

國立交通大學

電信工程學系碩士班

碩士論文

應用於網路安全之正規表示式字樣比對



**Pattern Matching for Regular Expression in Network  
Security**

研究生：吳柏庚

指導教授：李程輝 教授

中華民國九十六年七月

應用於網路安全之正規表示式字樣比對

# Pattern Matching for Regular Expression in Network Security

研究生：吳柏庚

Student: Bo-Geng Wu

指導教授：李程輝 教授

Advisor: Prof. Tsern-Huei Lee

國立交通大學

電信工程學系碩士班



Submitted to Institute of Communication Engineering  
College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Communication Engineering

July 2007

Hsinchu, Taiwan, Republic of China.

中華民國九十六年七月

# 應用於網路安全之正規表示式字樣比對

學生：吳柏庚

指導教授：李程輝 教授

國立交通大學電信工程學系碩士班

## 中文摘要

特徵(signature)比對在防止病毒/蠕蟲(virus/worm)應用中是個重要的技術。已經有許多大家所熟知的字樣比對(pattern matching)演算法，像是 KMP、BM 以及 AC 演算法。AC 演算法將字樣事先處理並建造一個可以同時比對多個字樣的有限狀態機。AC 演算法的另一個優點是它可以在所有環境底下保證其有一定的效能(performance)。因此，AC 演算法在很多系統中被廣泛的使用，特別是當效能為主要訴求時。但是，AC 演算法只能針對一般字樣(byte-string)比對，然而病毒/蠕蟲之特徵可以由簡單的正規表示式(regular expression)呈現。在本篇論文中，我們延展 AC 演算法，使其可以系統性的建構有限狀態字樣比對機，此字樣比對機可以在有限的輸入字串(string)中找出第一個字樣發生(first occurrence)的結束位置(ending position)且字樣可以是一般字樣或是正規表示式字樣。若我們事先使用可疑字樣過濾器(pre-filter)找出可疑字樣之起始位置，我們可以大幅度減低字樣比對機之複雜度。實驗結果顯示出我們所提出之字樣比對機之比對效能，比 ClamAV 的方法好很多。

# Pattern Matching for Regular Expression in Network Security

Student: Bo-Geng Wu

Advisor: Prof. Tsern-Huei Lee

Institute of Communication Engineering  
National Chiao Tung University

## Abstract

Signature matching is considered an important technique in anti-virus/worm applications. There are some well-known pattern matching algorithms such as Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), and Aho-Corasick (AC). The AC algorithm pre-processes the patterns and builds a finite automaton which can match multiple patterns simultaneously. Another advantage of the AC algorithm is that it guarantees deterministic performance under all circumstance. As a consequence, the AC algorithm is widely adopted in various systems, especially when worst-case performance is an important design factor. However, the AC algorithm was developed only for strings while virus/worm signatures could be specified by simple regular expressions. In this thesis, we generalize the AC algorithm to systematically construct a finite state pattern matching machine which can indicate the ending position in a finite input string for the first occurrence of virus/worm signatures that are specified by strings or simple regular expressions. We show that the complexity of the pattern matching machine can be largely reduced if some pre-filter scheme is used to locate the starting positions of suspicious sub-string in the scanned text string. Experimental results show that our proposed pattern matching machine yield much better throughput performance than the ClamAV implementation.

# 誌謝

由衷感謝我的指導教授--李程輝 教授。在我的碩士班兩年求學生涯中，充分的給予我研究上的指導。因為老師您的教誨與指導，使我了解做研究上所需要的態度與方法。在完成碩士論文的過程中，使我成長不少。

感謝網路技術實驗室的謝景融學長、黃迺倫學姐和黃郁文學長在課業以及研究上的指教，也要感謝一起奮鬥的同窗夥伴--嘉旂、建成和登煌，不論是生活上、研究上、課業上都能不吝指教，相互扶持，感謝你們。

感謝實驗室的大家，耀誼、世弘、凱文、勁文、錫堯在我研究上或生活上遇到瓶頸時給予我支持與鼓勵。感謝碩士班兩年生活曾經給予我幫助的所有人，謝謝你們。



最後，我要特別感謝我的父親吳松村先生與母親詹德美女士，感謝您們的栽培與教養，以及無限的關心和鼓勵，每當遭遇挫折，家庭是您們給予我溫暖的避風港。感謝兄長吳柏昇先生、吳柏宗先生，謝謝你們給予我信心，讓我勇於接受任何挑戰。同時，感謝我的女友--李侑臻小姐在我努力做研究時所給予我的一切支持與包容。由衷的感謝大家

謹將此論文獻給所有愛我以及我愛的人。

西元2007年7月 於風城交大

# 目錄

|                                  |           |
|----------------------------------|-----------|
| 中文摘要                             | i         |
| English Abstract                 | ii        |
| 誌謝                               | iii       |
| 目錄                               | iv        |
| 表目錄                              | vi        |
| 圖目錄                              | vii       |
| <b>第一章 簡介</b>                    | <b>1</b>  |
| 1.1 研究背景                         | 1         |
| 1.2 研究動機                         | 2         |
| <b>第二章 相關工作</b>                  | <b>4</b>  |
| 2.1 Aho-Corasick 演算法             | 4         |
| 2.2 可疑字樣過濾器 (pre-filter)         | 6         |
| <b>第三章 入侵偵測系統</b>                | <b>9</b>  |
| 3.1 入侵偵測系統簡介                     | 9         |
| 3.2 ClamAV                       | 10        |
| <b>第四章 應用於網路安全之正規表示式字樣之比對演算法</b> | <b>13</b> |
| 4.1 相關問題與定義                      | 13        |
| 4.1.1 相關問題                       | 13        |
| 4.1.2 相關定義                       | 14        |
| 4.2 應用於網路安全之正規表示式字樣之比對演算法        | 16        |
| 4.2.1 僅有*運算子之正規表示式               | 16        |
| 4.2.2 僅有{min, max}運算子之正規表示式      | 24        |
| 4.2.3 {min, max}以及*運算子之正規表示式     | 26        |

---

|                                    |           |
|------------------------------------|-----------|
| 4.2.4 程式流程圖及演算法                    | 29        |
| <b>第五章 可疑字樣過濾器 (pre-filter)之使用</b> | <b>34</b> |
| 5.1 相關原理                           | 34        |
| 5.2 使用 pre-filter 之範例              | 36        |
| <b>第六章 實驗模擬結果與比較</b>               | <b>40</b> |
| <b>第七章 結論</b>                      | <b>44</b> |
| <b>參考文獻</b>                        | <b>45</b> |



# 表目錄

---

表4-1 正規表示式之特殊字元

14





---

# 圖目錄

---

|      |                               |    |
|------|-------------------------------|----|
| 圖2-1 | 字樣{he, she, his, hers}之AC演算法  | 5  |
| 圖2-2 | 輸入字串 $S=\{ushers\}$ 的範例走法     | 6  |
| 圖2-3 | 找出可疑的字樣起始位置之程序                | 8  |
| 圖3-1 | ClamAV比對動作示意圖                 | 12 |
| 圖4-1 | 字串表示狀態圖                       | 15 |
| 圖4-2 | 狀態 $S$ 之子樹                    | 15 |
| 圖4-3 | 一個 $*$ 運算子之範例                 | 18 |
| 圖4-4 | 兩個 $*$ 運算子之範例                 | 22 |
| 圖4-5 | 一個{min, max}運算子之範例            | 25 |
| 圖4-6 | {min, max}以及 $*$ 運算子之正規表示式範例  | 29 |
| 圖4-7 | 程式流程圖                         | 30 |
| 圖5-1 | 使用pre-filter之範例               | 38 |
| 圖6-1 | 效能比較圖(無感染, 10一般字樣、1正規表示式字樣)   | 41 |
| 圖6-2 | 效能比較圖(無感染, 100一般字樣、10正規表示式字樣) | 42 |
| 圖6-3 | 效能比較圖(有感染, 100一般字樣、10正規表示式字樣) | 42 |

# 第一章

## 簡介

---

### 1.1 研究背景

由於電腦與電腦網路的快速成長與進步，以致於人類處理電腦病毒(virus)與蠕蟲(worm)的速度比不上它們漫延擴展的速度；它們只需要很短暫的時間就可以在網路(Internet)上漫延開來，造成人類社會經濟上龐大的損失。因此，在它們漫延擴展的同時，若能快速且有效率的偵測出來，就能避免有漏洞的系統被它們感染，使得傷害減至最低。



現今的病毒/蠕蟲偵測技術可以區分為三種：(1) 協定(protocol)分析，(2) 異常行為，(3) 病毒字樣(pattern)比對。字樣比對是一種在封包(packet)中尋找所定義字樣的一種技術，病毒/蠕蟲會在被感染的檔案或是電腦中以一段惡意碼的方式呈現，利用字樣比對技術以及這一段惡意碼，我們可以正確地偵測出病毒/蠕蟲。因此，病毒/蠕蟲之惡意碼就必需快速被定義出來，所幸，現今惡意碼的定義技術可以很快地找出新的病毒/蠕蟲之惡意碼。

本篇論文提出一種有限狀態機(finite state machine)建造的程序，此有限狀態機可以用來做字樣比對。許多的字樣比對演算法已經在過去被發表，像是 Knuth-Morris-Pratt (KMP) [4]，Boyer-Moore (BM) [5]和 Aho-Corasick (AC) [6]。KMP 和 BM 演算法只能針對單一字樣比對，而不允許多個字樣同時比對。而 AC 演算法可以針對多個字樣建造一個有限狀態機，可以同時比對所有定義的字樣；

且保證其在任何情況下有一定的效能(performance)。因此，AC 演算法在許多的系統中被廣泛的使用，特別是當效能為主要訴求時。

我們的字樣比對有限狀態機結構可以指出第一個病毒/蠕蟲惡意碼字樣發生的位置，字樣可以是一般字樣(byte-string)或是由正規表示式(regular expression)呈現的字樣。本篇論文中提到的正規表示式之特殊字元(wildcard)定義於 ClamAV(Clam Anti-Virus)中[1][2]，ClamAV 是一個自由軟體且可任意修改的防毒程式。

## 1.2 研究動機

利用 AC 演算法來做字樣比對雖然很有效率，但是它只能針對一般字樣做比對，若病毒/蠕蟲之惡意碼是利用正規表示式的方式呈現，則 AC 演算法無法對其做比對。正規表示式可以被非決定性有限狀態機(non-deterministic finite automata, NFA)所識別，非決定性有限狀態機等效於決定性有限狀態機(deterministic finite automata, DFA)，因此我們可以使用決定性有限狀態機來比對利用正規表示式呈現的字樣。決定性有限狀態機所需要的狀態(state)會隨正規表示式呈現的字樣長度成指數性的增加。一個正規表示式呈現的字樣就需要一個決定性有限狀態機，若有很多的字樣都是利用正規表示式呈現，則比對的程序會耗費許多的時間。我們可以將多個決定性有限狀態機結合成一個，但是，卻仍然沒有解決因為正規表示式造成的大量狀態。本篇論文我們提出了一個不一樣的方法來建造一個決定性有限狀態機，可以同時比對多個由正規表示式呈現的字樣以及一般字樣。

論文剩下的章節介紹如下：第二章介紹 AC 演算法與可疑字樣過濾器

(pre-filter)的想法；第三章介紹一些網路入侵偵測系統，像是 ClamAV；第四章是本篇論文提出之演算法；若利用可疑字樣過濾器找出字樣的嫌疑起始位置，如此可以大量的改進比對的速度，此方法記載於第五章；第六章是實驗模擬結果與比較；第七章是結論。



# 第二章

## 相關工作

### 2.1 Aho-Corasick 演算法

AC 演算法針對多個字樣建造一個有限狀態機，可以同時比對多個字樣；且保證其在任何情況下有一定的效能。因此，AC 演算法在許多的系統中被廣泛的使用，特別是當效能為主要訴求時。

AC 演算法建造一個有限狀態機來做字樣比對，此有限狀態機的動作由 3 個函式(function)來決定：(1) *goto* 函式，(2) *failure* 函式以及(3) *output* 函式。有限狀態機中的每一個狀態都代表一個號碼，其中，狀態 0 代表起始狀態。字串(string)是由一連串的符號所(symbol)組成，當我們輸入一段字串進入有限狀態機時，它會從起始狀態開始走，利用現在的狀態(current state)號碼以及輸入的符號，查詢 *goto* 函式或是 *failure* 函式決定下一步走到那一個狀態。假使走到某一個非起始狀態 *S*，則代表現在所輸入字串的字尾(suffix)是某一個字樣的字首(prefix)。圖 2-1 是一個 AC 演算法的範例，它包含 4 個字樣{he, she, his, hers}。

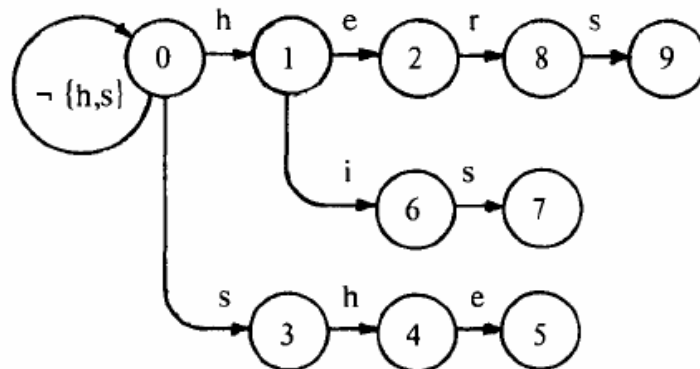


圖2-1(a) goto 函式

|                    |   |   |   |   |   |   |   |   |   |
|--------------------|---|---|---|---|---|---|---|---|---|
| <i>i</i>           | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <b>failure (i)</b> | 0 | 0 | 0 | 1 | 2 | 0 | 3 | 0 | 3 |

圖2-1(b) failure 函式

|          |                   |
|----------|-------------------|
| <i>i</i> | <i>output (i)</i> |
| 2        | {he}              |
| 5        | {she, he}         |
| 7        | {his}             |
| 9        | {hers}            |

