國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

TRIPS 樹探勘演算法的改進

An Improved TRIPS Tree Mining Algorithm

佘健生

Chien-Sheng Sher

指導教授：顏嗣鈞 教授

Advisor:Yen Hsu-Chun, Professor

中華民國 97 年 7 月

July, 2008

誌謝

本論文承蒙本所顏嗣鈞教授悉心指導，感謝老師這幾年來的指導與教誨，讓我深刻了解到面對研究的態度與觀念，以及邏輯推理、觀念釐清、思考啟發和解決問題的方法，師恩浩瀚、永銘於心，僅此獻上我最高的謝意。論文口試期間，幸蒙郭斯彥教授、雷欽隆教授、莊仁輝教授、黃秋煌教授於百忙之中撥空指正，使論文更臻完善，在此也致上萬分感激。

另外，我也要感謝實驗室裡遇到了非常多好學長跟同學們，不只是於研究方面可以一起討論思考，在日常生活上也能如家人般和樂相處。所以在此要對計算理論實驗室的演龍、建良、紹祁、春成、紀憲、柏源、成典、克仁、浩仁、聖穎、子璿、育翰、翔宇、於芳、奇錚、文鴻、淑棻、志聰、佑叡、積瑞、亭遠、湘筠、昱鈞、秉賢、奕廷、人豪、昱凱、鄭維等好友致上誠摯的感謝。

最後，我要感謝父母的養育跟栽培以及家人的支持與體諒，讓我能無後顧之憂完成研究所的學業，因此在此我要將此篇論文獻給各位，願大家一同分享這份喜悅。

佘健生　僅識

於　計算理論研究室

中華民國九十七年七月

摘要

在處理大量資料時，將會需要一些特別的技巧來從資料中獲得比較有用的資訊，資料探勘（Data mining）便是一項從大量資料中尋找出隱藏在這些資料裡有用、相關資訊的技巧。在目前許多對於資料探勘的研究上，進展的方向也從探勘頻繁出現的物件集合逐漸朝向更加複雜的結構進行探勘，例如樹結構（tree）或圖（graph）等。

在本文中，我們所提到的這個探勘頻繁子樹的問題，在過去已經被證實了可用於許多廣範圍的應用上，像是生物資訊（bioinformatics）、XML 處理、計算語言學（computational linguistics）和 web 使用率探勘上。TRIPS 是一個從樹資料庫中探勘子樹的新演算法，這個演算法比起一些過去的演算法還快，並且可以很廣泛的用於針對嵌入或歸納子樹，而子樹可以是有標記、無標記、有序、無序等。本文中主要會針對 TRIPS 演算法對於嵌入、有序子樹和嵌入、無序子樹上提出改進的做法，在實驗結果上也可以看到我們的做法在速度上所獲得的提升。

在本文中，我們會先稍微介紹樹探勘演算法的概要與應用，然後說明介紹相關的基礎定義，再來詳細介紹一些有關的樹探勘演算法，並且之後會針對 Shirsh Tatikonda 提出的 TRIPS [1] 演算法，提出我們改進的地方，以及在實驗上所驗證出來的結果。

目錄

口試委員審定書…… i

誌謝…… ii

摘要…… iii

第一章 序論…… 1

第二章 理論基礎…… 6

2.1 基本定義…… 6

2.2 問題定義…… 12

第三章 樹探勘演算法…… 14

3.1 頻繁子集問題…… 14

3.2 TreeMiner & PatternMatcher 演算法…… 16

3.3 SLEUTH 演算法…… 24

第四章 TRIPS 演算法改進處 …… 28

4.1 TRIPS 演算法 …… 28

4.2 針對 TRIPS 演算法在資料結構上的改進…… 36

4.2.1 針對歸納、有序子樹的改進做法 …… 36

4.2.2 針對歸納、無序子樹的改進做法 …… 38

4.2.3 針對嵌入、有序子樹的改進做法 …… 39

4.2.4 針對嵌入、無序子樹的改進做法 …… 43

第五章 實驗結果與分析…… 46

5.1 實驗環境…… 46

5.2 實驗方法與結果…… 47

第六章 結論及未來工作展望…… 60

參考文獻…… 61

圖目錄

2.1 樹資料庫範例…… 8

2.2 歸納子樹的對應圖…… 9

2.3 嵌入子樹的對應圖…… 10

2.4 前序追蹤和後序追蹤的範例圖…… 11

2.5 一個樹資料庫…… 12

2.6 對嵌入、有序子樹進行樹探勘後的結果…… 13

2.7 對嵌入、無序子樹進行樹探勘後的結果…… 13

3.1 特性 1 的解說例圖…… 16

3.2 候選子樹的搜尋空間示意圖…… 18

3.3 Treeminer 的合法延伸例圖…… 19

3.4 兩種不同做法的解說例圖…… 23

3.5 標準形式例圖…… 25

3.6 標準形式反例解說圖…… 25

4.1 TRIPS 的資料結果例圖…… 30

4.2 TRIPS 最左路徑延伸例圖…… 31

4.3 TRIPS 演算法的做法例圖…… 32

4.4 改良後對歸納、有序子樹的資料結構例圖…… 37

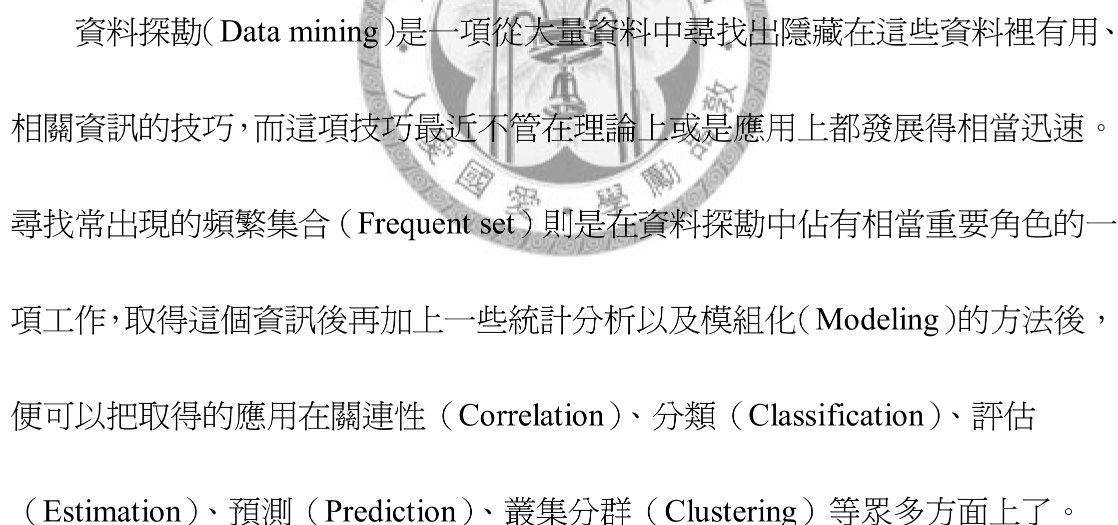
4.5 TRIPS 對歸納、有序子樹的做法說明…… 37

4.6 對歸納、有序子樹的改良做法說明⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 38
4.7 對歸納、無序子樹的示意說明圖⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 39
4.8 TRIPS 對嵌入、有序子樹的做法說明⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 40
4.9 新資料結構的特性說明⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 41
4.10 對嵌入、有序子樹的改良做法說明⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 42
4.11 對嵌入、無序子樹的改進資料結構例圖⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 43
4.12 對嵌入、無序子樹的證明例圖⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 44
針對中文樹結構資料庫的實驗結果
5.1 支持度在 10%下不同棵數的樹對歸納有序子樹之執行時間比較⋯⋯ 48
5.2 35000 棵樹對歸納有序子樹在不同最小支持度下的執行時間 ⋯⋯⋯ 49
5.3 在支持度 10%下各數量的樹對嵌入有序子樹之執行時間比較⋯⋯⋯ 50
5.4 5000 棵樹在不同最小支持度下對嵌入有序子樹的執行時間⋯⋯⋯⋯ 50
5.5 10000 棵樹在不同最小支持度下對嵌入有序子樹的執行時間⋯⋯⋯ 51
5.6 支持度在 10%時不同棵數的樹對嵌入無序子樹之執行時間比較⋯⋯ 52
5.7 5000 棵樹在不同最小支持度下對嵌入無序子樹的執行時間⋯⋯⋯⋯ 52
5.8 10000 棵樹在不同最小支持度下對嵌入無序子樹的執行時間⋯⋯⋯ 53
針對 CSLOGS 的實驗結果
5.9 支持度在 10%時不同棵數的樹對歸納有序子樹之執行時間比較⋯⋯ 55
5.10 50000 棵樹在不同最小支持度下對歸納有序子樹的執行時間⋯⋯⋯ 55
5.11 支持度在 10%時不同棵數的樹對嵌入有序子樹之執行時間比較⋯⋯ 56
5.12 10000 棵樹在不同最小支持度下對嵌入有序子樹的執行時間⋯⋯⋯ 56
5.13 10000 棵樹在不同最小支持度下對嵌入無序子樹的執行時間⋯⋯⋯ 57
針對 TREEBANK 的實驗結果
5.14 對歸納子樹的樹探勘結果比較⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 59
5.15 對嵌入子樹的社探勘結果比較⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯ 59

第一章

序論

1.1 研究動機與目的



資料探勘(Data mining)是一項從大量資料中尋找出隱藏在這些資料裡有用、相關資訊的技巧，而這項技巧最近不管在理論上或是應用上都發展得相當迅速。尋找常出現的頻繁集合（Frequent set）則是在資料探勘中佔有相當重要角色的一項工作，取得這個資訊後再加上一些統計分析以及模組化(Modeling)的方法後，便可以把取得的應用在關連性（Correlation）、分類（Classification）、評估（Estimation）、預測（Prediction）、叢集分群（Clustering）等眾多方面上了。

針對頻繁集合進行探勘的研究動機，最早是出自於分析超級市場中的交易資料，像是評估顧客購買商品的習慣上，然而頻繁集合的這個資訊，便能很清楚描述出哪些商品是最常被顧客一起購買，進而提供商家關於商品、擺設之類的改進。在這個問題中，便是由各項商品構成物件，而顧客交易的商品則是會形成由物件

