{j} 4_i = \sum_{i=1}^{j} m_i + \sum_{i=1}^{j} \gamma_i < \sum_{i=1}^{j} 0_i + \sum_{i=1}^{j} \gamma_i \). Since \( \sum_{i=1}^{j} \gamma_i + \sum_{i=1}^{j} 0_i = j \), we then have the following result.

**Proposition 5.3.5** \( \sum_{i=1}^{j} 4_i < j \) for each \( j \geq 1 \).

From Proposition 5.3.5, while string \( S \) with length \( \log(S) \) is inputted, the total times of the \( 4_j \) must be less than \( \log(S) \). Actually, if you observe Algorithm 5.1.1, each line of the algorithm executed at most once for each input \( a_j \) except Lines 3–5. This shows the time complexity of Algorithm 5.1.1 is sublinear, i.e., the time complexity of Algorithm 5.1.1 less than or equal to \( \log(S) \). In fact, the worst case, \( \sum_{j=1}^{\log(S)} = \log(S) - 3 \). For example, suppose the input string \( S \) is \( ACAA \cdots A \) with length greater than 4. One can see that \( n_1 = n_2 = 0 \),
$4_1 = 4_2 = 4_3 = 0$ and $4_j = 1$ for $j \geq 4$. That is, while $a_1$, $a_2$, $a_3$ inputted, the Line 4 of the algorithm 5.1.1 is not executed and Line 4 of Algorithm 5.1.1 is executed exactly once for each input $a_j$, $j \geq 4$. Finally, we conclude that the upper bound of the time complexity of Algorithm 5.1.1 is $\lg(S) - 3$ for an input string $S$. 


Chapter 6

Concluding Remarks

This thesis presented focuses on the exact pattern matching problems. In the early work, many algorithms are introduced to solve pattern matching problems. The advantage of the naive algorithm is that it is simple but its time complexity is $O(nm)$. The advantage of the B.M algorithm is that it can avoid unnecessary comparisons and its time complexity is $O(n)$. But when the size of the alphabets is small, the bad character rule does not affect it too much. And it is difficult to be applied to accomplish multiple keystrings matching concurrently. The advantage of the K.M.P algorithm is that its time complexity is $O(n)$. But it can not compare with multiple keystings concurrently. The A.C algorithm improves K.M.P algorithm such that it can compare multiple keystings. But this improved algorithm can not obtain exact outputs if there is an infix relation between keywords.

In this research, we present an algorithm to construct an mc-DFA with
multiple classes of keystrings which processes the text in a single pass and outputs a match signal if any of the keystring occurs in the text. The mc-DFA solves the problem of infix checking too. Moreover, in Chapter 5, we present a pattern matching algorithm which applies the next function to detect the occurrence of a prefix square of an on-line inputted string.

In future work, the mc-DFA is concerned with dealing in the dynamic classes of keystrings. For example, if one wants to insert some keystrings to the set of keystrings or delete any keystrings from the set of keystrings. How can we use the original mc-DFA to deal with the new set of keystrings? It is imperative and useful if we don’t need to reconstruct the mc-DFA for small changes of the set of keystrings. On the other hand, the study of some special substrings and relations is required. The prefix square presented in Chapter 5 is one kind of string and the infix checking presented in Chapter 3 is another. There are many variations of strings which must be detected in applications. For example, the shuffle relation, the palindrome, the primitive, . . . , etc. These are essential topics concerning strings of the pattern matching problems too.
References


[10] Robert, R.J., and Maclis, D., The Restriction Enzyme Database,  

    prefix square detection algorithm, submitted for publication.