圖2-1(c) output 函式

圖 2-1 字樣{he, she, his, hers}之 AC 演算法

在圖 2-1 中，共有 10 個狀態(0,...,9)，*goto* 是個需要兩個參數的函式，一個參數是現在的狀態，另一個則是輸入的符號；回傳會有兩種訊息，當可以成功走到下一個狀態，回傳狀態號碼；反之，回傳失敗訊息。以圖 2-1(a)為例，若現在之狀態為 1，輸入符號為 *i*，回傳  $goto(1,i)=6$ ；若輸入符號非 *e* 或是 *i*，則回傳失敗訊息。

除了起始狀態，每一個狀態都有一個其對應的 *failure* 函式。簡單來說，在有限狀態機中的某一個非起始狀態 *S*，若其代表的字串為 *s*，也許可以在有限狀態機中找一個字串 *s* 最長字尾的字首字串 *t*，其代表的狀態 *T*，就是狀態 *S* 的 *failure* 函式；若找不到最長字尾的字首字串 *t*，則以起始狀態為其 *failure* 函式。以圖 2-1 為例，狀態 7 所代表的字串為 *his*，它可以在圖 2-1(a)中找到最長字尾的字首字串 *h*，也就是狀態 3；換句話說， $failure(7)=3$ 。若有某些字樣在狀態 *A* 被比對到，則我們將這些字樣放入 *output(A)* 中，以圖 2-1 為例，狀態 5 有兩個字樣被比對到，分別是 *he* 以及 *she*，所以  $output(5)={he, she}$ 。

利用AC的三個函式以及有限狀態機，我們可以用來做字樣比對。輸入一串字串 $S=\{c_1c_2\dots c_n\}$ ，其中 $c_i$  ( $1\leq i\leq n$ )代表一個符號，以狀態 0 為起始狀態，查詢 *goto*，若回傳狀態號碼，則以此狀態為現在狀態，並輸入下一個符號；若回傳失敗訊息，則去查詢*failure*函式，以*failure*函式走到的狀態為現在狀態，再重新查詢*goto*直到走得下去為止。若經過某些狀態的*output*函式是非空的，代表有某些字樣被比對到。

舉例來說，以圖 2-1 為例，若輸入字串  $S=\{ushers\}$ ，其有限狀態機的走法如圖 2-2 所示。當現在狀態是 4，輸入符號  $e$  時，因為  $goto(4,e)=5$ ，現在狀態變為 5，因為  $output(5)$ 是非空的，查詢 *output* 函式得知在位置 4 比對到字樣  $\{he, she\}$ 。輸入下一個符號  $r$ ，因為  $goto(5,r)$ 回傳失敗訊息，查詢  $failure(5)=2$ ，以狀態 2 為現在狀態，重新查詢  $goto(2,r)=8$ ，以狀態 8 為現在狀態，繼續輸入剩下的符號，可以發現在位置 6 比對到字樣  $\{hers\}$ 。

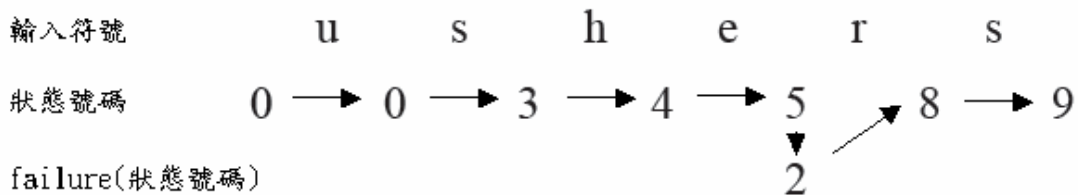


圖 2-2 輸入字串  $S=\{ushers\}$ 的範例走法

## 2.2 可疑字樣過濾器(pre-filter)[8]

使用 AC 演算法做字樣比對，我們可以同時比對多個字樣，且保證其在任何情況下有一定的效能。我們可以知道，若輸入的字串長度為  $n$ ，則狀態轉移(state transition)至多只會有  $2n-1$  次。因此，AC 演算法在許多的系統中被廣泛的使用，

特別是當效能為主要訴求時。但是，隨著高速網路傳輸技術的進步，AC 演算法的效能無法一直維持在網路的速度，若能有個方法可以一次處理多個符號，勢必可大幅增加比對的效能。

可疑字樣過濾器可以在輸入字串中，過濾出有嫌疑的字樣起始位置，只有可疑的位置才需要利用 AC 演算法建造有限狀態機來做比對，如此可以大幅增加字樣比對的效能。

找出嫌疑字樣起始位置的方法，採用[8]所提出的SHIFT表。假使一個符號代表 1-位元組(byte)，所有的字樣都取其前K-位元組，以L-位元組( $L < K$ )為一組做HASH，HASH成M-位元(bit)( $M < 8 * L$ )；若L-位元組的起始位置在 $K-L+1-N$ ，HASH的結果為i，則在SHIFT表中的第i個位置紀錄N值，若SHIFT表中第i個位置已經被填入n且 $N < n$ ，則 $\text{SHIFT}(i) = N$ 。當所有字樣的L-位元組都已經HASH至SHIFT表，假使 $\text{SHIFT}(i)$ 沒有被填入任何值( $0 \leq i \leq 2^M - 1$ )，則 $\text{SHIFT}(i) = K - L + 1$ 。

我們在輸入字串中以一個 K-位元組為搜尋視窗(search window)，將搜尋視窗的最後 L-位元組 HASH，假使 HASH 的結果為 i，查詢  $\text{SHIFT}(i)$  中的值，若是一個非 0 的值 N，則我們將搜尋視窗往後移 N-位元組；若是 0，則表示搜尋視窗的起始位置是某個可疑的字樣起始位置。

以圖 2-3 為例，若比對字樣為{*abcde, fghij*}，輸入字串  $S = \{xxxabcde\}$ ， $K=4$ ， $L=3$ ，搜尋視窗的長度為 4-位元組，一開始對其最後 3-位元組(*xxa*)HASH 並查詢 SHIFT 表，得知搜尋視窗需往後移  $K-L+1=2$ -位元組；對 *abc* 做 HASH 並查詢 SHIFT 表，得知搜尋視窗需往後移 1-位元組；此時搜尋視窗內含{*abcd*}，HASH *bcd* 查詢 SHIFT 表得到 0，此時搜尋視窗的起始字元 *a* 為一字樣之可疑起始位置。



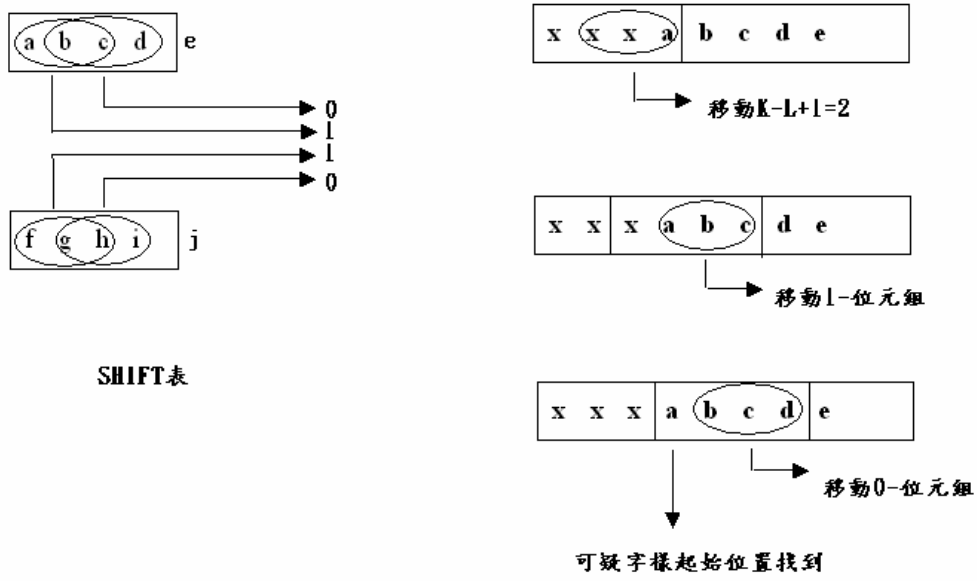


圖 2-3 比對字樣  $\{abcde, fghij\}$ ，輸入字串  $S = \{xxxabcde\}$ ， $K=4$ ， $L=3$ ，找出可疑的字樣起始位置之

程序




## 第三章

# 入侵偵測系統

由於人們對於網際網路應用的倚賴，加以透過網路交易、網路行銷的廠商亦不在少數，因此不論是個人或者企業都可能成為網路攻擊的潛在目標。網路攻擊事件的發生，不僅個人網路安全失去保障，企業更可能因網路駭客的攻擊，損傷企業形象，更可能連帶失去客戶的信任。

### 3.1 入侵偵測系統簡介



入侵偵測系統(Intrusion Detection System, IDS)可以是軟體或是硬體，設計來監測資訊系統或網路系統上可能潛在的惡意破壞活動。入侵偵測的原理簡單的說，是將從資訊系統或網路系統上收集到的活動資訊，與惡意破壞活動特徵(signature)辨識資料庫做比對以判斷是否有惡意或未授權的活動正在進行。依據其應用方法的不同，大致可以分為兩大類：(1)網路型入侵偵測系統(NIDS)，(2)主機型入侵偵測系統(HIDS)。

網路型入侵偵測系統(NIDS)：與網路連結，並分析從網路收集來的封包。從封包取得的資訊與攻擊特徵資訊做比對，如果封包資訊並不符合攻擊特徵資料庫中的攻擊，網路流量便會被判定為正常的流量；相反的，如果封包的資訊符合資料庫中的某項攻擊特徵，反制攻擊的機制便會被啟動。ClamAV就是一個網路型入侵偵測系統軟體。

主機型入侵偵測系統(HIDS)：從主機系統稽核日誌檔案演進而來。傳統的作法是，系統管理人員在每天作業結束前，在日誌檔案中檢查是否有任何可疑的非法行為。這種方式不僅耗時費力，而且無法即時的偵測出潛在的惡意活動。現今的主機型入侵偵測系統在每一台主要的主機上安裝一個代理程式(agent)，執行監控的工作，若有任何系統事件(event)被記錄至日誌檔案中，代理程式便會立即將系統事件與攻擊特徵資料庫做比對，有些主機型入侵偵測系統能夠監控應用程式的日誌檔案，甚至檢查系統的檔案是否遭更改過。

快速增加的安全漏洞：以比對特徵為基礎(signature-based)的安全機制在 1990 年中期成形，當時已知的安全漏洞數目並不多。根據CERT的資料，在 1995 年，只有 171 個安全漏洞被發佈。隨著安全漏洞的數目不斷快速增加，加上變形攻擊(mutations)的出現，上述情形已逐漸改觀。針對每一種不同的攻擊，以比對特徵為基礎的產品都需要一個相對應的攻擊特徵，在 2001 年，每一天至少需增加 6 個攻擊特徵到資料庫中。



以下將簡單的介紹網路型入侵偵測系統軟體--ClamAV，是一個免費而且開放原始碼的防毒軟體。

## 3.2 ClamAV[3]

ClamAV 是個免費而且開放原始碼的防毒軟體，軟體與病毒碼的更新皆由社群免費發佈。目前 ClamAV 主要是使用在由 Linux、FreeBSD 等 Unix-like 系統架設的郵件伺服器上，提供電子郵件的病毒掃描服務。ClamAV 本身是在文字介面下運作，但也有許多圖形介面的前端工具 (GUI front-end) 可用，另外由於其開放原始碼的特性，在 Windows 與 Mac OS X 平台都有其移植版。

ClamAV 病毒定義資料庫包含兩種類型的病毒字樣，(1)由一連串的正常字元(character)所組成的基本病毒字樣，(2)由正規表示式呈現的字樣。由正規表示式呈現的字樣由多個基本病毒字樣所組成，在此我們稱基本病毒字樣為子字樣(sub-pattern)，子字樣之間由特殊字元(wildcard)隔開，要比對此一類型的病毒字樣，子字樣必需依序被比對到。ClamAV 所定義的特殊字元有 4 種，(1)\*：比對任意數量的任意位元組，(2){*min, max*}：比對 *min*~*max* 個任意位元組，(3)?: 比對任意一個位元組，(4)(*a|b*)：比對 *a* 字元或是 *b* 字元。

ClamAV 的實做方法，是以 AC 演算法為基本想法。為了快速查詢輸入的符號會讓有限狀態機走到哪一個狀態，ClamAV 使用一個包含 256 個元素(element)的查詢陣列(array)當作一個節點(node)之樹狀結構，256 個位置分別代表 ASCII 字元所對應的位置。由此可知，ClamAV 的記憶體(memory)使用量由樹狀結構的深度來決定，深度愈深，樹狀結構的節點就愈多，所需要的記憶體量就會愈大。每一個節點大概需要 1,049-位元組(查詢陣列和一些輔助比對的資訊)。

因為 AC 演算法建構有限狀態機會隨著字樣的長度變長而使得樹狀結構的深度變深。ClamAV 的病毒定義資料庫只允許每一個字樣最多耗掉 2K-位元組左右的記憶體，如此看來，若 ClamAV 完全使用 AC 演算法勢必因為記憶體的不足而不可行。

ClamAV 的做法對 AC 演算法做了些微修改，因為 ClamAV 的病毒定義資料庫記憶體的限制，所以它只對每一個字樣的前 2 個位元組建立 AC 演算法之有限狀態機，深度為 2 的我們稱為葉節點(leaf node)並存一個連結清單(linked list)給這一個節點；舉例來說，字樣{*abcde*}，ClamAV 會將{*ab*}建立 AC 演算法有限狀態機，並存連結清單{*cde*}給代表字串{*ab*}的這一個葉節點。當我們在比對時，如