所組成的集合，最後再由所有顧客交易所形成的集合來建構出資料庫，接著頻繁集合的探勘就能夠尋找出資料庫中頻繁出現的集合。

目前在這類資料探勘的研究上有一個趨勢就是往探勘更複雜的例子發展，而不是過去只針對物件的集合而已，像是單一表格的關連式資料庫如 XML 資料庫，以及多表格的關連式資料庫、化學分子資料庫、圖形資料庫等。然而因爲資料庫中包含了更加的資料類型，所以這些資料本身所加入的相互關係便有許多實務及理論上的問題需要有待解決。原本這些複雜、結構化的資料中，最一般、也最重要的形式就是圖形了，不過圖形一般而言會有比較討厭的理論特性，造成計算複雜度上的提升，像現在已知就沒有很有效率的演算法能夠解決一個圖形是否和另外一個子圖同構（isomorphic）的問題，而且再加上目前也沒有比較有效率的演算法去有系統的列舉出一個已知圖形中所具有的子圖。因此，針對一般圖形將會遇到相當嚴重的效率問題。所幸許多實際資料庫並非是由需要指數運算量的圖形所組成，造成圖形演算法會變得相當複雜的原因，常常都是因爲圖形中存在有迴路（cycle）。不過許多例子中，資料庫中圖形的迴路數目可能是有限的，而有些甚至根本就沒有迴路，例如說如果這些圖形是樹結構的話，那麼現在便有許多有效率的演算法可以解決這類問題。所以對於一些樹結構的資料庫或較少迴路的圖形資料庫來進行樹探勘演算法的研究，不只能符合實務的觀點而且還能得到計算上的效率。

樹探勘和圖形探勘等針對複雜型樣（pattern）的探勘，其主要的目的就是要

尋找出資料庫裡各資料之間的共同結構或者是共同關係，如果使用者得到這個資訊後，便能夠對這些資料進行更進一步的分類或預測等工作。所以相關的應用變會使用在分析網頁習慣（web usage）、XML 資料的分類與維護、生物資訊學（bioinformatics）、計算語言學（computational linguistics）、分析網路多點傳送（network multicast）等許多方面上。舉 XML 的例子來說，XML 文件中的元素（element）或屬性（attribute）可以對應成樹的節點，而元素和子元素（subelement）或屬性和值（value）之間的關係則是可以對應成樹的邊，因此我們便可以把 XML 文件轉成許多樹，然後進行樹探勘的工作，以找出眾多 XML 文件之間彼此共同的關係，再進行 XML 文件的分類。像是在 [1] 中，就使用到了 TreeMiner [2] 演算法來幫助進行 XML 文件的分類。另外在網頁習慣的方面，我們可以把每一位使用者瀏覽網站的動作轉變成一個存取樹（access tree），當中首頁可表示成樹根、使用者瀏覽過的網頁則可表示成節點，而網頁間的連結關係則是可以表示成樹的邊，因此收集了許多使用者的存取樹成為了資料庫後，就能夠分析使用者們瀏覽網頁的行為與習慣，然後可以針對網頁的不足處進行修改與加強。在 [2] 中，Mohammed J. Zaki 便使用了 Log Markup Language 來描述網站的使用記錄（web log），然後分析瀏覽網站的使用者來源，一類是來自學術網路的使用者，另一類是其他網域的使用者。此外在其他方面上，像是生物資訊等地方，Mohammed J. Zaki 在提出 TreeMiner 演算法時，也在該篇論文中提到了把 RNA 結構轉變成樹結構，然後再透過 TreeMiner 的樹探勘演算法幫助分析 RNA 結構的資料庫。

因此到現在爲止，也已經有許多人提出了在樹狀結構資料庫中針對各種不同樣式的子樹（Subtree）進行挖掘的演算法了，像是針對各種應用問題中所需要的地方，選擇如「有根的（rooted）」、「無根的（unrooted）」、「有序的（ordered）」、「無序的（unordered）」、「歸納的（induced）」、「嵌入的（embedded）」等各種不同組合所要探勘的子樹。例如由 Asai 所提出的 FREQT[?]演算法就是用於探勘所有頻繁、有根、歸納、有序樹狀結構的做法，Asai 與 Nijssen 也提出了 Unot [3] 和 uFreqt [4] 這兩個用來尋找頻繁、有根、歸納、無序樹狀結構的演算法。而 Zaki 則提出了 TreeMiner 這個用來尋找頻繁、有根、嵌入、有序樹狀結構的方法，然後他也另外提出了 SLEUTH [5] 這個改良自 TreeMiner，並且是用於找尋頻繁、有根、嵌入、無序樹狀結構的演算法。此外，Shirsh Tatikonda 也提出了 TRIPS [5] 演算法不但速度更快速，而且能夠很泛用的使用在尋找頻繁、有根、歸納或嵌入、無序或有序的樹狀結構上。

本文主要是針對 TRIPS 演算法在探勘頻繁、有根、歸納、有序或無序的樹狀結構上，以及探勘頻繁、有根、嵌入、有序或無序的樹狀結構上，提出改進的方法，以提升在尋找時的速度。

1.2 論文架構

本論文其餘的組織結構如下。在後面第 2 章的部分，我們會講述到一些與樹結構有關、本論文中會使用到樹探勘演算法的基本理論。接著在第 3 章中，則是會詳細描述各演算法比較詳細的運作內容的內容。第 4 章的部分是本論文所提出來改進 TRIPS 的資料結構做法。第 5 章是改良做法的實驗結果與分析。第 6 章則是提出結論與未來工作的展望。最後面的是參考資料。

第二章

基礎理論

在本章中，我們將會介紹一些本論文會使用到的 tree 的相關定義與特性，以及我們問題中會用到的 subtree 相關定義，然後還有在 tree mining 問題中會使用到的基本定義與數值，並且詳細描述 tree mining 問題與結果。

2.1 基本定義

2.1.1 圖（graph）

定義：一個有向圖 G = (V , E)由以下的兩個部分所組成：

1. 所有節點（vertex）的集合 V，
2. 所有邊（edge）的集合 E，而對於所有的邊 e ∈ E，e = (v_1,v_2)，其中 v_1 和 v_2 ∈ V。

2.1.2 樹（tree）

一棵樹是一般而言是個沒有迴路（cycle）的無向連通圖（undirected, connected graph）。由許多棵樹所形成的集合，則稱爲森林（forest）。在本篇論文中主要會針對的樹結構是有根、標記樹，所以此樹可以用 T = (V , E , L , r)來表示，其中 V = {V_1 , V_2 , …}爲節點的集合，E = {(x , v) | x , y ∈ V}，L = {l_1 , l_2 , …}爲標記的集合；樹中的節點可透過一標記函數（labeling function）l : V → L，將所有點都對應到 L 的集合中，所以 L(x)便代表節點 x 的標記；r ∈ V 則是一個特別的節點，稱之爲樹根。在這棵樹中，如果 x , y ∈ V 並且存在有一條從 x 走到 y 的路徑，那麼就代表 x 可以稱爲 y 的祖先（ancestor），這可以表示爲 x ⇐ py，其中 p 所表示的是這條從 x 到 y 的路徑其長度爲 p。當 p 爲 1 時，也就是 x ⇐ 1y，則可以把 x 稱爲 y 的父節點（parent）並且稱 y 爲 x 的子節點（child）。如果 x 和 y 兩點都具有同一個父親時，x 和 y 的關係就可以稱爲兄弟（sibilngs），又如果 x 和 y 是擁有同樣的祖先時，x 和 y 的關係可以稱爲親戚（cousins）。所有節點中，除了樹根節點外，還會分成內部節點（internal node）和樹葉節點（leaf），有子節點的爲內部節點，無子節點的爲樹葉節點。深度（depth）爲從樹根走到某個節點的距離。高度（height）則是從樹根到最底層樹葉節點的路徑長度。另外，如果一棵樹中所有節點的子節點，左右的兄弟節點間位置都是有順序時，則爲有序樹（ordered tree），如果是無次序時則爲無序樹（unordered tree）。圖 2.1 就是一個範例的樹資料庫。

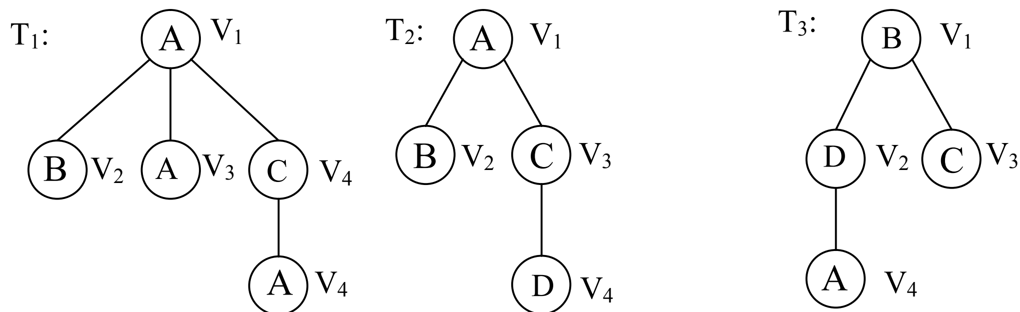


圖 2.1：樹資料庫範例

2.1.3 子樹（subtree）

同構子樹（isomorphic subtree）

定義：一棵樹 $S = (V_S, E_S)$和一棵樹 $T = (V_T, E_T)$，我們可以說 S 是 T 的同構子樹，若且爲若存在有一個一對一（onr-to-one）的映射函數 $\varphi : V_S \rightarrow V_T$，使得$(x , y) \in E_S \Leftrightarrow (\varphi(x), \varphi(y)) \in E_T$。

φ 爲映成（onto）時，則 S 和 T 可稱爲同構。

歸納子樹（induced subtree）

定義：一棵標記樹 $S = (V_S, E_S)$被稱爲另一棵標記樹 $T = (V_T, E_T)$的歸納子樹，記爲 $S \preccurlyeq_i T$，若且爲若 S 是 T 的同構子樹，並且映射函數 φ 確保標記關係，例如對$\forall x \in V_S$，讓 $l(x) = l(\varphi(x))$。

歸納子樹可以確保父節點和子節點的關係（parent-child relationships），也就是說

在歸納子樹 S 中任意兩個點如果是父節點-子節點關係的話，這兩個點在包含歸納子樹 S 的樹 T 中所對應到的兩個點也會是父節點-子節點的關係。圖 2.2 是歸納子樹的對應解說圖。

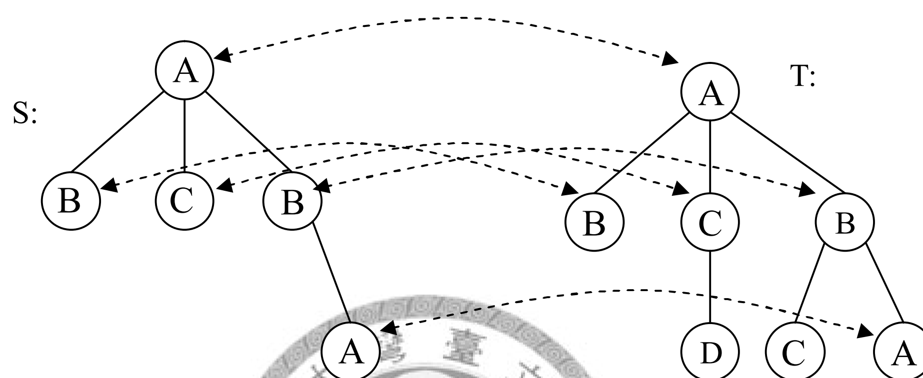


圖 2.2：歸納子樹的對應圖

國立臺灣大學 愛國 愛學 勤 品 敦

嵌入子樹（embedded subtree）

定義：一棵標記樹 $S = (V_S, E_S)$被稱爲另外一棵標記樹 $T = (V_T, E_T)$的嵌入子樹時，記爲 $S \preccurlyeq_e T$，若且爲若存在一個一對一的映射函數 $\varphi: V_S \rightarrow V_T$，以致於能滿足: i) $(x, y) \in E_S \Leftrightarrow \varphi(x) \preccurlyeq_p \varphi(y)$，ii) $l(x) = l(\varphi(x))$。

嵌入子樹能夠確保節點的祖先和子孫關係（ancestor-descendant relationship），也就是說在嵌入子樹 S 中的任意兩個點如果是祖先和子孫的關係時，這兩點在包含嵌入子樹的樹 T 中所對應到的兩個點也會是祖先和子孫的關係。圖 2.3 是嵌入子樹的對應解說圖。

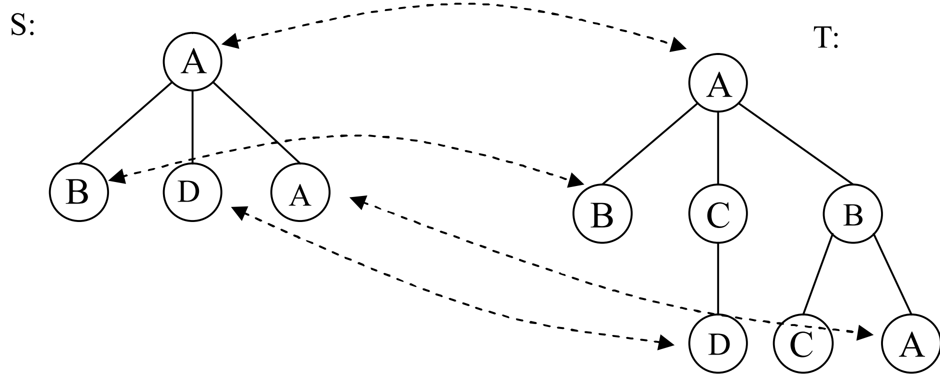


圖 2.3：嵌入子樹的對應圖

2.1.4 支持度（support）

令 D 為一群樹的資料庫（森林）為 $\{T_i \mid i = 1, 2, \ldots, n\}$，而 $\delta_T(S)$為某棵子樹 S 在一棵樹 T 中的發生次數（可以是歸納子樹或嵌入子樹），以及 d_T 為一個指標變數，其值為 0 或 1，然後詳細的定義如下：

$$d_T(S) = \begin{cases} 0, & \text{if } \delta T(S) > 0 \\ 1, & \text{if } \delta T(S) = 0 \end{cases}$$

因此一棵子樹 S 在資料庫 D 中的支持度就定義為：

$$\sigma(S) = \sum_{T \in D} d_T(S)$$

2.1.5 頻繁（frequent）& 不頻繁(unfrequent)

一棵子樹 S 的支持度如果超過使用者所設定的最小支持度（minmum support）時，則 S 便為頻繁的子樹。反之，如果 S 的支持度未達最小支持度時，則 S 為

不頻繁的子樹。

2.1.6 樹追蹤（tree traversal）

後序追蹤（post-order traversal）

一個節點只有在他的所有子節點都被追蹤到後才會被追蹤。

前序追蹤（pre-order traversal）

一個節點被追蹤到時，他的所有子節點都仍尚未被追蹤到，這個追蹤法也可稱爲深度優先追蹤。執行前序追蹤後，會得到深度優先序列（depth first order sequence）。

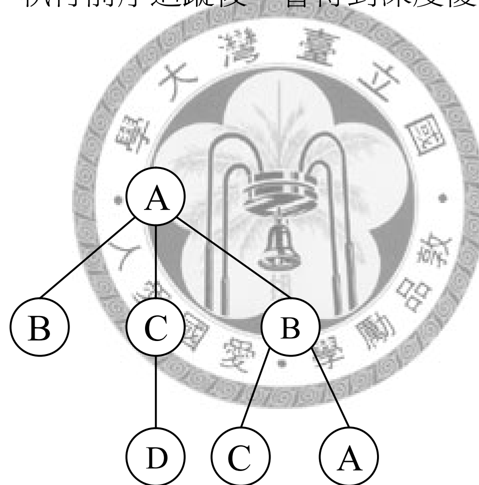


圖 2.4：前序追蹤和後序追蹤的範例圖

從圖 2.4 的樹進行前序追蹤會得到 ABCDBCA，進行後序追蹤會得到 BDCCABA。

樹表示法（tree representation）

在一些相關演算法當中，會直接把樹編碼成特定的序列，以便能讓一個序列

能夠唯一表示同個結構的樹和方便產生子樹等地方。一般常用的有利用深度優先序列，再加上回溯符號。例如上圖中的深度優先序列為 ABCDBCA，加上回溯符號'$'的話，就會成為 AB$CD$$BC$A，其中回溯符號加入的地方是深度優先追蹤執行到樹的樹葉節點（leaf）時，每往後退一步就加入一個回溯符號。

2.2 問題定義

樹探勘工作（Tree Mining Tasks）

給予一群樹的集合 D 和一個由使用者所設定的最小支持度數值，這時我們可以定義幾種樹探勘的工作，像是針對有根/無根、有序/無序、歸納/嵌入子樹等。在本論文中，主要是列舉出 D 中所有頻繁、有根、嵌入的有序或無序子樹。

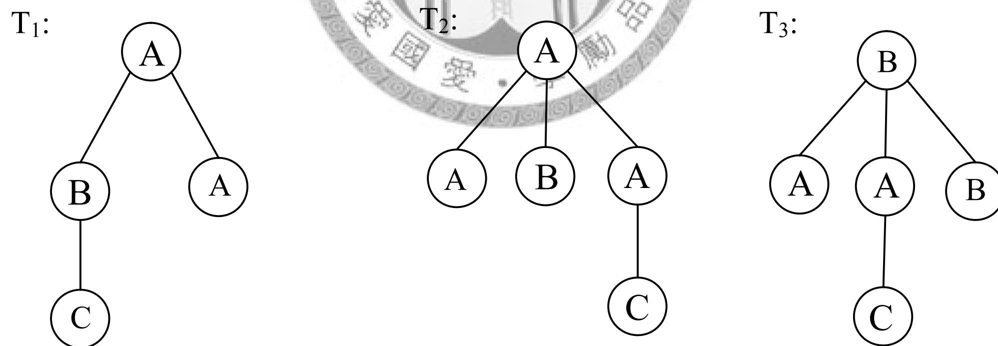


圖 2.5：一個樹資料庫

從圖 2.5 中，假設有三顆樹的集合 D，標記集合 L = {A , B , C}，並且使用者所設定的最小支持度為 2，則經過針對頻繁、嵌入、有序子樹的樹探勘工作後，會得到圖 2.6 的結果。

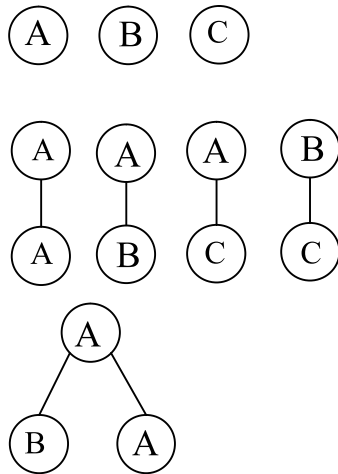


圖 2.6：對嵌入、有序子樹進行樹探勘後的結果

而如果是針對頻繁、嵌入、無序子樹的樹探勘工作後，則會得到圖 2.7 的結果。

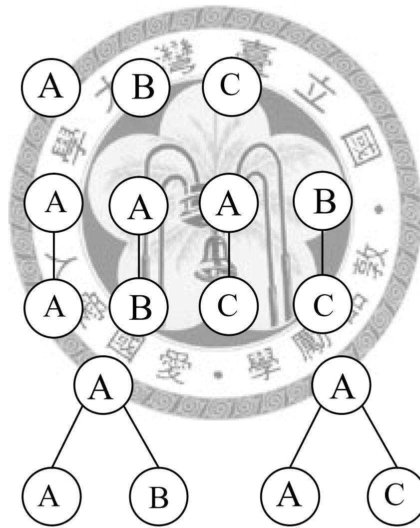


圖 2.7：對嵌入、無序子樹進行樹探勘後的結果

在上面，我們講解完了一些與樹有關的定義和特性，並且描述完了 Tree mining 的問題後，在下一章中，我們便會詳細說明一些樹探勘演算法的運作方式。

第三章

樹探勘演算法

在本章中我們將會介紹一些資料探勘（data mining）相關問題的做法所會用到的特性，和一些基本的樹探勘演算法（tree mining algorithm）以及這些演算法的基本構成與運作方式。

3.1 頻繁子集問題（frequent subset）

在資料探勘問題中，有一類問題就是尋找頻繁子集的問題，而樹探勘問題便是屬於這一類，不過尋找的子集在樹探勘問題中就如之前所描述的一樣，是改爲尋找可能出現的子樹。在頻繁子集問題中，一般的演算法基本上都是要經過兩個步驟，一個是候選者的產生（Candidate Generation），另外一個是支持度的計算（support counting）。候選者的產生就是要把所有可能出現的子集列舉出來，因

此這個步驟將會遇到的挑戰就是產生候選者的搜尋空間（search space）會指數增加，所以將會造成計算量大幅增加。支持度的計算就是要把所產生出來的子集，計算其在原本集合的資料庫中，所出現的支持度爲多少，這個步驟可能會遇到的問題是要如何避免計算到重複出現的子集合。

3.1.1 反單調特性（anti-monotone property）

在大部分相關的演算法中，對於子集的支持度都會存在有一個特性，這個特性就是支持度的單調特性（support monotonicity）。底下我們會針對樹探勘的問題描述這個特性。