果在有限狀態機中走到了葉節點，ClamAV 會對此葉節點之連結清單內所有字樣的所有位元組做逐一的比對。其動作示意如圖 3-1。

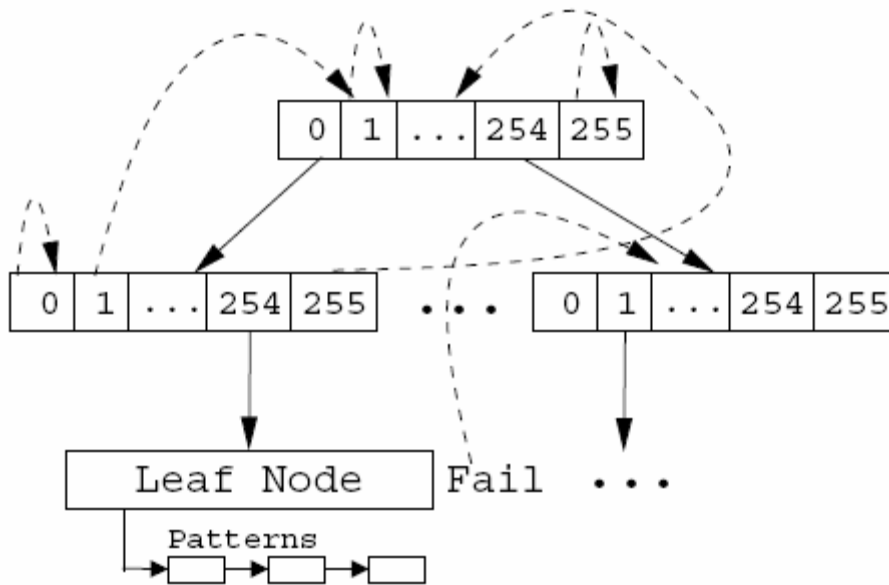


圖 3-1 ClamAV 比對動作示意圖，實線代表 *goto*，虛線代表 *failure*



## 第四章

# 應用於網路安全之正規表示式字樣比對演算法

我們提出的字樣比對有限狀態機結構可以指出第一個病毒/蠕蟲惡意碼字樣發生的位置，字樣可以是一般字樣或是由正規表示式呈現的字樣。本篇論文中提到的正規表示式特殊字元定義於 ClamAV 中，ClamAV 是一個自由軟體且可任意修改的防毒程式。

### 4.1 相關問題與定義

#### 4.1.1 相關問題



在本篇論文中，我們提出了一個問題，若AC演算法建造之有限狀態機可以比對一般字樣所組成的集合 $X$ ，我們如何在原來的 $X$ 中加入 $n$ 個由正規表示式呈現的字樣( $RE_1, RE_2, \dots, RE_n$ )? 本篇論文所提及的正規表示式特殊字元有 3 種，(1)\*：比對任意數量的任意位元組，(2)?: 比對任意一個位元組，(3){ $min, max$ }：比對 $min \sim max$ 個任意位元組，如表 4-1 所示。每一個正規表示式至少包含了一個特殊字元。簡單來說，我們的目的是要建造包含正規表示式之有限狀態字樣比對機(finite state pattern matching machine)，也就是  $X' = X \cup RE_1 \cup RE_2 \cup \dots \cup RE_n$ 。

| 特殊字元                        | 比對行為                              |
|-----------------------------|-----------------------------------|
| *                           | 比對任意數量的任意位元組                      |
| ??                          | 比對任意一個位元組                         |
| { <i>min</i> , <i>max</i> } | 比對 <i>min</i> ~ <i>max</i> 個任意位元組 |

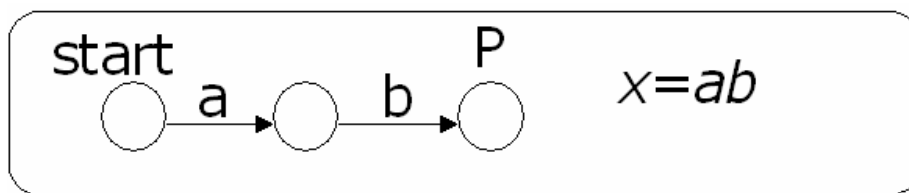
表 4-1 正規表示式之特殊字元

本篇論文提出一個有限狀態字樣比對機的建構程序，當字串  $x$  輸入字樣比對機，它可以在  $x$  中標出第一次某個字樣發生的結束位置(ending position)。假使有一個字樣  $m$  在字串  $x$  中被比對到，若其在  $x$  中之結束位置比其它字樣  $n$  在  $x$  中比對到的結束位置小，如此，我們稱字樣  $m$  在  $x$  中之結束位置為第一次出現字樣之結束位置。一個位置可能同時比對到多個字樣。

假使一般字樣所成之集合  $X$  之 *goto* 狀態圖  $G$  已經經由 AC 演算法建構出來， $G$  的起始狀態我們定義為  $I$ ；倘若  $X$  是一個空集合，則  $G$  只包含了  $I$  這一個狀態，且  $goto(I, a)=I$ ， $a$  代表所有可能的字元。

### 4.1.2 相關定義

在介紹本篇論文所提出的演算法之前，有一些定義必須要在此說明。若有一個字串為  $xy$ ，我們稱  $x$  為字串  $xy$  的字首(prefix)， $y$  為字串  $xy$  的字尾(suffix)；若  $y$  不是一個空字串，我們可以稱  $x$  為字串  $xy$  的完全字首(proper prefix)；同理，若  $x$  不是一個空字串，我們可以稱  $y$  為字串  $xy$  的完全字尾(proper suffix)。在一個 *goto* 狀態圖中，字串  $x$  從起始狀態開始走，若走至狀態  $P$ ，我們稱狀態  $P$  代表字串  $x$ ，如圖 4-1 所示。

圖 4-1 字串表示狀態圖，字串  $x=ab$ ，代表狀態 P

在 *goto* 狀態圖的起始狀態有可能有自我迴圈(self-loop)，若 *goto* 狀態圖存在自我迴圈，我們將自我迴圈移除，它就可以稱為一顆樹(tree)。以下不考慮自我迴圈，假使  $goto(P, a)=R$ ， $a$  為任意的符號，我們可以稱狀態 P 為狀態 R 之父狀態；狀態 R 為狀態 P 之子狀態。若在 *goto* 狀態圖中，從狀態 P 開始走，存在一非空字串  $x$  走至狀態 R，我們稱狀態 R 為狀態 P 之子孫(descendant)狀態。一顆樹包含了狀態 S 以及其所有子孫狀態，我們稱這棵樹為狀態 S 之子樹(sub-tree)，如圖 4-2 所示。若某一狀態之 *output* 函式為非空，我們可以稱這一個狀態為終了狀態(final state)，表示比對到某一字樣。

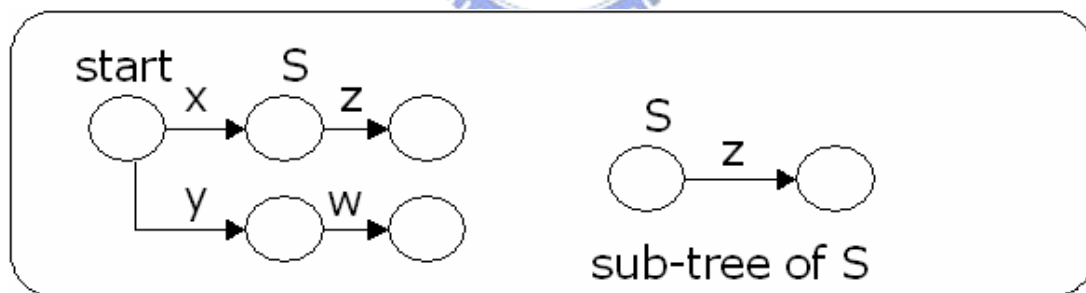


圖 4-2 狀態 S 之子樹

在 *goto* 狀態圖中，若一個狀態  $P$  所代表的字串為  $p$ ，在此 *goto* 狀態圖中所有代表字串  $up$  的狀態  $P'$  我們稱之為狀態  $P$  的同伴狀態(companion state)，其中字串  $u$  為任意字串。基於此點，我們可以說某一狀態是自己的同伴狀態。

本篇論文提出之有限狀態字樣比對機結構可能包含多個分離的 *goto* 狀態



圖，彼此利用 *failure* 函式做連接。

## 4.2 應用於網路安全之正規表示式字樣比對演算法

### 4.2.1 僅有\*運算子之正規表示式

\*運算元用來比對任意數量的任意位元組，我們可以清楚地知道，若\*運算元出現在字樣的第一個或最後一個位置，則它可以被省略。我們先從最簡單的正規表示式開始說明， $x_1 * x_2$ ， $x_1$ 和 $x_2$ 是非空的字串，經由下面的程序說明我們如何建構我們的有限狀態字樣比對機。

1. 複製原來一般字串所建立的 *goto* 狀態圖  $G$ ，稱此一複製圖為  $D$ ，並標示其起始狀態為  $NI$ 。
2. 利用 AC 演算法之 *enter* 函式，在 *goto* 狀態圖  $G$  中加入字串  $x_1x_2$ ，若  $a$  是字串  $x_1x_2$  的第一個字元，則改變原來  $goto(I, a)=I$  的值。字串  $x_1$  所代表的狀態標示為  $Q$ ，我們稱此狀態為換圖狀態(*switch state*)。此加入字串  $x_1x_2$  後的 *goto* 狀態圖稱為  $G'$ 。
3. 在複製之狀態圖  $D$  中加入字串  $x_2$ ，稱加入字串  $x_2$  後之狀態圖為  $D'$ 。若  $a$  是字串的  $x_2$  第一個字元，則改變原來  $goto(NI, a)=NI$  的值。
4. 在  $D'$  中加入字串  $x_1x_2$ ，在此程序新增的狀態稱為虛擬狀態(*virtual state*)，新增字串  $x_1x_2$  後的狀態圖我們稱它為  $D''$ 。所以  $G'$  被  $D''$  所包含，我們可以在  $D''$  找到所有  $G'$  所對應的狀態，以利我們之後更新 *failure* 函式。
5. 獨立計算  $G'$  和  $D''$  每一個狀態之 *failure* 函式以及 *output* 函式。若在  $D''$  狀態圖中的某個狀態  $S'$ ，它的 *failure* 函式是個虛擬狀態，也就是說  $failure(S')=P'$  是個虛擬狀態，則重新計算 *failure* 函式  $P' \leftarrow failure(P')$ ，直到  $P'$  不是一個虛擬狀態為止。

6. 將 goto 狀態圖  $G'$  中代表  $Q$  的狀態及其同伴狀態找出，針對這些狀態的子樹所有狀態更新其 *failure* 函式。若需要更新之狀態為狀態  $S$ ，更新  $failure(S)=failure(S')$ ， $S'$  是狀態  $S$  對應到 goto 狀態圖  $D''$  之狀態。
7. 若在 goto 狀態圖  $G'$  有某一個狀態  $S$  所代表的字串為  $ux_1vx_2$ ，則狀態  $S$  之 *output* 函式必須更新為  $output(S) \leftarrow output(S) \cup \{x_1 * x_2\}$ 。
8. 刪除所有 goto 狀態圖中之虛擬狀態，並使用 goto 狀態圖  $G'$  和  $D''$  來做字樣比對。

以下用一個簡單的範例來說明上面增加比對字樣  $\{x_1 * x_2\}$  的做法，若

$X' = X \cup RE_1$ , where  $X = \{abed edbc, bedab, cedabc\}$ ,  $RE_1 = ab * edb$ ,  $x_1 = ab$ ,  $x_2 = edb$ , 如圖 4-3 所示。

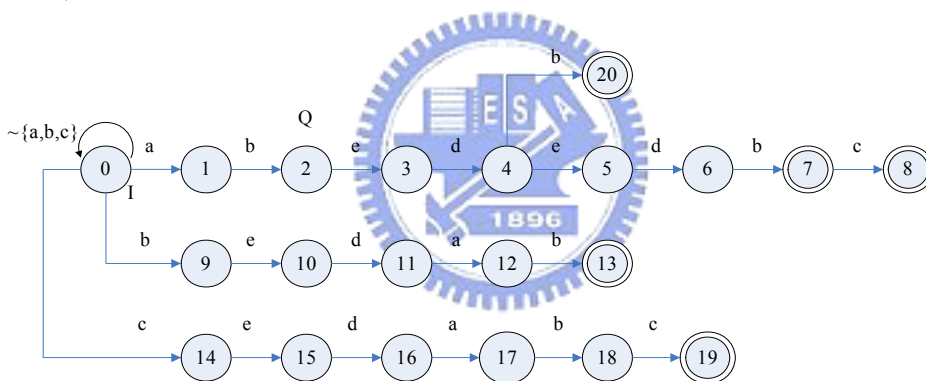


圖 4-3 (a) goto 狀態圖  $G'$

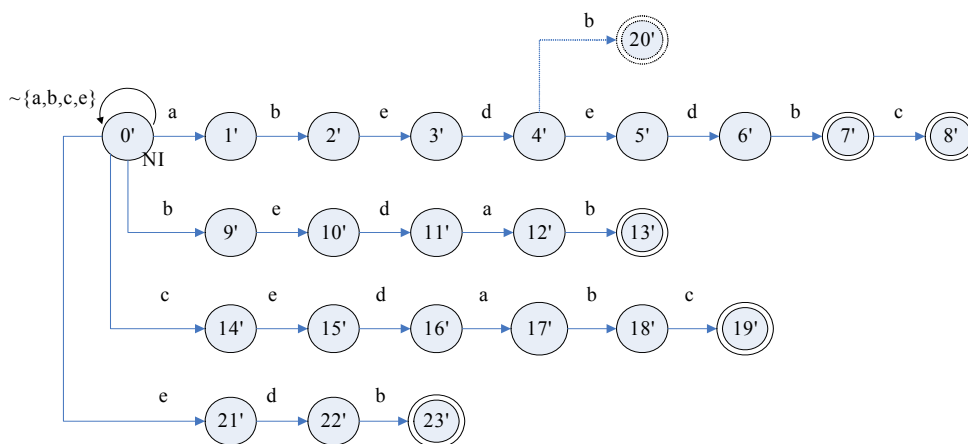


圖 4-3 (b) goto 狀態圖  $D''$

|            |    |    |    |     |     |     |     |     |     |     |    |
|------------|----|----|----|-----|-----|-----|-----|-----|-----|-----|----|
| i          | 0  | 1  | 2  | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10 |
| failure(i) | -- | 0  | 9' | 10' | 11' | 21' | 22' | 23' | 14' | 0   | 0  |
| i          | 11 | 12 | 13 | 14  | 15  | 16  | 17  | 18  | 19  | 20  |    |
| failure(i) | 0  | 1  | 2' | 0   | 0   | 0   | 1   | 2'  | 14' | 23' |    |

圖 4-3 (c) goto 狀態圖  $G'$  之 failure 函式

|             |     |     |     |     |     |     |     |     |     |     |     |     |
|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| i'          | 0'  | 1'  | 2'  | 3'  | 4'  | 5'  | 6'  | 7'  | 8'  | 9'  | 10' | 11' |
| failure(i') | --  | 0'  | 9'  | 10' | 11' | 21' | 22' | 23' | 14' | 0'  | 21' | 22' |
| i'          | 12' | 13' | 14' | 15' | 16' | 17' | 18' | 19' | 20' | 21' | 22' | 23' |
| failure(i') | 1'  | 2'  | 0'  | 21' | 22' | 1'  | 2'  | 14' | 23' | 0'  | 0'  | 9'  |

圖 4-3 (d) goto 狀態圖  $D''$  之 failure 函式

圖 4-3 一個\*運算子範例

圖 4-3(a) 為 goto 狀態圖  $G'$ ，它可以比對  $\overline{X} = X \cup \{abedb\}$ 。圖中  $\sim\{a, b, c\}$  代表不是  $a$ 、 $b$  或是  $c$ ，雙圈狀態表示終了狀態。根據步驟 7 可以得知， $G'$  中之狀態 7 為一個終了狀態，因為狀態 7 所代表的字串為  $x_1edx_2$ ，其中字串  $x_1$  為  $ab$ ，字串  $x_2$  為  $edb$ 。圖 4-3(b) 為 goto 狀態圖  $D''$ ，狀態 20' 是由步驟 4 所新增，所以為一虛擬狀態。虛擬狀態用來輔助 failure 函式計算，在掃描字串前虛擬狀態將被移除。圖 4-3(c)、圖 4-3(d) 分別為  $G'$  和  $D''$  的 failure 函式，在  $G'$  中代表  $Q$  及其同伴狀態的子樹所有狀態均需更新其 failure 函式值(步驟 6)。

字串輸入有限狀態字樣比對機從  $G'$  之起始狀態開始走，若在代表  $Q$  或其同伴狀態之子樹任一個狀態發生失敗訊息，代表  $G'$  已經比對到字串  $x_1$ ，之後跳入  $D'$  中且不會再跳回  $G'$ 。簡單的說， $RE_1$  可以在  $G'$  比對到，假使某一狀態所代表的字串為  $ux_1vx_2$  (圖 4-3(a) 狀態 7、狀態 20)；或是在  $D'$  被比對到，假使某一狀態所代表的字串為  $ux_2$  (圖 4-3(b) 狀態 7'、狀

態 23')。我們可以稱代表 $Q$ 的狀態為換圖狀態，此種狀態是因為 $*$ 運算子而被創造出來。根據以上步驟，有限狀態字樣比對機  $X' = X \cup \{RE_1\}$  被建構出來。

同理，若要加入多個僅包含 $*$ 運算子的正規表示式  $RE_1, \dots, RE_n$ ，建構程序和上面所述類似，假使我們要加入正規表示式字樣  $x_1 * x_2 * x_3$ ，其有限狀態字樣比對機建構程序如下：

1. 複製兩個原來一般字串所建立的 *goto* 狀態圖  $G$ ，稱複製圖為  $D_1$  以及  $D_2$ ，分別標示其起始狀態為  $NI_1$  以及  $NI_2$ 。
2. 利用 AC 演算法之 *enter* 函式，在  $G$  中加入字串  $x_1x_2x_3$ ， $D_1$  加入字串  $x_2x_3$ ， $D_2$  加入字串  $x_3$ ，加完字串的圖分別稱作  $G'$ 、 $D_1'$  以及  $D_2'$ 。代表字串  $x_1$  及  $x_1x_2$  的狀態分別標示為  $Q_1$  及  $Q_2$  (於  $G'$  中)；代表字串  $x_2$  的狀態標示為  $P$  (於  $D_1'$  中)。
3. 在  $D_1'$  中加入字串  $x_1x_2x_3$ ， $D_2'$  加入字串  $x_1x_2x_3$  以及  $x_2x_3$ ，在此程序新增的狀態稱為虛擬狀態，以利 *failure* 函式運算。
4. 獨立計算 *goto* 狀態圖  $G'$ 、 $D_1'$  以及  $D_2'$  之 *failure* 函式。
5. 更新狀態  $Q_2$  和  $P$  以及它們同伴狀態的子樹所有狀態  $S$  的 *failure* 函式， $failure(S)=T$ ， $T$  是狀態  $S$  表示的字串在  $D_2'$  的最長完全字尾之字串  $t$  所代表的狀態。同理，更新狀態  $Q_1$  及其同伴狀態之子樹狀態但非  $Q_2$  及其同伴狀態之子樹狀態的狀態  $S$  的 *failure* 函式， $failure(S)=T$ ， $T$  是狀態  $S$  表示的字串在  $D_1'$  的最長完全字尾之字串  $t$  所代表的狀態。
6. 若在 *goto* 狀態圖  $G'$  有某一個狀態  $S$  所代表的字串為  $ux_1vx_2wx_3$ ，則狀態  $S$  之 *output* 函式必須更新為  $output(S) \leftarrow output(S) \cup \{x_1 * x_2 * x_3\}$ ；同理，若  $D_1'$  有某一狀態  $R$  所代表的字串為  $ux_2vx_3$ ，狀態  $R$  之 *output* 函式必須更新為  $output(R) \leftarrow output(R) \cup \{x_1 * x_2 * x_3\}$ 。

我們稱狀態  $P$  為狀態  $Q_2$  的兄弟狀態 (sibling state)，因為它們由同一個 $*$ 運算子建造出來。假使我們把上面的想法用情況 (condition) 來表示，情況 1 表示錯

誤訊息在狀態 $Q_1$ 或其同伴狀態的子樹某一狀態發生；情況 2 表示錯誤訊息在狀態 $Q_2$ 或 $P$ 或它們同伴狀態的子樹某一狀態發生。我們利用旗標(flag) $FQ_i$ 來區分情況 $i$ 是否成立。也就是說，若情況 $i$ 成立， $FQ_i=1$ ；反之， $FQ_i=0$ 。由此可知， $(FQ_1, FQ_2)=(0,0), (1,0), (1,1)$ 分別代表 $goto$ 狀態圖 $G'$ 、 $D_1'$ 以及 $D_2'$ 。我們稱任意 $(FQ_1, FQ_2)$ 的組合為一個組態(configuration)，並非每一個組態都會發生，舉例來說， $(FQ_1, FQ_2)=(0,1)$ 是不會發生的，因為狀態 $Q_2$ 是屬於狀態 $Q_1$ 的子樹，當失敗訊息發生在狀態 $Q_2$ 或其同伴狀態之子樹任一狀態時，必定也表示發生於狀態 $Q_1$ 或其同伴狀態之子樹任一個狀態。

根據以上之建構程序，我們清楚地知道， $goto$  狀態圖  $G'$  可以比對  $\bar{X} = X \cup \{x_1x_2x_3\} \cup \{ux_1vx_2wx_3\}$ ， $(u,v,w)$  至少有一個不是空字串； $goto$  狀態圖  $D_1'$  可以比對  $\bar{\bar{X}} = X \cup \{x_2x_3\} \cup \{ux_2vx_3\}$ ， $(u,v)$  至少有一個不是空字串； $goto$  狀態圖  $D_2'$  可以比對  $\bar{\bar{\bar{X}}} = X \cup \{ux_3\}$ ， $u$  可以是任意字串。根據以上步驟，有限狀態字樣比對機  $X' = X \cup \{x_1 * x_2 * x_3\}$  被建構出來。

以下我們舉一個例子來說明加入比對字樣  $\{x_1 * x_2 * x_3\}$ ，若  $X' = X \cup \{RE_1\}$ ，其中  $X = \{abedebc, bedad, cedabc\}$ ， $RE_1 = x_1 * x_2 * x_3 = ab * edb * c$ ，如圖 4-4 所示，包含其所有組態的  $goto$  狀態圖及其  $failure$  函式。

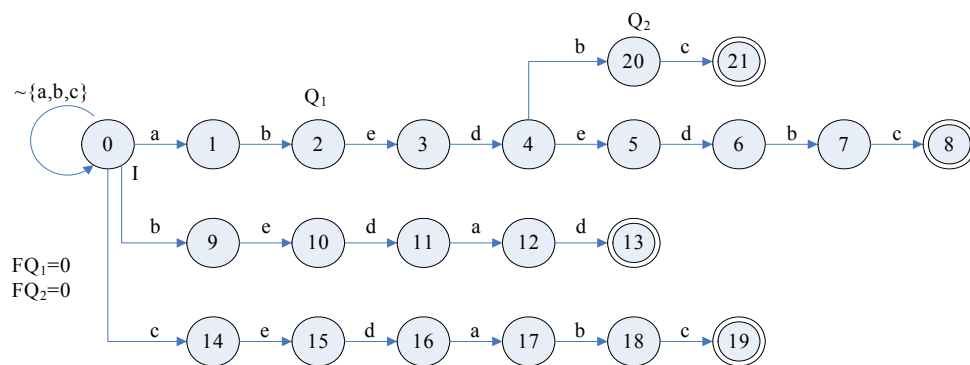


圖 4-4(a)  $goto$  狀態圖  $G'$

|            |    |    |           |           |           |           |           |           |           |           |           |
|------------|----|----|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| i          | 0  | 1  | 2         | 3         | 4         | 5         | 6         | 7         | 8         | 9         | 10        |
| failure(i) | -- | 0  | <u>31</u> | <u>32</u> | <u>33</u> | <u>44</u> | <u>45</u> | <u>46</u> | <u>47</u> | 0         | 0         |
| i          | 11 | 12 | 13        | 14        | 15        | 16        | 17        | 18        | 19        | 20        | 21        |
| failure(i) | 0  | 1  | 0         | 0         | 0         | 0         | 1         | <u>24</u> | <u>36</u> | <u>46</u> | <u>47</u> |

圖 4-4(b)  $G'$  之 failure 函式

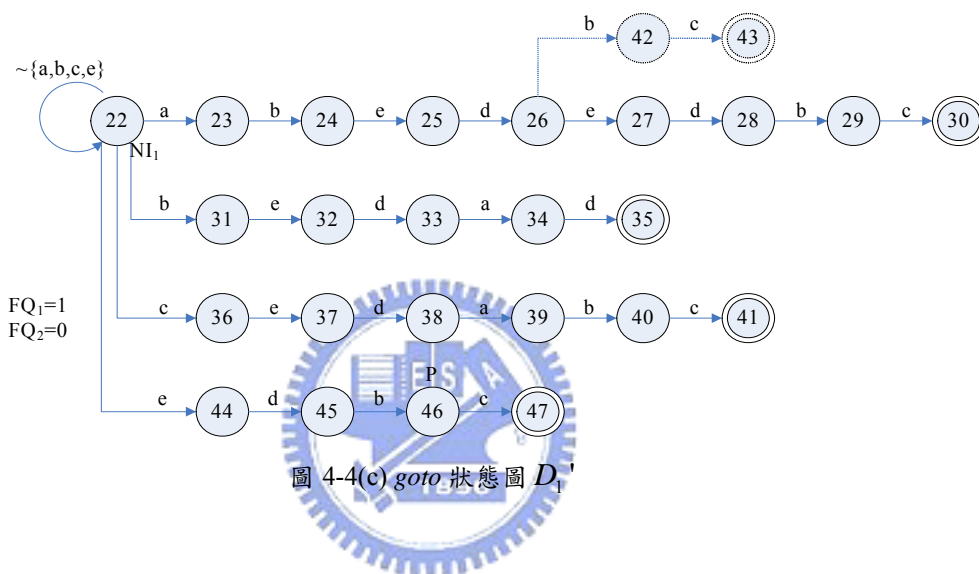


圖 4-4(c) goto 狀態圖  $D_1'$

|            |    |    |    |    |    |    |    |           |           |    |    |           |           |
|------------|----|----|----|----|----|----|----|-----------|-----------|----|----|-----------|-----------|
| i          | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29        | 30        | 31 | 32 | 33        | 34        |
| failure(i) | -- | 22 | 31 | 32 | 33 | 44 | 45 | <u>57</u> | <u>62</u> | 22 | 44 | 45        | 23        |
| i          | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42        | 43        | 44 | 45 | 46        | 47        |
| failure(i) | 22 | 22 | 44 | 45 | 23 | 24 | 36 | <u>57</u> | <u>62</u> | 22 | 22 | <u>57</u> | <u>62</u> |

圖 4-4(d)  $D_1'$  之 failure 函式

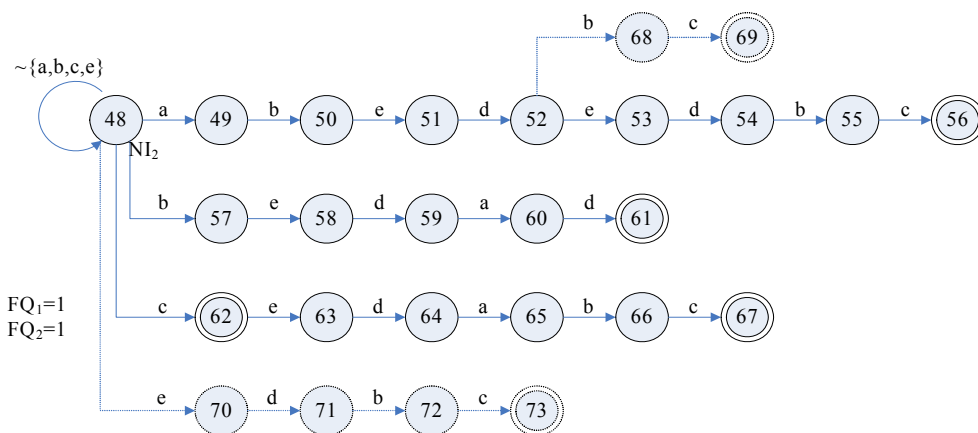


圖 4-4(e) goto 狀態圖  $D_2'$

|            |    |    |    |    |    |    |    |    |    |    |    |    |    |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|
| i          | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| failure(i) | -- | 48 | 57 | 58 | 59 | 48 | 48 | 57 | 62 | 48 | 48 | 48 | 49 |
| i          | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 |
| failure(i) | 48 | 48 | 48 | 48 | 49 | 50 | 62 | 57 | 62 | 48 | 48 | 57 | 62 |