特性 1（支持度的單調特性）

給定一個由許多樹所組成的資料庫（森林）$D = \{T_1, T_2, \ldots\}$，並且存在兩棵子樹 $S_x, S_y \preccurlyeq T_i, T_i \in D$，則

$$S_x \preccurlyeq S_y \Longrightarrow \text{support}(S_y) \le \text{support}(S_x)$$

這也就是說如果子樹 S_x 被包含於 S_y 的話，S_x 的支持度只會等於或大於 S_y 的支持度。換句話說就是如果一個集合是頻繁的，那麼該集合的所有子集合（subset）也會是頻繁的。

與這個特性相反過來的就是反單調特性了。如果子樹 S_y 包含子樹 S_x 的話，S_y 的支持度只會等於或小於 S_x 的支持度。因此這就是說如果有一個集合是不頻

繁的，那麼該集合的所有超集合（superbset）也會是不頻繁的。

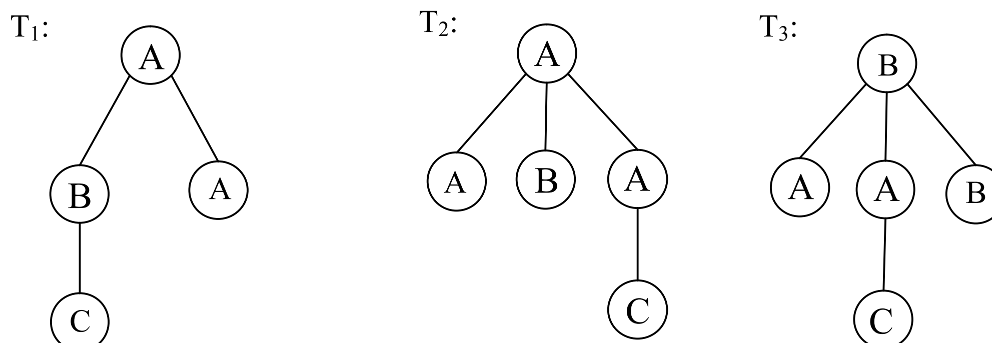


圖 3.1：特性 1 的解說例圖

就由圖 3.1 中，3 棵樹所形成的資料庫來說，這些樹只由 A、B、C 這 3 個標記的節點構成。因此舉個極端的例子，裡面並不存在有由 D 所標記的節點，所以任何由 D 節點所形成的樹結構，都不會出現在這個資料庫中，T_1、T_2、T_3 也不會包含有任何存在 D 節點的子樹。

因此利用反單調特性，在產生候選子樹的步驟上，我們便可以去除掉所有不頻繁的子樹，然後省去要計算包含這些不頻繁子樹之子樹超集合所需的計算量了。

在底下我們會介紹一些基本的樹探勘演算法以及本論文主要會改良的演算法。

3.2 TreeMiner & PatternMatcher 演算法

TreeMiner 演算法 [2] 是由 Mohammed J. Zaki 所提出的，這是一個針對頻繁、

有根、嵌入、有序子樹的問題所發展的演算法。這個演算法在產生候選子樹方面，除了有運用反單調特性之外，也採用了深度優先追蹤的樹結構表示法，並且候選子樹的產生，是從單一個節點開始生成，然後計算生成的候選子樹之支持度，如果超過使用者所設定的最小支持度時，就存留下來繼續生成更多節點的子樹，如果未達最小支持度時，便依照反單調特性而去除掉。而每個子樹在加入新的點時，都是加在這棵子樹的最右路徑（rightmost path）上的，利用這個方式就可以避免掉產生重複的候選子樹了，而且加上這個方法相當子樹的深度優先序列，因爲對一個深度優先序列來說，加在最右路徑上的點也等同於加在深度優先序列的最後方。所謂的最右路徑，就是從根（root）開始往葉節點（leaf）走，並且在遇到分支時，一定選擇搜尋最右邊的節點。

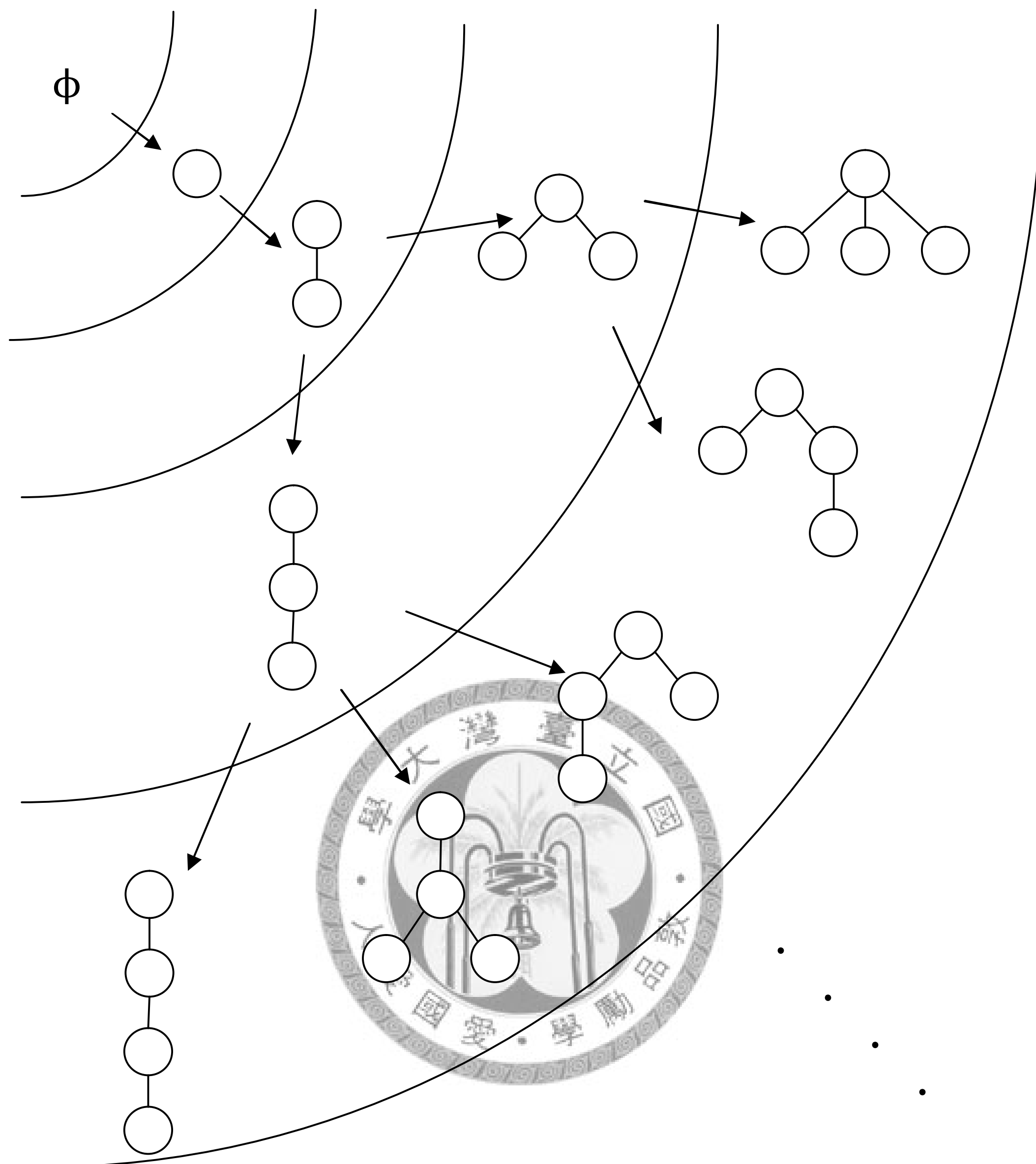
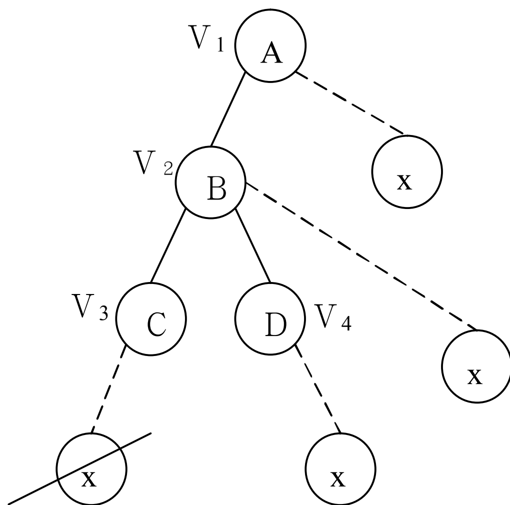


圖 3.2：候選子樹的搜尋空間示意圖

圖 3.2 為針對有序樹的搜尋圖，產生的子樹會從一個節點慢慢增加，並且是在子樹的最右路徑上增加新的節點。而在 TreeMiner 演算法中，則是還加入了等價類（equivalence class）的做法，其定義是為有兩個 k 個節點大小的子樹 X, Y 是屬於同樣的前序等價類（prefix equivalence class）時，若起為若這兩個子樹共同擁有 k – 1 個點的相同前置字串（prefix）。

這段描述的輔助定理，這個輔助定理在 [2] 的文章中有詳細的描述與證明。



Equivalence Class

前序字串（prefix string）：ABC$D

合法的加上節點：

(x , 1) 接到 V_1 ： ABC$D$$x$

(x , 2) 接到 V_2 ： ABCDx$$

(x , 4) 接到 V_4 ： ABC$Dx$$

不合法的加上節點：

(x , 3) 接到 V_3 ： ABCx$$D$$

圖 3.3：Treeminer 的合法延伸例圖

透過這樣產生候選子樹的規定，就可以避免產生重複的候選子樹了。底下是有關

這段描述的輔助定理，這個輔助定理在 [2] 的文章中有詳細的描述與證明。

輔助定理：

對任意 k ≥ 2，如果 S 是一個（k - 1）個節點的子樹，則任何一個從 S 的最右路徑增加一個節點所形成子樹 T 都會是 k 個節點的子樹。而如果 T 是 k 個節點的子樹時，那一定存在一個唯一的（k - 1）個節點的子樹 S，能讓 S 從最右邊路徑加入一個節點後形成 T。

證明：

如果 S 從最右路徑加上一個節點形成 T 後，很明顯的|T| = k。因此，一個新的節點 $V_k = k$ 被連接到 S 最右路徑上的一個節點時，它也會是 T 的前序追蹤序列的最後一個節點。另一方面，假設若數個（k-1）節點的子樹在最右路徑上加入 k 節點後都能形成 T，然而這時如果移開 T 的最後一個節點 k 時，就只會形成一個 T 的前置字串而已。所以加入 k 形成 T 的方式是唯一的，而 T 的前置字串也會是唯一的。