圖 4-4(f)  $D_2'$  之 failure 函式

圖 4-4 兩個\*運算子範例

$D_1'$  中的狀態 46(狀態P)是  $G'$  中狀態 20(狀態 $Q_2$ )的兄弟狀態，因為它們由同一個\*運算子所創造出來。 $G'$  的狀態 8 可以比對到一般字樣  $\{abededbc\}$  以及正規表示式字樣  $\{RE_1\}$ ，因為其表示的字串為  $x_1edx_2x_3$ ；同理， $D_1'$  中的狀態 30，其表示之字串為  $abedx_2x_3$ ，亦可比對字樣  $\{RE_1\}$ 。

若我們有  $n$  個僅有\*運算子的正規表示式字樣， $RE_1, \dots, RE_n$ ，其\*運算子共有  $m$  個，我們把移除所有\*運算子的字串稱做  $SRE$ ，舉例來說，若  $RE_k = a*b*c$ ，則其  $SRE_k = abc$ 。原來的goto狀態圖  $G$  需要加入所有  $SRE_k (1 \leq k \leq n)$ ，加入完後的圖我們稱之為  $G'$ 。 $m$  個\*運算子共有  $m$  個換圖狀態 ( $Q_1 \dots Q_m$ )，若任意兩個換圖狀態  $Q_i, Q_j$  其表示字串是一樣的，則我們可以將兩個換圖狀態視為一樣而合成一

個。所以在  $G'$  中的換圖狀態會有  $M$  個 ( $Q_1 \dots Q_M, M \leq m$ )。

接著我們要決定要加入哪些  $SRE_K (1 \leq K \leq n)$  的字尾給其對應的組態 ( $FQ_1, \dots, FQ_M$ )，若每一個屬於某個  $RE_K$  的旗標  $FQ_i$  均為 0，則在此組態圖  $D$  中加入字串  $SRE_K$ ；若至少有一個屬於  $RE_K$  的旗標  $FQ_i$  為 1，則我們必須找出一個屬於  $RE_K$  且  $FQ_X$  為 1 的旗標狀態  $Q_X$ ，且狀態  $Q_X$  是所有屬於  $RE_K$  且  $FQ_i$  為 1 的旗標狀態之子樹。若狀態  $Q_X$  所代表的字串為  $u$ ， $SRE_K = \overline{uSRE_K}$ ，則在此組態圖  $D$  中加入字串  $\overline{SRE_K}$ ，此一加完字串的圖我們稱為  $D'$ 。

獨立針對每一個組態所對應的狀態圖  $D'$  以及  $G'$  計算其 *failure* 函式，接著更新某些狀態之 *failure* 函式。若某一狀態  $S$  屬於某個換圖狀態  $Q$  的子樹狀態，更新此一狀態之 *failure* 函式  $failure(S) = P, P$  是狀態  $S$  表示的字串在對應的組態 *goto* 狀態圖中的最長完全字尾之字串  $p$  所代表的狀態。

假使有一個正規表示式呈現之字樣為  $RE_K = re_1 * \dots * re_y$ ，其中  $re_i (1 \leq i \leq y)$  是代表其對應的字串。若 *goto* 狀態圖  $D$  加入某一字尾字串  $re_j \dots re_y$ ，則在此圖中某些代表字串  $u_j re_j \dots u_y re_y$  的狀態，其 *output* 函式必須加入此一字樣  $RE_K$ 。若進入此圖表示  $re_1 * \dots * re_{j-1}$  已經被比對到。

比對程序由  $G'$  之起始狀態開始，輸入字串  $x$  若在途中發生錯誤訊息，則會跳入其對應的組態 *goto* 狀態圖。若途中發現比對到某一字樣或是輸入字串終了則比對結束。



## 4.2.2 僅有 $\{min, max\}$ 運算子之正規表示式

$\{min, max\}$  運算子用來比對  $min \sim max$  個任意位元組，我們先從最簡單的正規表示式開始說明， $RE_1 = x_1 \{min, max\} x_2$ ， $x_1$  和  $x_2$  是非空的字串，經由下面所述說明我們如何建構我們的有限狀態字樣比對機。

首先將原來一般字串所建立的 *goto* 狀態圖  $G$  加入字串  $x_1$ ，並將  $\{min, max\}$  的訊息加入字串  $x_1$  所代表的狀態並標示其為  $\{min, max\}$ ，此一加完字串的圖稱為  $G'$ 。當我們走到標示  $\{min, max\}$  的狀態，表示字串  $x_1$  已經被比對到，此時指標 (pointer) 應該指到剩餘字樣的起始狀態，也就是  $\{min, max\} x_2$ ，當然， $\{min, max\}$  訊息也會被存在字串  $x_1$  所代表的狀態之所有同伴狀態。

假使走到一個標示為  $\{min, max\}$  的狀態或其同伴狀態，原來的 *goto* 狀態圖  $G'$  會繼續走，但會有一個分歧 (forked) 指標指向  $RE_1$  剩餘字樣的起始狀態並檢查是否可以比對到  $RE_1$ 。分歧指標的 *goto* 狀態圖  $F$  僅加入字串  $x_2$ 。定義一個變數-門檻值 (threshold)  $th = max - min$ ，進入  $F$  會先省略接下來輸入的  $min$  個位元組，因為接下來的  $min$  個位元組與比對無關。假設  $x_2 = a_1 \dots a_n$ ，*goto* 狀態圖  $F$  會有  $n+1$  個狀態 (包含起始狀態)，表示字串  $a_1 \dots a_i$  的狀態編上號碼  $i$ ，所以  $F$  共有狀態 0 到狀態  $n$ 。再定義一個變數  $ctr$  的起始值定為 0，若在 *goto* 狀態圖  $F$  之狀態  $i$  發生錯誤訊息，更新  $ctr = ctr + i - failure(i)$ 。若在起始狀態輸入字元  $a$ ，仍走回起始狀態，則將  $ctr$  加 1。每當  $ctr$  被更新時，檢查  $ctr$  是否大於  $th$ ，若是，則表示此條分歧指標已經不符合此  $\{min, max\}$  運算子之規則，此條分歧指標失效；若非，則繼續在  $F$  圖中走。也許會產生多個分歧指標，因為原來的 *goto* 狀態圖  $G'$  仍然在走。

以下用一個簡單的範例來說明上面增加比對字樣  $RE_1 = x_1 \{min, max\} x_2$  的做

法，若  $X' = X \cup RE_1$ , where  $X = \{abca, babd, cbabc\}$ ,  $RE_1 = ab\{3,7\}dadadb$ ， $x_1 = ab$ ， $x_2 = dadadb$ ， $\{min, max\} = \{3, 7\}$ ，如圖 4-5 所示。

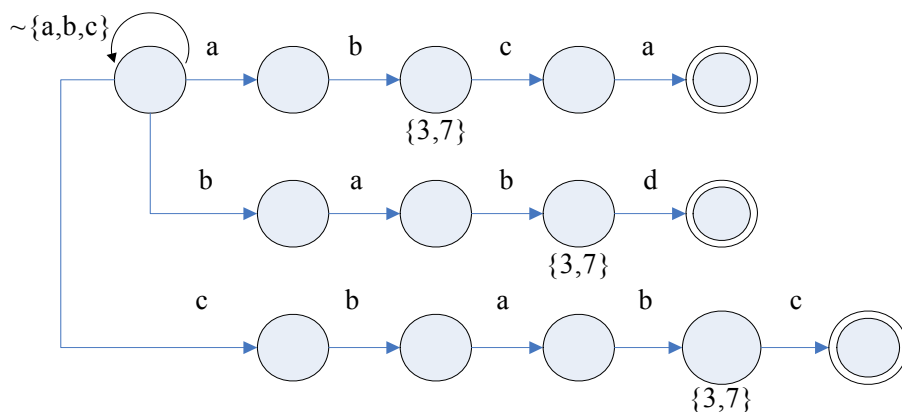


圖 4-5(a) goto 狀態圖  $G'$

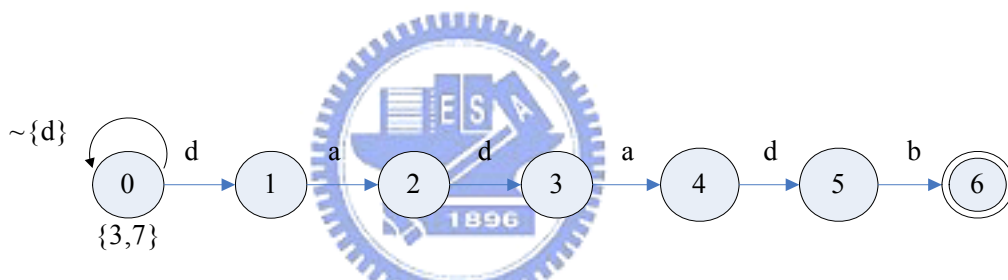


圖 4-5(b) goto 狀態圖 F

|            |    |   |   |   |   |   |   |
|------------|----|---|---|---|---|---|---|
| i          | 0  | 1 | 2 | 3 | 4 | 5 | 6 |
| failure(i) | -- | 0 | 0 | 1 | 2 | 3 | 0 |

圖 4-5(c) goto 狀態圖 F 之 failure 函式

圖 4-5 一個  $\{min, max\}$  運算子範例

上面範例顯示，goto 狀態圖 F 由字串  $\{dadadb\}$  建構出來，如圖 4-5(b)。在圖 4-5(a)中共有 3 個狀態標示  $\{3,7\}$ ，它們是代表字串  $x_1$  的狀態及其同伴狀態。當我們輸入字串  $x = cabccadadadbcdab$ ，我們可以在第 12 個位置比對到字樣  $RE_1$ ，其中第 4 個位置開始的 3(min)個字元  $cca$  在第一條分歧指標被省略。

### 4.2.3 $\{min, max\}$ 以及\*運算子之正規表示式

若我們要加入正規表示式字樣 $RE_1, \dots, RE_n$ 給有限狀態字樣比對機，假使 $RE_1, \dots, RE_n$ 含有\*運算子或 $\{min, max\}$ 運算子，我們該如何建構我們的有限狀態字樣比對機呢？我們必須先將含有 $\{min, max\}$ 運算子之字樣以 $\{min, max\}$ 運算子做分段，舉例來說，正規表示式 $RE = x_1 * x_2 * x_3 \{min_1, max_1\} x_4 * x_5 \{min_2, max_2\} x_6$ 會被分段成 $re_1 = x_1 * x_2 * x_3$ 、 $re_2 = x_4 * x_5$ 以及 $re_3 = x_6$ 。假使由一般字樣所成之集合 $X$ 建造的goto狀態圖 $G$ 已知，我們找出所有不包含 $\{min, max\}$ 運算子的正規表示式字樣以及至少含有一個 $\{min, max\}$ 運算子的正規表示式字樣第一段 $re_1$ ，將這些字樣以及第一段的字樣加入我們的有限狀態字樣比對機，其建構程序如同之前所介紹，唯一的不同點是，當加入含有至少一個 $\{min, max\}$ 運算子的正規表示式字樣第一段 $re_1$ 時，我們不需創造終了狀態，加完的goto狀態圖我們稱之為 $G'$ 。

含有至少一個 $\{min, max\}$ 運算子的正規表示式字樣的剩餘分段將依序被加入有限狀態字樣比對機。以 $RE = x_1 * x_2 * x_3 \{min_1, max_1\} x_4 * x_5 \{min_2, max_2\} x_6$ 為例，因為 $re_1$ 包含兩個\*運算子，所以它給予了 $G'$ 兩個換圖狀態 $Q_1, Q_2$ ， $\{min_1, max_1\}$ 的訊息會被存在代表字串 $ux_1vx_2wx_3$ 的狀態，若走到此一狀態或其同伴狀態則會有分歧指標指到第2段( $re_2$ )的起始狀態。同理，若有一個組態 $FQ_1=1, FQ_2=0$ ，則 $\{min_1, max_1\}$ 的訊息會被存在代表字串 $ux_2vx_3$ 的狀態；若有一個組態 $FQ_1=FQ_2=1$ ，則 $\{min_1, max_1\}$ 的訊息會被存在代表字串 $ux_3$ 的狀態。總而言之， $\{min_1, max_1\}$ 的訊息會被存在比對到 $re_1$ 的狀態。

若含有至少一個 $\{min, max\}$ 運算子的正規表示式的第一段 $re_1$ 有 $L$ 個\*運算子，也就是說 $re_1 = x_1 * x_2 * \dots * x_{L+1}$ ，它將會創造 $L$ 個換圖狀態給 $G'(Q_1, \dots, Q_L)$ ，某些可以比對到 $re_1$ 的狀態我們必須存 $\{min_1, max_1\}$ 的訊息給它，以利它之後分歧指

標指到  $re_2$  的起始位置。我們可以知道， $Q_i$  必定為  $Q_{i-1}$  之子樹狀態 ( $1 \leq i \leq L$ )，若要決定在某個組態中代表什麼樣字串的狀態需要存入  $\{min_1, max_1\}$  的訊息，我們可以在此組態中找出  $FQ_i=1$  且  $i$  為最大的換圖狀態  $Q_i$ ，則在此圖所有代表字串  $u_{i+1}x_{i+1}u_{i+2}x_{i+2}\dots u_{L+1}x_{L+1}$  的狀態需要存入  $\{min_1, max_1\}$  的訊息。

接著我們來討論如何加入  $re_2$  於有限狀態字樣比對機，簡單的說就是將  $re_2$  加入只有空字串的 *goto* 狀態圖。以  $re_2 = x_4 * x_5$  為例，*goto* 狀態圖  $F_1$  只包含了字串  $\{x_4, x_5\}$ ，其中代表字串  $x_4$  的狀態為一個換圖狀態  $P$ ；*goto* 狀態圖  $F_2$  只包含了字串  $\{x_5\}$ ， $F_1$  和  $F_2$  的 *failure* 函式先個別獨立運算，更新 *goto* 狀態圖  $F_1$  中換圖狀態  $P$  的子樹所有狀態之 *failure* 函式。*goto* 狀態圖  $F_1$  中代表字串  $x_4x_5$  的狀態以及 *goto* 狀態圖  $F_2$  中代表字串  $x_5$  的狀態，必須存入  $\{min_2, max_2\}$  的訊息，以利之後分歧指標指到  $re_3$  的起始位置。



當現在狀態為 *goto* 狀態圖  $F_1$  之起始狀態，若輸入符號  $a$ ，仍走回  $F_1$  之起始狀態，計數  $ctr_1$  加 1，假設在某個狀態  $S$  發生錯誤訊息，若狀態  $S$  不是換圖狀態  $P$  之子樹狀態，則計數  $ctr_1$  更新為  $ctr_1 = ctr_1 + |u| - |v|$ ， $u$  為代表狀態  $S$  的字串， $v$  是代表狀態 *failure*( $S$ ) 之字串，若  $a$  是一字串， $|a|$  是字串  $a$  的字串長度。若狀態  $S$  是換圖狀態  $P$  之子樹狀態，則分歧指標會指入 *goto* 狀態圖  $F_2$  中且  $ctr_1$  不再需要。

若第二段  $re_2$  共包含了  $L$  個 \* 運算子，也就是  $re_2 = x_1 * x_2 * \dots * x_{L+1}$ ，則共需  $L+1$  個 *goto* 狀態圖  $F_i$  ( $1 \leq i \leq L+1$ )，每個 *goto* 狀態圖  $F_i$  加入字串  $x_i x_{i+1} \dots x_{L+1}$ ，且有  $L+1-i$  個換圖狀態 ( $Q_1, Q_2, \dots, Q_{L+1-i}$ )。當我們在 *goto* 狀態圖  $F_1$  走時，若現在狀態是 *goto* 狀態圖  $F_1$  之起始狀態，輸入符號  $a$  後仍走回起始狀態，計數  $ctr_1$  加 1，若在狀態  $S$  發生錯誤訊息且狀態  $S$  不是換圖狀態  $Q$  之子樹狀態，則計數  $ctr_1$  更新為  $ctr_1 = ctr_1 + |u| - |v|$ ， $u$  為代表狀態  $S$  的字串， $v$  是代表狀態 *failure*( $S$ ) 之字串。若狀態  $S$  是換圖狀態  $Q_j$  之子樹狀態但非  $Q_{j+1}$  之子樹狀態，則分歧指標會指入 *goto* 狀態圖  $F_j$  中且  $ctr_1$  不再需要。

每當 $ctr_1$ 被更新時，檢查 $ctr_1$ 是否大於 $th$ ，若是，則表示此條分歧指標已經不符合此 $\{min, max\}$ 運算子之規則，此條分歧指標失效；若非，則此條分歧指標仍然有效。

同理，以正規表示式字樣 $RE = x_1 * x_2 * x_3 \{min_1, max_1\} x_4 * x_5 \{min_2, max_2\} x_6$ 為例，第三段 $re_3 = x_6$ 只需加入字串 $x_6$ 在一個空字串的 goto 狀態圖，且代表字串 $x_6$ 的狀態為終了狀態。假使第三段 $re_3$ 包含多個\*運算子，其建構程序和上述一樣，只是多了創造終了狀態的程序。

以下用一個範例來說明增加比對字樣 $RE_1 = ab*edb$ 以及 $RE_2 = abe\{3,5\}d*ca\{2,6\}bd$ 的做法，若 $X' = X \cup RE_1 \cup RE_2$ , where  $X = \{abedebc, bedab, cedabc\}$ ，如圖 4-6 所示。

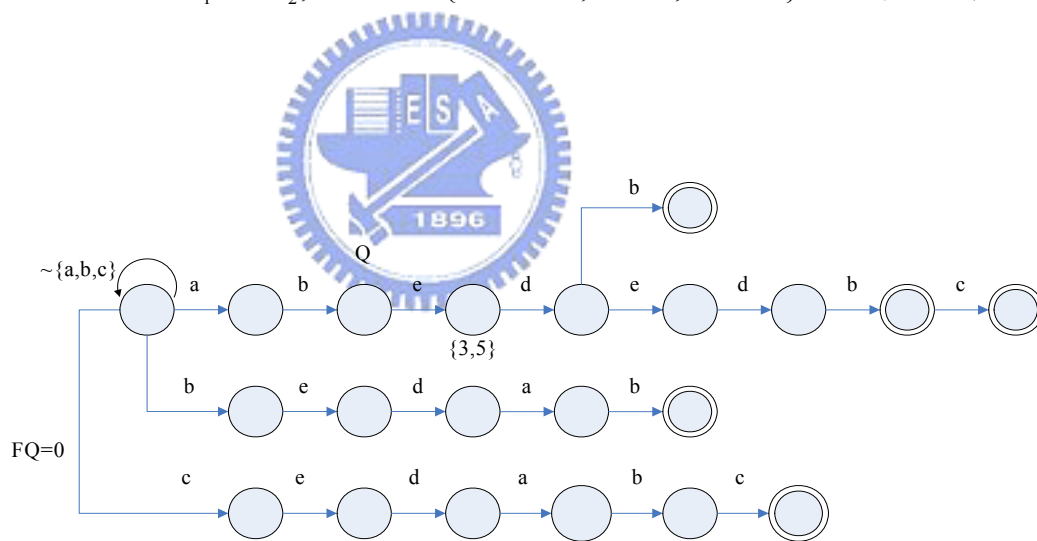


圖 4-6(a) goto 狀態圖  $G'$

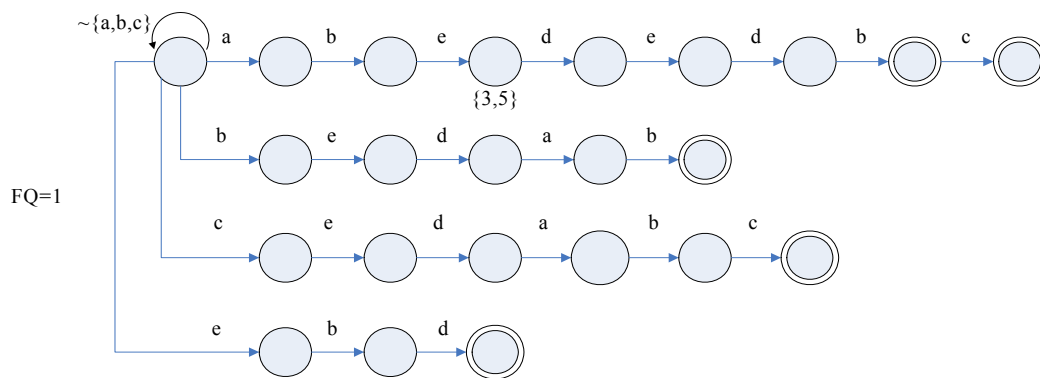


圖 4-6(b) goto 狀態圖  $D'$

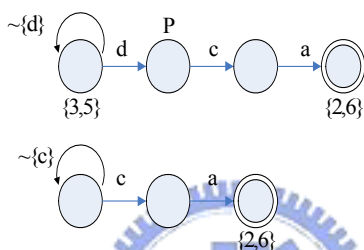


圖 4-6(c)  $RE_2$  第二段  $re_2$  之 goto 狀態圖

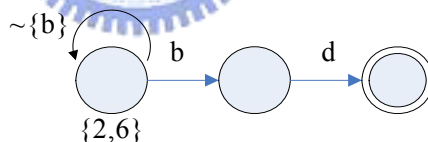


圖 4-6(d)  $RE_2$  第三段  $re_3$  之 goto 狀態圖

圖 4-6 {min, max} 以及 \* 運算子之正規表示式範例

## 4.2.4 程式流程圖及演算法

基於以上想法，若一個組態以一個 goto 狀態圖編號表示，我們可以將程式流程圖畫出，如圖 4-7 所示。

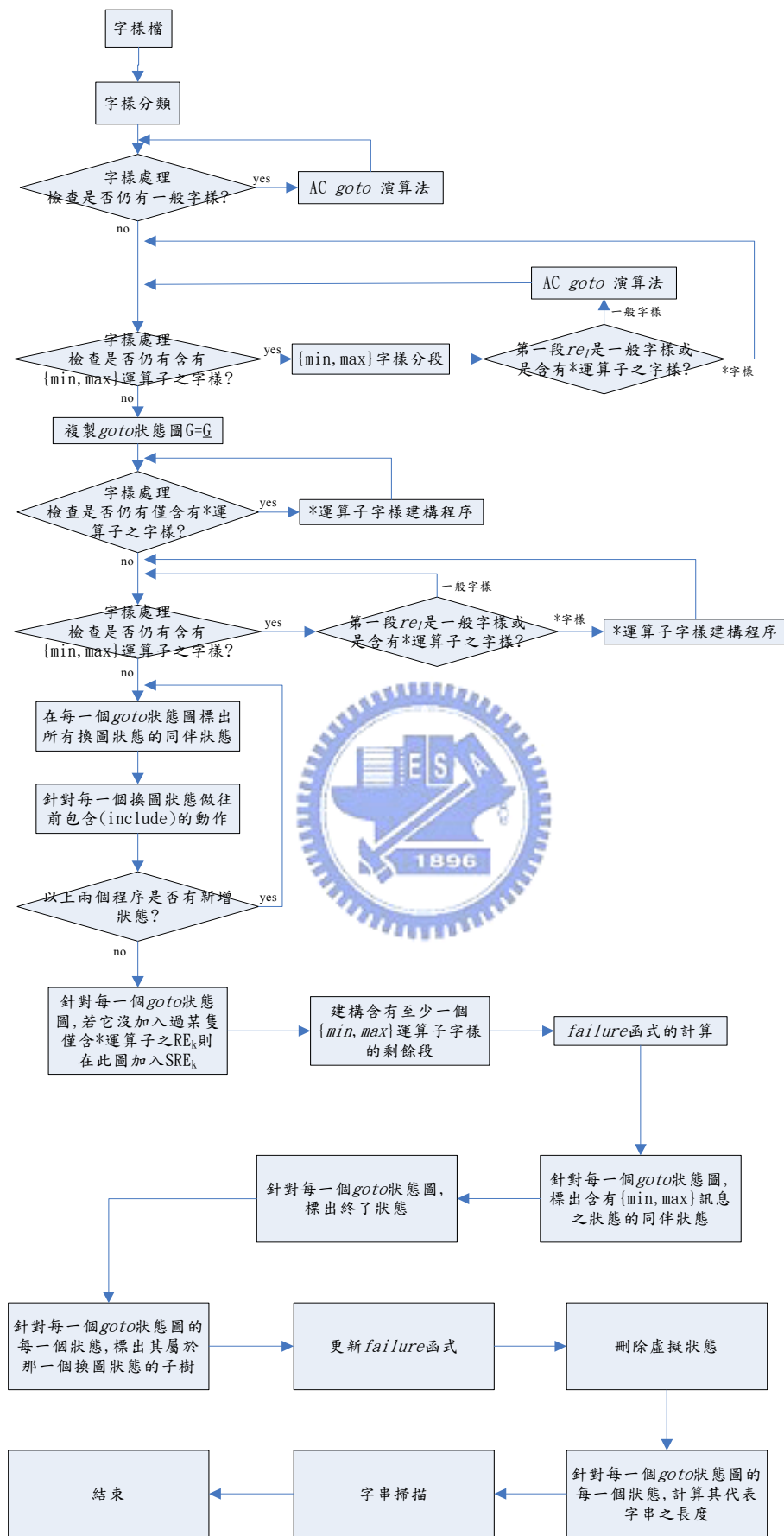


圖 4-7 程式流程圖

以下為字串掃描之演算法:

- **Scan algorithm**
- **Input** A text string  $x=a_1a_2\dots a_n$ , where  $a_i$  is an input symbol and a pattern matching machine  $M$  with many
  - goto graph  $goto$ , failure function  $failure$ , output function  $output$  and  $\{min, max\}$  traverse function  $M\_traverse$
- **Output** location at which the first pattern occur in  $x$
- **Method**
- **Begin**
- $Finish \leftarrow 0, id \leftarrow 0, state \leftarrow 0$
- **while**  $finish=0$  &  $x$  have not exhausted **do**
- **while**  $goto(id, state, a_i) \neq fail$  **do**
- $reg\_state \leftarrow state, state \leftarrow failure(id, state, 0), id \leftarrow failure(id, reg\_state, 1)$
- $State \leftarrow goto(id, state, a_i)$
- **if**  $output(id, state) \neq empty$  **then**
- $print\ output(id, state)$
- $finish = 1$
- **while**  $M\_queue.no \neq empty$  &  $finish=0$  **do**
- $M\_traverse(M\_queue.no, a_i)$
- **if**  $M\_queue.no.output=1$
- $print\ output(M\_queue.no.id, M\_queue.no.state)$
- $finish = 1$
- **if** the state is a state with  $\{min, max\}$  information &  $finish=0$  **then**
- $M\_queue.no.gra \leftarrow \{min, max\} graph\ id$
- $M\_queue.no.id \leftarrow \{min, max\} graph\ id$
- $M\_queue.no.state \leftarrow 0$
- $M\_queue.no.min \leftarrow min$
- $M\_queue.no.th \leftarrow max-min$
- $M\_queue.no.ctr \leftarrow 0$
- $M\_queue.no.no \leftarrow 0$
- $M\_queue.no.output \leftarrow 0$
- $a_i \leftarrow next\ character\ of\ x$
- **End**