另外，在 TreeMiner 演算法中，是透過屬於同一個等價類的子樹互相形成新的候選子樹。不過這些部分與本論文比較沒有關係，所以這裡就不加以詳述了。

TreeMiner 演算法：

```
TREEMINER (D,  minsup):

    F1 = { 只有 1 個節點頻繁子樹 };

    F2 = { 2 個節點的頻繁子樹所組成的等價類[P]1 };

    for 所有的等價類[P]1 ∈ F2 do Enumerate-Frequent-Subtrees( [P]1 );


ENUMERATE-FREQUENT-SUBTREES( [P] ):

    For 每個元素（x, i）∈ [P] do

        [P_x^i] = ϕ;

        for 每個元素（y, j）∈ [P] do

            R = { (x, i) ⊗ (y, j)};

            L(R) = { L(x) ∩⊗ L(y)};

            if 對於任何 R ∈ R ，R 是頻繁的話 then

                [P_x^i] = [P_x^i] ∪ {R};

            Enumerate-Frequent-Subtrees( [P_x^i] );
```

TREEMINER 是 TreeMiner 演算法的主程式部分，先找出所有 1 個節點和 2 個節點的頻繁子樹，然後再把 2 個節點的頻繁子樹所形成的等價類，傳給 ENUMERATE - FREQUENT - SUBTREES 執行。在 ENUMERATE - FREQUENT - SUBTREES 這個副程式中，將會以遞迴的形式列舉出所有的頻繁子樹。

PatternMatcher 演算法與 TreeMiner 演算法的許多地方都很相似，兩者的差異處在於 TreeMiner 在產生候選子樹所採用的策略是樣式成長(Pattern-growth)做法，而 PatternMatcher 是採用演繹基礎（Apriori-based）做法，這個做法與資料探勘中的 Apriori 演算法很相似。底下是 PatternMatcher 演算法：

PATTERNMATCHER (D, minsup):

1. F_1 = { 只有 1 個節點頻繁子樹 };

2. F_2 = { 2 個節點的頻繁子樹所組成的等價類$[P]_1$ };

3. for (k = 3; $F_{k-1} \neq \phi$; k = k + 1) **do**

4. C_k = { k 個節點的候選子樹所組成的等價類[P]k-1 }

5. **for** 在 D 中的所有樹 **do**

6. 對所有 $S \preccurlyeq T$, $S \in [P]_{k-1}$，增加次數;

7. C_k = { k 個節點的頻繁子樹所組成的等價類};

8. F_k = { 在 C_k 中所有頻繁子樹所構成的雜湊表};

9. 所有頻繁子樹的集合 = $\cup_k F_k$;

PatterMatcher 演算法的第 1 行與第 2 行也是和 TreeMiner 演算法的主程式部分相同，都是找出所有 1 個節點和 2 個節點的頻繁子樹，然後再把 2 個節點的頻繁子樹所形成的等價類。而第 3～9 行的部分，則是以迴圈的方式找出所有的頻繁子樹。

演繹基礎做法的演算法，在產生候選子樹的搜尋空間上，使以廣度優先搜尋（breadth-first search）的方式產生候選子樹，而樣式成長做法則是以深度優先搜尋（depth-first search）的方式產生。圖 3.4 是一個解說的例圖。

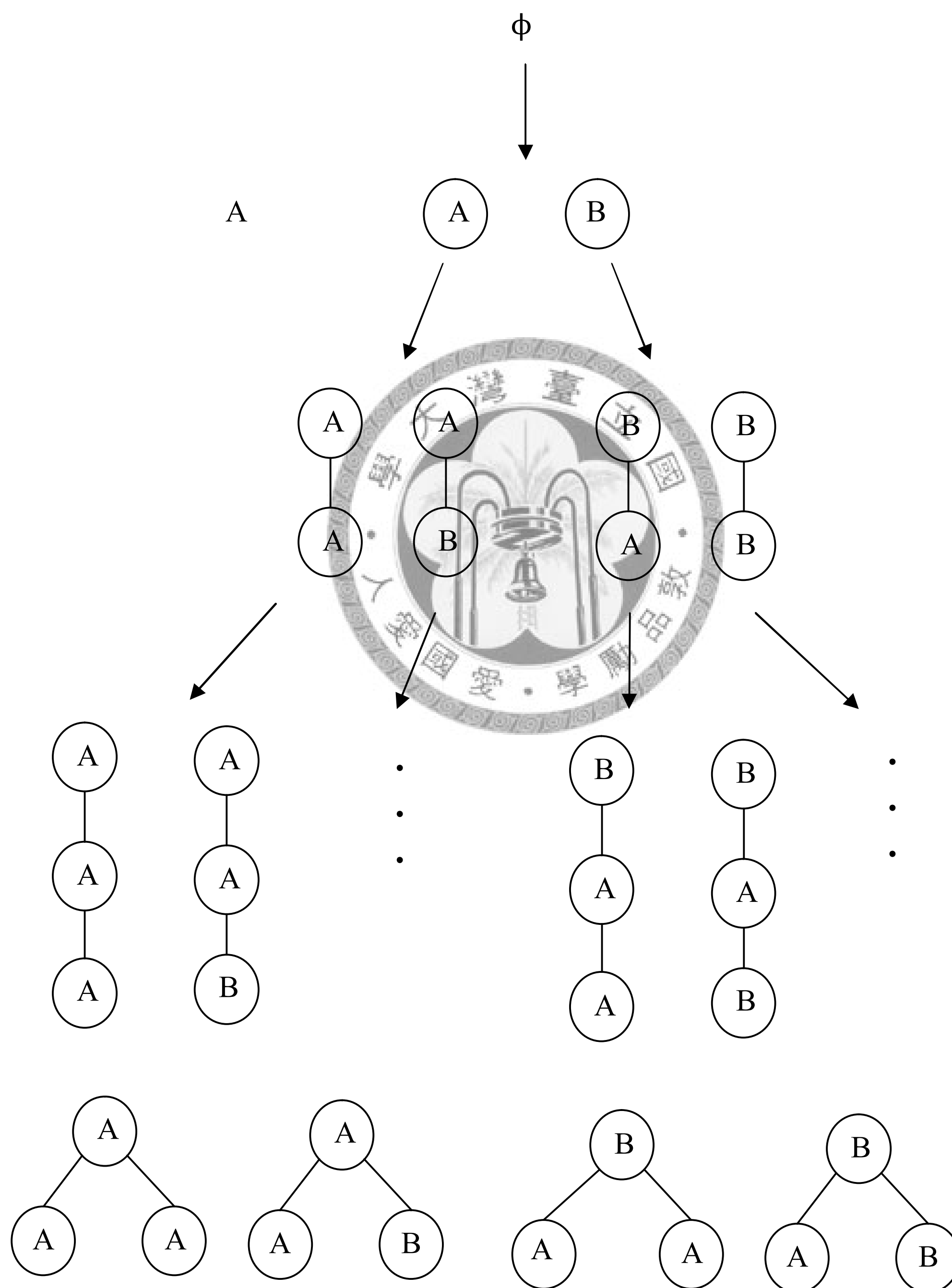


圖 3.4：兩種不同做法的解說例圖

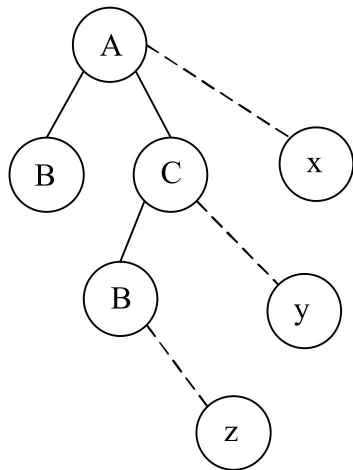
演繹基礎做法會一層一層的往下做，樣式成長做法則是會先從一條路徑做到底再退回來。

3.3 SLEUTH 演算法

SLEUTH 演算法[6]是由提出 TreeMiner 演算法的作者 Mohammed J. Zaki 自己改良 TreeMiner 演算法成爲可以針對頻繁、有根、嵌入、無序子樹的做法。SLEUTH 演算法基本上大部分的地方都與 TreeMiner 演算法相同，而其中不一樣的地方就在於候選子樹的產生以及支持度的計算上。在候選子樹的產生上，SLEUTH 演算法改良成了標準式延伸（Canonical Extension），以下便是標準式擴充的詳細內容。

標準式延伸（Canonical Extension）：

爲了確認一棵樹是否爲標準形式（Canonical form），我們需要確定每一個節點 $v \in T$，對於所有的 $I \in [1, k]$都會是 $T(c_i) \leq T(c_{i+1})$，這裡 $c_1, c_2, \ldots, c_k$ 是節點 v 的子節點的有序列表。因此我們產生一個新的候選子樹時只會把延伸成一棵標準式的樹，而這棵候選子樹的前置字串也會是標準形式。



x 點能夠加上的只有標記爲 C, D, …以後的節點，而 y 點能夠加上的只有標記爲 B,C, D, …以後的節點。而 z 點則是任何節點皆可。

圖 3.5：標準形式例圖

那爲何要採用這種做法呢？我們在這裡會舉個例子作說明。

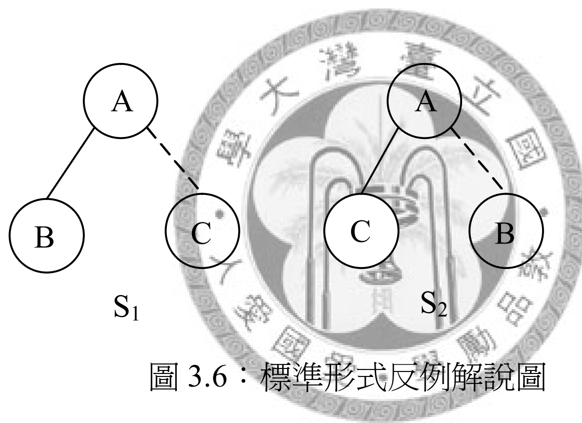


圖 3.6：標準形式反例解說圖

在圖 3.6，原本 S_1 的子樹 AB 加一個節點 C 到 A 節點上，在探勘有序子樹的候選子樹產生過程中是合法的延伸。而 S_2 的子樹 AC 加一個節點 B 到 A 節點上，在探勘有序子樹的候選子樹產生過程中也是合法的延伸。不過這兩個過程所產生出來的候選子樹，對無序子樹來說其實是相同的一棵子樹，因此如果採取過去的方法，就會重複計算了，所以造成了計算量的增加與時間的浪費。

SLEUTCH 演算法：

```
SLEUTH (D,  minsup):

1.F1 = { 只有 1 個節點頻繁子樹 };

2.F2 = { 2 個節點的頻繁子樹所組成的等價類[P]1 };

3.for 所有的等價類[P]1 ∈ F2 do Enumerate-Frequent-Subtrees( [P]1 );


ENUMERATE-FREQUENT-SUBTREES( [P] ):

4.    For 每個元素（x, i）∈ [P] do

5.        if check-canonical( P_x^i ) ;

6.        [P_x^i] = ϕ;

7.        for 每個元素（y, j）∈ [P] do

8.            if do-child-extension then L_d = dedcendant-scope-list-jion{(x, i), (y, j)};

9.            if do-cousin-extension then L_c = cousin-scope-list-jion{(x, i), (y, j)};

10.           if child 或 cousin extension 的結果是頻繁的話 then

11.               加入( y , j )和/或(y , k-1)到等價類[P_x^i]中;

12.           Enumerate-Frequent-Subtrees( [P_x^i] );
```

SLEUTCH 演算法與 TreeMiner 演算法主要的不同處是第 5 行和第 8、9 行。

第 5 行是確認子樹是否爲標準形式。而第 8 行是確認新增加的節點是否加在最右

路徑、最底下的葉節點上，因爲加在這裡的話就不需要確認是否爲標準形式了。而第 9 行是判斷新增加的節點是否加在最右路徑的其他地方，是的話便要確認這個節點的標記是否有比所左邊節點的標記大於或等於。

TRIPS 演算法是屬於比較相當泛用的做法，能夠針對頻繁、有根、標記或無標記、歸納或嵌入、有序或無序子樹進行樹探勘的工作，並且執行的速度更勝過 TreeMiner 等演算法。所以在下一章裡，我們便會詳細說明該演算法與針對演算法所改進的地方進行詳細的說明。

第四章

TRIPS 演算法與改進處

臺 灣 大 學 國 教 品 勵 學 · 愛 國 人

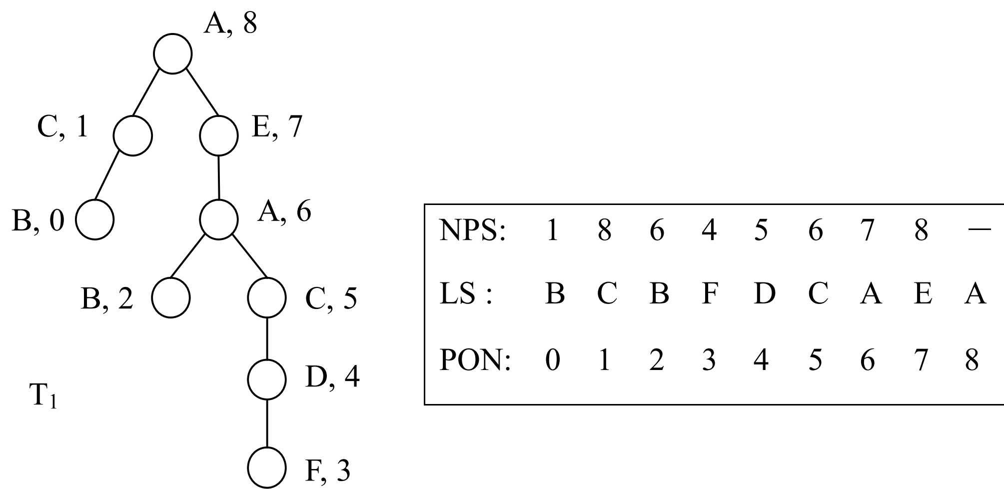
在本章中，我們將會詳細解說 TRIPS 演算法的機制與運作方式，並且針對不足的地方進行的改良，主要是在資料結構上進行改變，然後提出根據不同特性的子樹所使用的方法。

4.1 TRIPS 演算法

TRIPS 演算法[5]是 Shirish Tatikonda 所提出的樹探勘演算法，這個演算法與之前的演算法在許多地方上都有很大的不同，並且採用了許多不一樣的資料結構，不過 TRIPS 演算法的速度比 TreeMiner 還要更快。TRIPS 演算法主要是針對頻繁、有根、嵌入、有序子樹進行樹探勘工作，並且這個演算法也和 TreeMiner 演算法同屬樣式成長（Pattern-growth）類的做法，但 TRIPS 演算法所使用的資料結構與 TreeMiner 演算法不一樣，它用來表示樹的資料結構是後序追蹤序列（post-order

traversal sequence），使用這個後序追蹤的話，在產生候選子樹的步驟上，加入節點的位置也會不同於前序追蹤時是最右路徑，後序追蹤則是要從最左路徑上加入新節點，並且這在後序追蹤序列的表示法上時，新節點是加在最前頭。另外，在 TRIPS 演算法中，還額外增加了一個新的資料結構叫做 Prüfer 序列（Prüfer Sequence），Prüfer 序列是在 1918 年時由 Heinz Prüfer 首度提出來證明 Cayley’s formula [7]。Prüfer 序列提供了 n 個節點標記樹的集合與標記 1～n 所構成的長度 n-2 之序列集合的互相雙射（bijection）。一棵 n 個節點的樹，其 Prüfer 序列可由簡單的迴圈計算得到，初始時是空的序列，然後在每一個步驟時，移除葉節點中有最小標記的節點，並且將該葉節點的父節點加在已經建立好的部分 Prüfer 序列上。透過簡單的數學歸納法，可以證得一個已知的由標記 1～n 所組成、長度 n-2 的序列 S，會存在有一個唯一的標記樹，其 Prüfer 序列爲 S。

在 TRIPS 演算法中，它使用了節點的後序追蹤數字作爲標記的爲一集合，並且用來建立成 Prüfer 序列，稱爲計數 Prüfer 序列（Numbered Prüfer Sequence，簡稱 NPS）。另外，一棵樹的標記序列（Label Sequence，簡稱 LS）則是由葉節點的標記序列組成。最後所有的序列再由後序數字（post-order nember，簡稱 PON）重新排序。圖 4.1 是說明這個資料結構的例子。



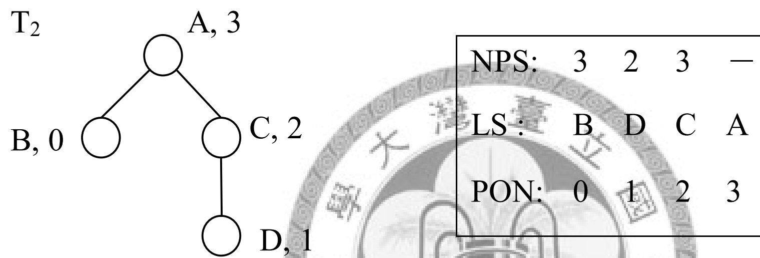


圖 4.1：TRIPS 的資料結果例圖

下面是 TRIPS 演算法的最左路徑候選子樹生成方式，其實和 TreeMiner 演算法的最右路徑產生候選子樹的方式相當類似。

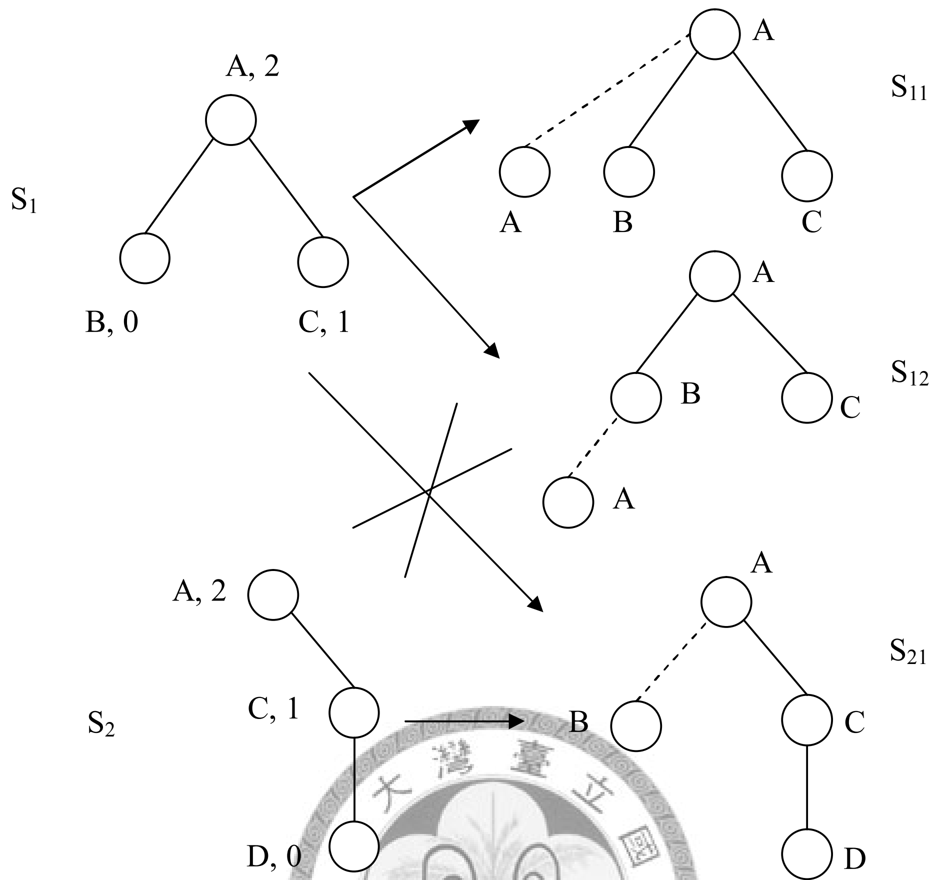
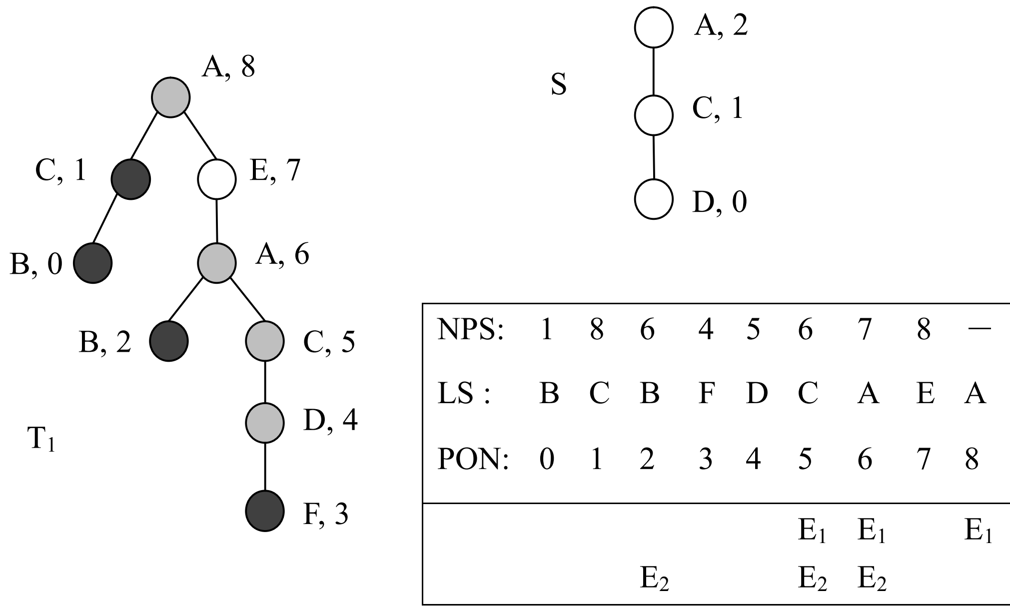
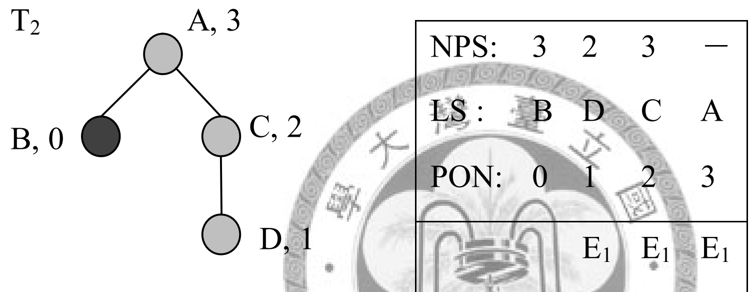