註： $failure(id, state, 0)$ 為第  $id$  個圖的  $state$  狀態  $failure$  到的那一個狀態；

$failure(id, state, 1)$ 為第  $id$  個圖的  $state$  狀態  $failure$  到的那一個圖。



以下為分歧指標字串掃描(有遇到含有  $\{min, max\}$  資訊的狀態)演算法  $M\_traverse$ :

```

■ M_traverse algorithm
■ Input      M_queue.no and the input character a;
■ Output    location at which the first pattern occur in x
■ Method
■ Begin
■      M_graph ← M_queue.no.gra
■      M_id ← M_queue.no.id
■      M_state ← M_queue.no.state
■      min ← M_queue.no.min
■      th ← M_queue.no.th
■      ctr ← M_queue.no.ctr
■      no ← M_queue.no.no
■      out ← M_queue.no.out
■      no ← no+1
■      if no > min then
■          if M_graph = M_id then
■              while goto(M_id, M_state, a) = fail do
■                  if fail into the original graph then
■                      Ctr ← ctr + |M_state| - failure(M_id, M_state, 0)
■                      reg_state ← M_state
■                      M_state ← failure(M_id, M_state, 0)
■                      M_id ← failure(M_id, reg_state, 1)
■                  else
■                      reg_state ← M_state
■                      M_state ← failure(M_id, M_state, 0)
■                      M_id ← failure(M_id, reg_state, 1)
■                  if ctr > th then
■                      M_queue.no is not available
■                  else
■                      M_state ← goto(M_id, M_state, a;)
■          else
■              while goto(M_id, M_state, a) = fail do
■                  reg_state ← M_state
■                  M_state ← failure(M_id, M_state, 0)
■                  M_id ← failure(M_id, reg_state, 1)
■                  M_state ← goto(M_id, M_state, a;)
■      M_queue.no.id ← M_id

```

- M\_queue.no.state ← M\_state
- M\_queue.no.ctr ← ctr
- if *output*(M\_id, M\_state) ≠ empty then
- M\_queue.no.output ← 1
- if the state is a state with {min, max} information then
- put some information into M\_queue.no
- End



## 第五章

# 可疑字樣過濾器(pre-filter)之使用

### 5.1 相關原理

上一章節所提的結構適用於當所掃描字串尚未知道可疑字樣起始位置時使用。在防毒的應用中，可以使用簡單的可疑字樣過濾器來找出可疑字樣的位置，以下我們稱這一種可疑字樣過濾器為 pre-filter。我們使用 2.2 節所介紹的方法來找出字樣在輸入字串  $x$  中之可疑起始位置。

找出嫌疑字樣起始位置的方法，採用[8]所提出的SHIFT表。假使一個符號代表 1-位元組，所有的字樣都取其前  $K$ -位元組，正規表示式若由  $B$  個 \* 運算子或  $\{min, max\}$  運算子所構成，則此正規表示式共產生  $B+1$  個字樣，舉例來說，若  $RE = x_1 * x_2 \{2,8\} x_3 * x_4$ ， $RE$  產生了 4 個字樣， $x_1$ 、 $x_2$ 、 $x_3$  以及  $x_4$ 。每個字樣以  $L$ -位元組 ( $L < K$ ) 為一組做 HASH，HASH 成  $M$ -位元 (bit) ( $M < 8 * L$ )；若  $L$ -位元組的起始位置在  $K-L+1-N$ ，HASH 的結果為  $i$ ，則在 SHIFT 表中的第  $i$  個位置紀錄  $N$  值，也就是  $SHIFT(i) = N$ ；若 SHIFT 表中第  $i$  個位置已經被填入  $n$  且  $N < n$ ，則  $SHIFT(i) = N$ 。當所有字樣的  $L$ -位元組都已經 HASH 至 SHIFT 表，假使  $SHIFT(i)$  沒有被填入任何值 ( $0 \leq i \leq 2^M - 1$ )，則  $SHIFT(i) = K - L + 1$ 。

我們在輸入字串中以一個  $K$ -位元組為搜尋視窗，將搜尋視窗的最後  $L$ -位元組 HASH，假使 HASH 的結果為  $i$ ，查詢  $SHIFT(i)$  中的值，若是一個非 0 的值  $N$ ，

則我們將搜尋視窗往後移  $N$ -位元組；若是  $0$ ，則表示搜尋視窗的起始位置是某個可疑的字樣起始位置。將這一個可疑位置輸入驗證器(verification engine)中做驗證。驗證是否可以比對到字樣，若比對到字樣，則比對程序結束；若沒有比對到任一個字樣，則將搜尋視窗往後移一個位元組繼續以上程序。本章節我們將說明透過 pre-filter 後如何建構我們的有限狀態字樣比對機。

若由一般字樣所成的集合  $X$  之 goto 狀態圖為  $G$ ，加入  $n$  個正規表示式字樣  $RE_1, \dots, RE_n$ ，先將僅包含 \* 運算子之正規表示式字樣  $RE_k$  以及至少包含一個  $\{min, max\}$  運算子之字樣第一段  $re_{L1}$  找出，針對這一些  $RE_k$  與  $re_{L1}$  於  $G$  中加入  $SRE_k$  與  $sre_{L1}$ ，並將自我迴圈移除，稱此圖為  $G'$ 。至少包含一個  $\{min, max\}$  運算子之字樣剩餘段之 goto 狀態圖建法與上一章節相同，在此不再贅述。若某一個 goto 狀態圖的起始狀態為某一個至少包含一個  $\{min, max\}$  運算子之字樣剩餘段的起始位置，我們可以稱這種 goto 狀態圖為  $\{min, max\}$  圖。舉例來說，圖 4-6 (d) 就是一個  $\{min, max\}$  圖。



透過 pre-filter 濾出可疑字樣位置後輸入驗證機做比對，驗證機的結構和上一章節所述之有限狀態字樣比對機類似。在此介紹一種特殊狀態 'END'。上一章節所介紹之有限狀態字樣比對機與此一章節介紹的驗證機主要差別在於驗證機之 goto 狀態圖  $G'$  以及非  $\{min, max\}$  圖之起始狀態沒有自我迴圈，且這些起始狀態  $S$  的 failure 函式  $failure(S)=END$ 。

接著我們討論其 failure 函式之計算，每一個 goto 狀態圖的 failure 函式先個別獨立計算，之後更新 failure 函式(方法和上一章節相同)。若某非  $\{min, max\}$  圖  $F$  之某一狀態  $S$  之 failure 函式  $failure(S)=P$  仍在原來的圖  $F$ ，將此狀態之 failure 函式改為  $failure(S)=END$ ；若狀態  $S$  之 failure 函式  $failure(S)=P$  跳到另一張圖  $F'$ ，則狀態  $P$  所代表的字串為狀態  $S$  所代表的字串在圖  $F'$  中之最長完全字尾。

當使用 pre-filter 找到字樣可疑位置，我們必須從  $G'$  以及之前曾經進入過的非  $\{min, max\}$  圖之起始狀態開始走。當我們在  $G'$  中走時，所有的字樣都有可能被比對到，且有可能在走的過程遇到存有  $\{min, max\}$  訊息之狀態產生分歧指標。當有字樣被比對到或是輸入字串終了，字樣比對即結束。當錯誤訊息遇到 END，此一指標也宣告失效。所以我們可以清楚地知道，一般字樣所成之集合  $X$  只有可能在 goto 狀態圖  $G'$  被比對到。

## 5.2 使用 pre-filter 之範例

以下用一個範例來說明使用 pre-filter 時的驗證機之做法：

若  $X' = X \cup RE_1 \cup RE_2$ , where  $X = \{abededzc, bedab, cedabc\}$ ,  $RE_1 = ab*edb$ ,

$RE_2 = abe*dc\{3,5\}d*cx\{2,6\}bd$ ，如圖 5-1 所示。

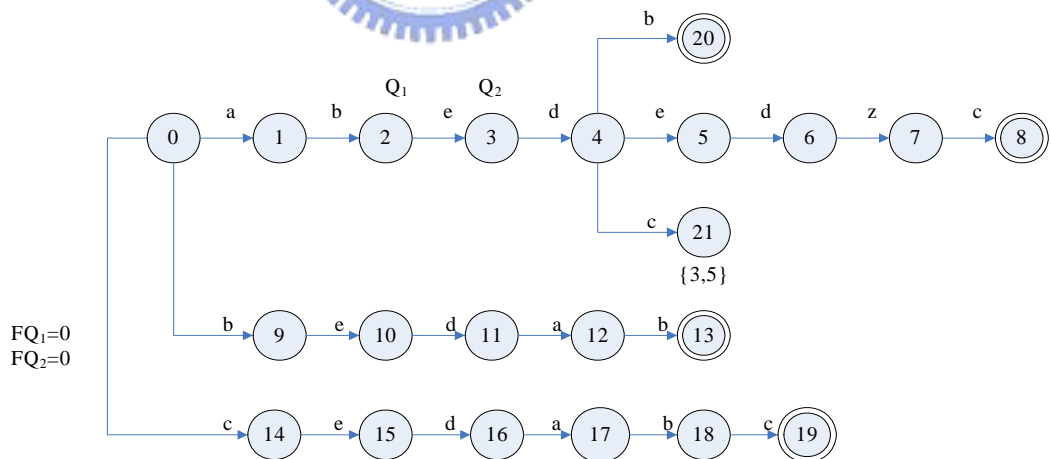


圖 5-1 (a) goto 狀態圖  $G'$

|            |     |     |    |     |     |     |     |    |    |     |     |
|------------|-----|-----|----|-----|-----|-----|-----|----|----|-----|-----|
| i          | 0   | 1   | 2  | 3   | 4   | 5   | 6   | 7  | 8  | 9   | 10  |
| failure(i) | END | END | 22 | 29  | 30  | 29  | 30  | 26 | 26 | END | END |
| i          | 11  | 12  | 13 | 14  | 15  | 16  | 17  | 18 | 19 | 20  | 21  |
| failure(i) | END | END | 22 | END | END | END | END | 22 | 22 | 26  | 28  |

圖 5-1 (b) goto 狀態圖 G' 之 failure 函式

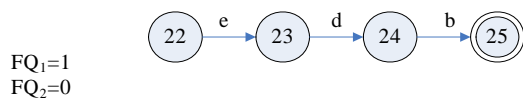


圖 5-1 (c) goto 狀態圖 F<sub>1</sub>

|            |     |     |     |     |
|------------|-----|-----|-----|-----|
| i          | 22  | 23  | 24  | 25  |
| failure(i) | END | END | END | END |

圖 5-1 (d) goto 狀態圖 F<sub>1</sub> 之 failure 函式

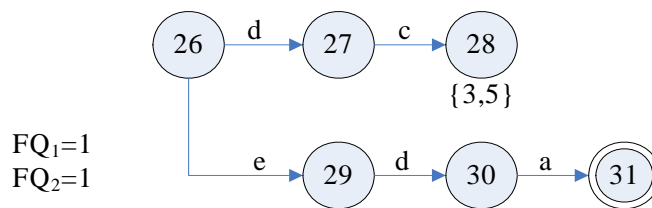


圖 5-1 (e) goto 狀態圖 F<sub>2</sub>

|            |     |     |     |     |     |     |
|------------|-----|-----|-----|-----|-----|-----|
| i          | 26  | 27  | 28  | 29  | 30  | 31  |
| failure(i) | END | END | END | END | END | END |

圖 5-1 (f) goto 狀態圖 F<sub>2</sub> 之 failure 函式

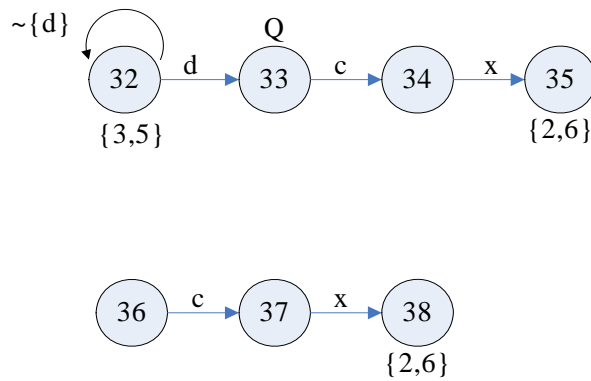


圖 5-1 (g) 正規表示式字樣  $RE_2$  第二段 goto 狀態圖

|            |    |    |    |    |
|------------|----|----|----|----|
| i          | 32 | 33 | 34 | 35 |
| failure(i) | -- | 36 | 37 | 38 |

|            |     |     |     |
|------------|-----|-----|-----|
| i          | 36  | 37  | 38  |
| failure(i) | END | END | END |

圖 5-1 (h) 正規表示式字樣  $RE_2$  第二段 goto 狀態圖之 failure 函式

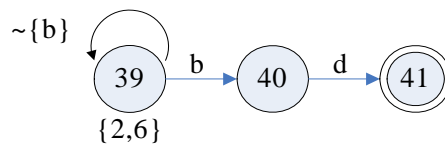


圖 5-1 (i) 正規表示式字樣  $RE_2$  第三段 goto 狀態圖

|            |    |    |    |
|------------|----|----|----|
| i          | 39 | 40 | 41 |
| failure(i) | -- | 39 | 39 |

圖 5-1 (j) 正規表示式字樣  $RE_2$  第三段 goto 狀態圖之 failure 函式

圖 5-1 使用 pre-filter 之範例

由圖 5-1 所示，goto 狀態圖  $G'$  以及非  $\{min, max\}$  圖之起始狀態皆沒有自我迴

圈，在  $G'$  中有兩個換圖狀態可換到圖 5-1 (c)、圖 5-1 (e)，當指標進入圖  $F_1$ ， $RE_2$  以及一般字樣所成之集合  $X$  將不再有機會被比對到。





## 第六章

### 實驗模擬結果與比較

本章我們將我們所提出之有限狀態字樣比對機之效能與 ClamAV 做比較。在 3-2 節我們已經介紹過 ClamAV 實做的方法，它只對所有字樣的前兩個位元組使用 AC 演算法建造有限狀態機，前 2 個位元組相同的字樣視為同一個群組 (group)，當前兩個位元組被比對到，則 ClamAV 針對此群組所有字樣逐一比對。它將正規表示式表示之字樣以 \*、?? 以及  $\{min, max\}$  運算子做分段，在比對的過程中會有個資料結構記錄比對成功到那一段。假使有一個正規表示式字樣有  $n$  段，若其前  $m$  段已經比對到且第  $m$  段在輸入字串  $x$  中比對到的位置為  $i$ ，接著若有另一段在位置  $j$  被比對到，若此段不是字樣的第  $m+1$  段或是  $i$  和  $j$  的位置不符合第  $m$  段和第  $m+1$  段之間運算子的規則，則此比對失效。舉例來說，正規表示式字樣  $RE = x_1 ?? x_2 \{3,5\} x_3 \{1,3\} x_4$ ，假設第一段  $x_1$  在輸入字串  $x$  的第  $i$  個位置比對到，第二段  $x_2$  在輸入字串  $x$  的第  $(i+|x_2|+1)$  個位置比對到，資料結構會記錄前兩段已經在輸入字串  $x$  的第  $(i+|x_2|+1)$  個位置比對到，若有另一段在位置  $j$  被比對到，假使此段不是  $x_3$ ，則此比對失效；若此段為  $x_3$ ，驗證位置  $i$  與位置  $j$  是否符合  $3 \leq j - i - |x_2| - 1 - |x_3| \leq 5$ ，符合則更新資料結構；反之，比對失效。