圖 4.2：TRIPS 最左路徑延伸例圖

在圖 4.2 中，S_1 只會生成 S_{11} 和 S_{12}，而無法生成 S_{21}。S_2 才能夠生成 S_{21}。

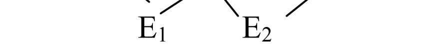
嵌入列表（Embedding List）是 TRIPS 演算法中，用來儲存子樹出現在資料庫樹結構中的狀態，主要是由兩行陣列所構成，然後第 1 行儲存和子樹節點符合的 PON，而第 2 行儲存子樹結構的指標，並且各節點間用-1、-2 區隔開來。在圖 4.3 中，我們會舉個例子詳細說明候選子樹的生成和嵌入列表的內容。





S 在 T_1 中的嵌入列表

8	6	-1	5	5	-1	4	4	-1
-1	-1	-2	0	1	-2	3	4	-2



E_1 E_2

S 在 T_1 中產生的候選子樹生成點

E_1: (C , 2) , (B , 2) , (F , 0)

E_2: (B , 2) , (F , 0)

S 在 T_2 中的嵌入列表

3	-1	2	-1	1	-1
-1	-2	0	-2	2	-2

E_3

S 在 T_2 中產生的候選子樹生成點

E_3:(B , 2)

延伸點的雜湊表

延伸點	支持度
(B , 2)	2
(C , 2)	1
(F , 0)	1

圖 4.3：TRIPS 演算法的做法例圖

子樹 S 在 T_1 中出現 2 次，分別是 E_1(PON：8、5、4)和 E_2(PON：6、5、4)，在 T_2 中出現 1 次(PON：3、2、1)。在 E_1 能夠新加入節點成爲候選子樹的地方是(LS , PON)= (B , 0)、(C , 1)、(B , 2)、(F , 3)，其中(B , 0)、(C , 1)、(B , 2)是接到 S 中的(A , 2)，所以會構成延伸點（Extension point）爲(LS , S 中被連接的 PON) = (B , 2)、(C , 2)，而(F , 3)是接到 S 中的(D , 0)，所以延伸點是(F , 0)。另外在 E2 能夠新加入節點成爲候選子樹的地方(B , 2)和(F , 3)，而這兩個節點形成延伸點爲(B , 2)、(F , 0)。T_2 所包含的 E_3 中能新加入成爲候選子樹的地方只有(B , 0)，而(B , 0)的延伸點爲(B , 2)。在用來計算支持度的雜湊表（hash table）中，延伸點(B , 2)在 T_1 與 T_2 都出現過，所以支持度爲 2，而(C , 2)和(F , 0)都只有在 T_1 中出現而已，所以支持度只有 1。

底下我們會詳細介紹 TRIPS 的演算法部分。

TRIPS 演算法：

演算法 1：子樹探勘演算法

要求參數： D = {T_1 , T_2 , …… , T_N} , minsup

1: F_1 = readTrees (D)

2: **for** 所有 F_1 中的節點 **do**

3: minTrees (NULL , (v , -1) , D)

4: **end for**

演算法 2：探勘一個被給予的子樹

minTrees (pat , (l , pos) , tidlist)

1: newpat = extend (pat , l , pos)

2: newtidlist = NULL

3: **for** 所有 tidlist 中的 T **do**

4:     **if** (l , pos)是在 T 中 pat 的延伸點時 **then**

5:         更新 T 的嵌入列表

6:         增加 T 到 newtidlist

7:     **end if**

8: **end for**

9: H = NULL

10: **for** 所有 newtidlist 中的 T **do**

11:     掃瞄 newpat 在 T 中的延伸點

12:     加入新產生的延伸點到 H 中

13: **end for**

14: **for** H 中所有的 h **do**

15:     **if** h.support ≥ minsup **then**

16:         minTrees (newpat , (h.l , h.pos) , newtidlist)

17:     **end if**

18: **end for**

在演算法 1 中，輸入參數資料庫 D 和最小支持度 minsup 後，第 1 行的執行的是掃瞄 D 中有出現的頻繁節點，並且回傳至 F_1 中。然後接下來的迴圈便是針對 F_1 中所以節點 v 執行演算法 2，其中的 minTrees (NULL , (v , -1) , D)，因爲尙未產生任何 pat（子樹），所以傳 NULL，而(v , -1)爲傳入 v 節點，-1 表示初始值，v 節點並未接到任何子樹上，D 爲所有樹組成的資料庫。

在演算法 2 中，第 1 行的 newpat = extend (pat , l , pos)，是表示延伸舊的 pat（子樹）成爲新的 newpat（新子樹），l 代表新增節點的標記，pos 代表新增節點接上 pat 的 PON 位置。第 2 行爲初始化 newtidlist，然後在第 3 行時判斷 D 中的 T 中是否存在有 newpat，如果有的話更新 T 原本的 pat 的嵌入列表成爲 newpat 的嵌入列表，並且把 T 加入到 newtidlist 中。第 9 行是初始化雜湊表 H。第 10～12 行，是針對在 newtidlist 中的樹 T，掃瞄在包含 newpat 下時可能會出現的延伸點，然後把延伸點加入到雜湊表 H 中，並計算支持度。然後在第 14～16 行中，便會針對雜湊表 H 各延伸點的支持度是否有超過 minsup，如果有才利用遞迴方式再執行演算法 2。

在這個 TRIPS 演算法中，有兩個地方運用到了反單調特性，第一個就是第 3～6 行中，在 newtidlist 中可以說是去除了其他不包含 newpat 的樹 T，所以減少了之後所要搜尋的空間。而在第 14～16 行的部分，就跟一般樹探勘演算法一樣，去除掉不頻繁的候選子樹。另外，在第 4、11 行的部分，TRIPS 的資料結構也大大加快了執行速度，這也就是爲何 TRIPS 演算法會比 TreeMiner 演算法還快了。

4.2 TRIPS 演算法的改進做法

由於 TRIPS 演算法是比較泛用的做法，可以針對頻繁、有根、標記或無標記、歸納或嵌入、有序或無序子樹作樹探勘工作，所以我們可以針對其中特定的子樹類型，進行改進加快執行速度。在這裡我們便要解說本論文所提出的改進做法。

4.2.1 針對歸納、有序子樹的改進做法

在原本的 TRIPS 演算法中，如果是要針對歸納、有序子樹進行樹探勘工作的話，就是在演算法 2 中的第 3、4 行部分與第 10、11、12 行的迴圈中，只針對父節點與子節點的關係確認，而沒有延伸到祖先節點間的關係確認，不過就原本的資料結構中所提供的資訊而言，仍然必須進行一些多餘的步驟，因此將會造成時間上的浪費。

在本篇論文中，我們針對歸納、有序子樹的改進做法，便是除了原本資料結構中計數 Prüfer（NPS）序列所提供的父節點資訊外，額外再增加一個 LCB 序列。這個 LCB 序列中，我們會儲存一個樹中每個節點的左兄弟節點位置，而如果沒有左兄弟節點時時，則是填入-1 來代表。圖 4.4 便是改良後的資料結構範例。

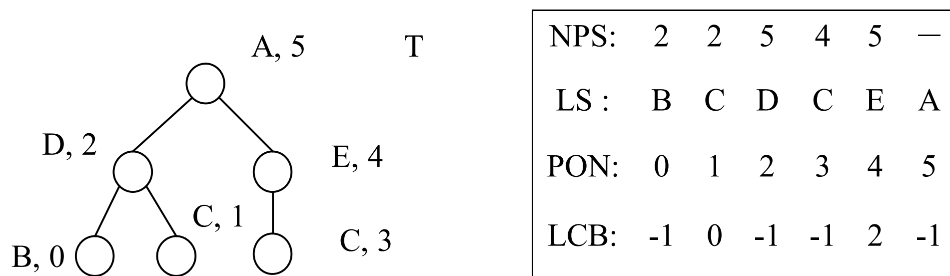


圖 4.4：改良後對歸納、有序子樹的資料結構例圖

就如同原本的 TRIPS 演算法一樣，LCB 序列也可以很容易的在建構樹的資料結構時計算出來。要算出 LCB 序列的話，只要透過 NPS 序列針對各節點的表格往左邊尋找第 1 個兄弟節點便可得到。