所以說，ClamAV 的做法有可能使該被比對到的字樣沒有被比對到 (false negative)，舉例來說，正規表示式字樣  $RE = x_1 ?? x_2 \{3,5\} x_3 \{1,3\} x_4$ ，假設輸入字串  $x = x_1 x_1 g x_2 t t x_3 k k x_4$ ，正常來說我們應該要比對到  $x_1 g x_2 t t x_3 k k x_4$ ，但是 ClamAV 並不會比對到，因為當比對到第二次  $x_1$  時，比對就已經失效了。

我們從 ClamAV 的字樣資料庫任選數個一般字樣以及正規表示式字樣來做

比較，每一個正規表示式字樣至少包含了一個 $\{min, max\}$ 運算子或是 $*$ 運算子，當我們使用 pre-filter 時，我們取其前 10-位元組(K)，以 3-位元組(L)為一組做 HASH，將 24 個位元(L\*8-bits)HASH 成 16 位元，所以 SHIFT 表共有  $2^{16}$  位置 (entry)。圖 6-1 是當我們選了 10 隻一般字樣以及 1 隻正規表示式字樣時，我們任選掃描檔案大小，從 20KB~100MB，且欲掃描之檔案中沒有任何的病毒字樣，其 CPU 的執行消耗時間比較。

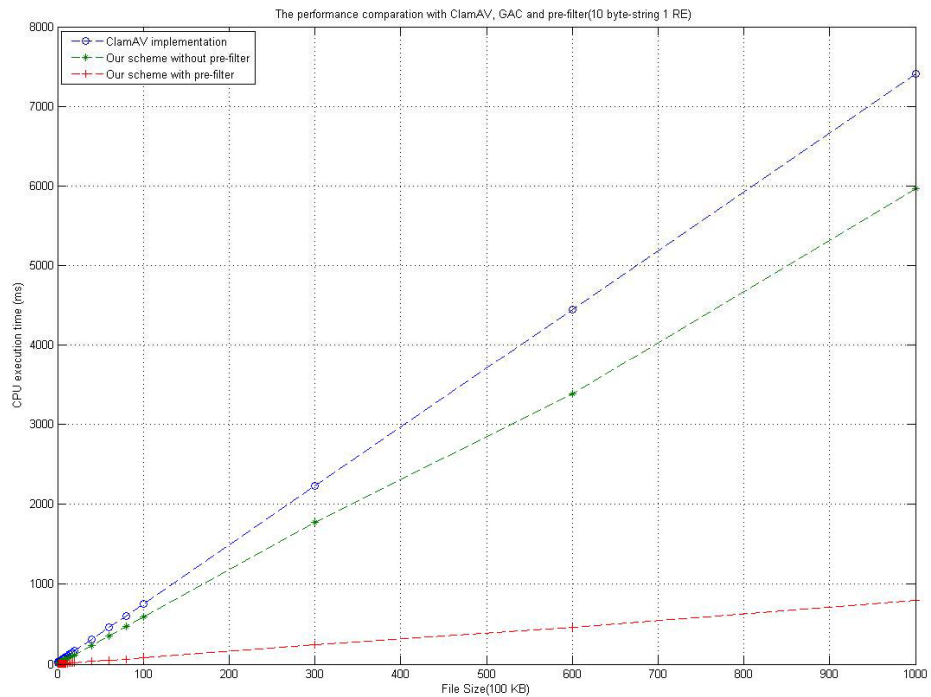


圖 6-1 檔案中無字樣時，針對不同掃描檔案大小，ClamAV、我們提出之方法效能比較(10 個一般字樣，1 個正規表示式字樣)

圖 6-2 是當我們選了 100 隻一般字樣以及 10 隻正規表示式字樣時，我們任選掃描檔案大小，從 20KB~100MB，且掃描檔案中沒有任何的病毒字樣，其 CPU 的執行消耗時間比較。由圖 6-1 以及圖 6-2 可以看出，CPU 執行消耗時間與掃描檔案大小成正比，我們提出的不使用 pre-filter 的方法比 ClamAV 的效能好大約 15%，若使用 pre-filter，效能大概會好 7 倍左右。

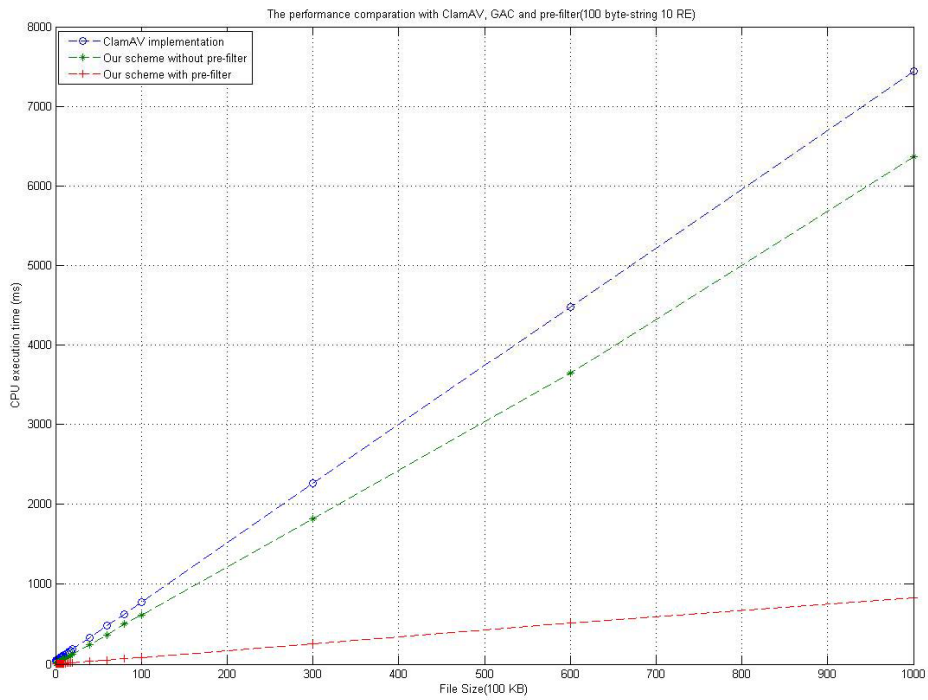


圖 6-2 檔案中無字樣時，針對不同掃描檔案大小，ClamAV、我們提出之方法效能比較(100個一般字樣，10個正規表示式字樣)

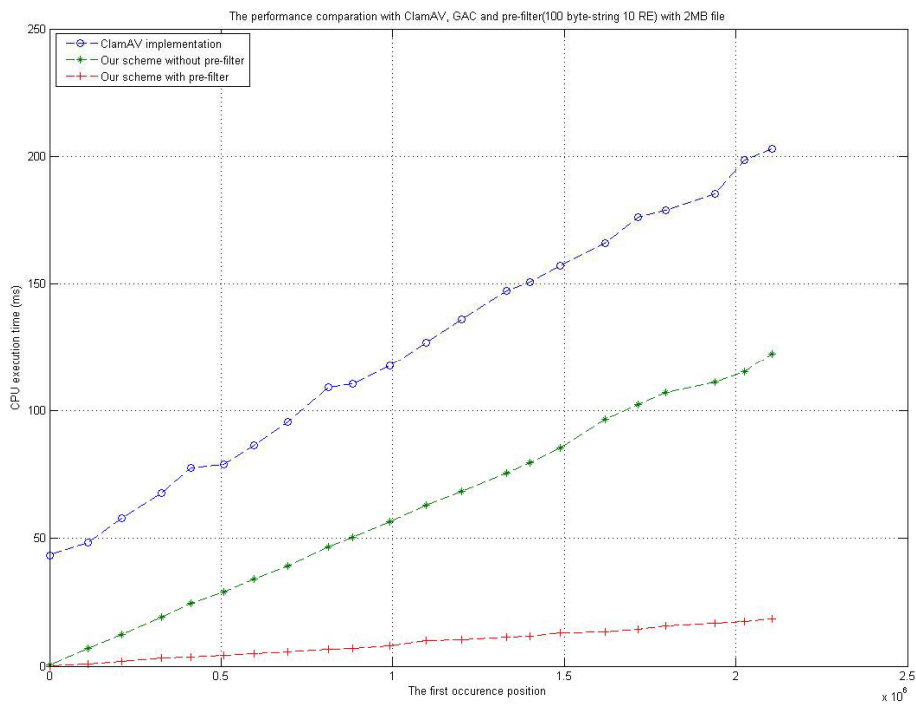


圖 6-3 檔案中有字樣時，ClamAV、我們提出之方法效能比較

圖 6-3 是當我們選了 100 隻一般字樣以及 10 隻正規表示式字樣時，選了一個大概 2MB 大小的掃描檔案，在檔案中的任意位置插入某一個正規表示式字樣，其 CPU 的執行消耗時間比較。由圖 6-3 可以看出，CPU 執行消耗時間與字樣在檔案中的位置成正比，我們提出的不使用 pre-filter 的方法比 ClamAV 的效能好大約 50%，若使用 pre-filter，效能大概會好 7 倍左右。當字樣在檔案的一開始，我們發現 ClamAV 也需要一些 CPU 執行消耗時間，那是因為有 4 個字樣與我們所要比對的字樣同一群組，5 個字樣逐一比對會花一些時間。




# 第七章

## 結論

---

本篇論文提出一種有限狀態機建造的程序，此有限狀態機可以用來做字樣比對。許多的字樣比對演算法已經在過去被發表，像是 Knuth-Morris-Pratt (KMP)，Boyer-Moore (BM)和 Aho-Corasick (AC)。KMP 和 BM 演算法只能對單一字樣比對，而不允許多個字樣同時比對。而 AC 演算法針對多個字樣建造一個有限狀態機，可以同時比對所有的字樣；且保證其在任何情況下有一定的效能。因此，AC 演算法在許多的系統中被廣泛的使用，特別是當效能為主要訴求時。



我們提出的字樣比對有限狀態機結構可以指出第一個病毒/蠕蟲惡意碼字樣發生的位置，字樣可以是一般字樣或是由某些特殊字元(\*、{*min*, *max*})所構成的正規表示式字樣。我們的字樣比對機與 AC 演算法類似，但是我們是由多個 *goto* 狀態圖組成，其間使用 *failure* 函式來做連接。我們所提出之方法若沒有使用 *pre-filter*，我們的效能已經比 ClamAV 好，若加上 *pre-filter* 更可大幅改善比對之效能。

## 參考文獻

---

- [1] Clam Anti-Virus signature database, <http://www.clamav.net>.
- [2] Clam Anti-Virus 0.90.3 user manual, <http://www.clamav.net>.
- [3] Clam Anti-Virus, <http://zh.wikipedia.org/wiki/ClamAV>.
- [4] D. E. Knuth, J. H. Morris, and V. R. Pratt, “Fast pattern matching in strings,” TR CS-74-440, Stanford University, Stanford, California, 1974.
- [5] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” Communications of the ACM, Vol. 20, October 1977, pp. 762-772.
- [6] A. V. Aho and M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search,” Communications of the ACM, Vol. 18, June 1975, pp. 333-340.
- [7] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Communications of the ACM, Vol. 13, no. 7, pp. 422-426, July 1970.
- [8] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching,” Technical Report, May 1994.

- 
- [9] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok, “Avfs: An On-Access Anti-Virus File System,” proceedings of the 13<sup>th</sup> USENIX Security Symposium, 2004.
- [10] T. H. Lee, ”Generalized Aho-Corasick Algorithm for Signatures Based Anti-Virus Applications,” ICCCN, August 2007.
- [11] D. Moore, C. Shannon, and J. Brown, “Code-Red: a case study on the spread and victims of an Internet worm,”in Proc. ACM/USENIX Internet Measurement Workshop, France, Nov. 2002.
- [12] CAIDA. Dynamic graphs of the Nimda worm, <http://www.caida.org/dynamic/analysis/security/nimda>.
- [13] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, “Inside the Slammer worm,” IEEE Security and Privacy, 1(4): 33-39, July 2003.
- [14] S. E. Schechter, J. Jung, and A. W. Berger, “Fast Detection of scanning worm infections,” 7<sup>th</sup> International Symposium on Recent Advances in Intrusion Detection (RAID), French Riviera, September 2004.
- [15] D. Seeley, “A tour of the worm,” in Proc. of the Winter Usenix Conference, San Diego, CA, 1989.

- [16] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, “Deep packet inspection using parallel bloom filters,” Symposium on High-Performance Interconnect (HotI), Stanford, CA, pp. 44-51, August 2003.
- [17] Creating signatures for ClamAV, <http://www.clamav.net>.





# 簡 歷

## 一. 個人資料

|        |  |
|--------|--|
| 姓名     | 吳柏庚  |
| 英文名    | Birken   |
| 出生年月日  | 1983/03/18   |
| 出生地    | 台北市萬華區   |
| 性別     | 男  |
| E-mail | <a href="mailto:birkenwu.cm94g@nctu.edu.tw">birkenwu.cm94g@nctu.edu.tw</a> |
| 實驗室    | 交大電信所 網路技術實驗室(工程四館 823 室)  |

## 二. 學歷

|     |                      |
|-----|----------------------|
| 國小  | 台北市立龍山國民小學           |
| 國中  | 私立延平高級中學國中部          |
| 高中  | 台北市立大安高級工業職業學校 電子科   |
| 大學  | 國立高雄第一科技大學 電腦與通訊工程系  |
| 研究所 | 國立交通大學 電信工程學系碩士班 系統組 |