圖 4.5 中，我們會講解一下原本 TRIPS 做法與改良做法的說明與比較：

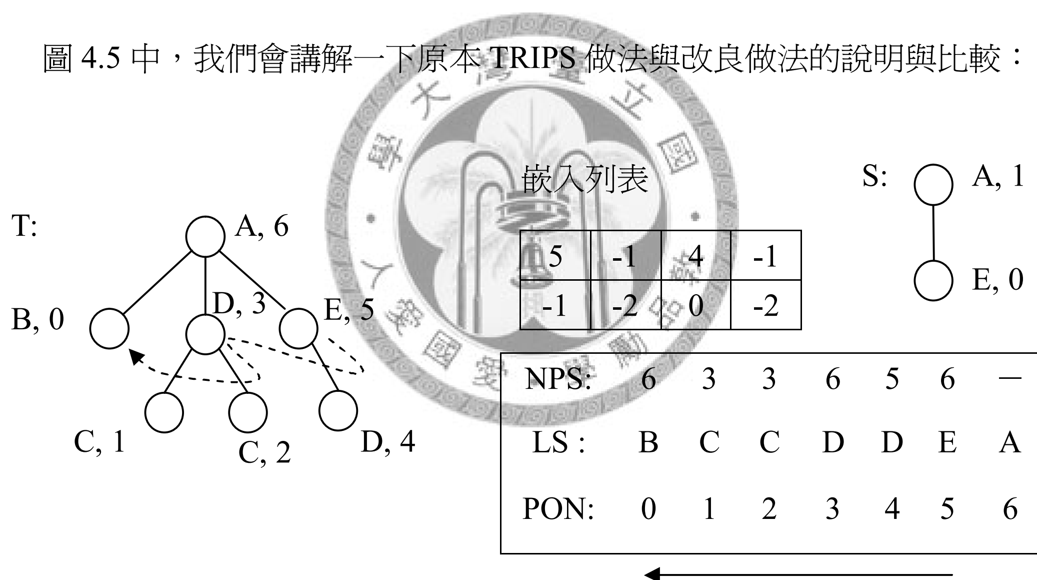


圖 4.5：TRIPS 對歸納、有序子樹的做法說明

在圖 4.5 中，如果目前的歸納子樹為 S，原本 TRIPS 演算法中在尋找候選子樹的延伸點時，將會依序沿著(D , 4)開始往資料結構表格中的左邊尋找到(B , 0)，其中合法的延伸只有(D , 4)、(D , 3)、(B , 0)三點，而(C , 1)、(C , 2)則不是歸納子樹的合法延伸點。因此舊的做法便會多跑兩個點。

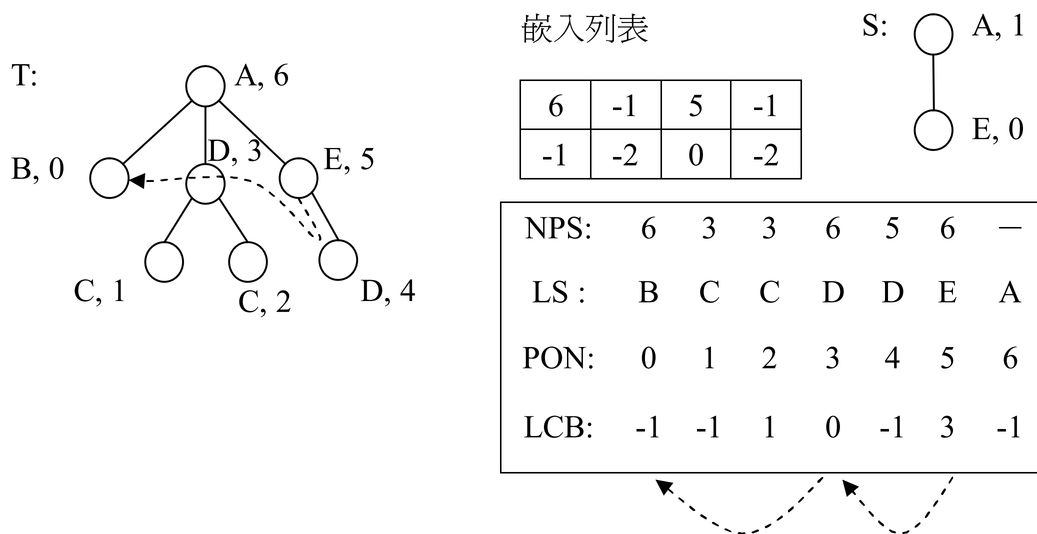


圖 4.6：對歸納、有序子樹的改良做法說明

在圖 4.6 中，利用新擴充的資料結構，於新做法裡當我們尋找子樹 S 中(E , 0)的候選子樹延伸點時，便可以從原本 T 中的(D , 4)點開始尋找，這時會先加入(D , 4)點，然後再查詢(D , 4)點的 LCB 繼續尋找，這時會遇到-1，所以離開。接著再尋找子樹 S 中(A , 1)的候選子樹延伸點時，就可以利用嵌入列表中上一個點所對應到樹 T 中的點(E , 5)開始，尋找(E , 5)的左兄弟節點(D , 3)，以及尋找(D , 3)的左兄弟節點(B , 0)。因此這樣便可以不用尋找到(D , 3)的兩個子節點了。

這裡我們稍為評估一下兩種做法的差異，TRIPS 演算法的做法在最佳狀況底下，所會尋找的節點頂多和新做法一樣，不過若是在最差情況底下，就一棵高度 K 的完滿二元樹來說，在一次候選子樹的延伸點搜尋中，將會多尋找$(2^k - 1) - (2k + 1)$個節點。而在下一章中，我們會有兩種做法針對實際資料的實驗比較。

4.2.2 針對歸納、無序子樹的改進做法

延伸在上一小節中所提出的改良做法，要把改良做法改爲針對無序子樹的話，很簡單的只要在資料結構上再加入右兄弟節點的資訊即可，然後進行歸納子樹的候選子樹延伸點尋找時，除了針對歸納子樹的樹葉節點進行的延伸點尋找外，其他內部節點的延伸點尋找都可透過左兄弟節點和右兄弟節點兩個資訊從兩方向，尋找出無序子樹可能的延伸點。另外，在進行這類延伸點的尋找時，也必須要像 SLETCH 演算法一樣加入是否符合標準式延伸（Canonical Extension）的判斷，以避免重複產生同構的候選子樹。

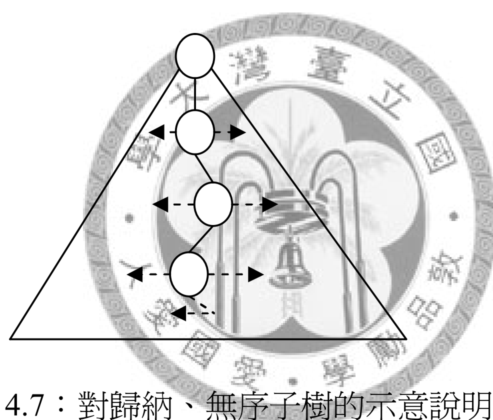
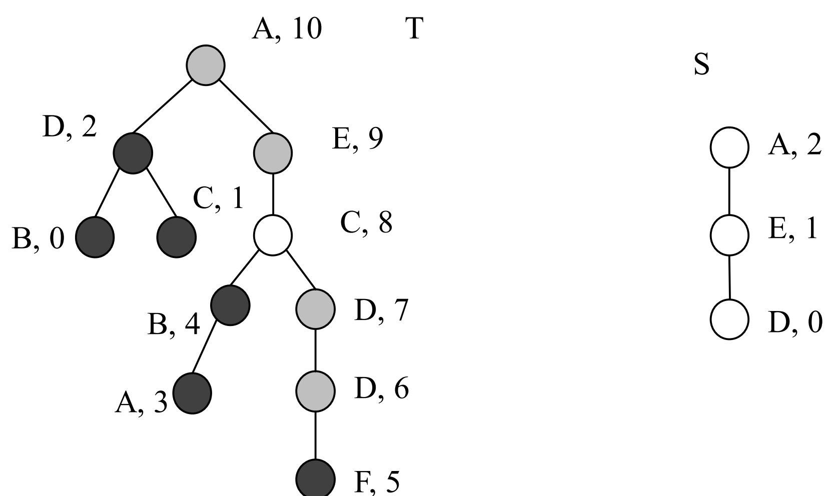


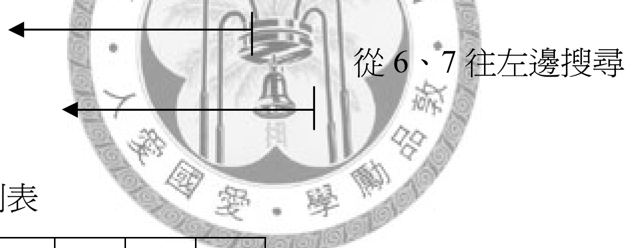
圖 4.7：對歸納、無序子樹的示意說明圖

4.2.3 針對嵌入、有序子樹的改進做法

在 TRIPS 演算法的資料結構中，計數 Prüfer 序列以及標記序列 LS 能夠幫助 TRIPS 演算法 2 中的第 3、4 行和第 10、11、12 行的迴圈加快速度，原因在於標記序列 LS 是依照後序追蹤序列的順序排列的，所以在儲存 pat 和 newpat 嵌入列表的陣列中最後一段存有節點對應的區塊間，可以直接根據節點所對到的位置，再往 LS 的左邊進行搜尋符合的節點。圖 4.8 是我們的舉例說明。



NPS:	1	2	10	4	8	6	7	8	9	10	－
LS :	B	C	D	A	B	F	D	D	C	E	A
PON:	0	1	2	3	4	5	6	7	8	9	10
							E_1			E_1	E_1
								E_2		E_2	E_2



S 在 T_1 中的嵌入列表

10	-1	9	-1	7	6	-1
-1	-2	0	-2	2	2	-2

嵌入列表最後一個區點有兩個節點，代表 S 在 T 中出現 2 次。

圖 4.8：TRIPS 對嵌入、有序子樹的做法說明

所以在圖 4.8 中，根據嵌入列表最後兩個節點的位置 6 和 7，往 LS 的左邊尋找即可，然後再根據 NPS 提供的資料，判斷 POS 是否正確或者是所要接到的點、何時要停止搜尋等。

但是這個資料結構的設計，在針對嵌入子樹的某些情況下會比較花費時間，像是在 TRIPS 演算法 2 中的第 4 行尋找(l , pos)是否為 T 中的延伸點時，若是以之前的範例作說明 l = B , pos = 2，那麼我們從資料結構的 PON 位置 6、7 往前搜尋時，會在 PON4 的位置上發現到相符的標記 B，但這時就必須要測試 B 點是否有連接到子樹 S 中的 (A , 2) 節點所對應到 T 中的(A , 10)節點，或者是在那之前先連接到其他子樹 S 其他節點所對應到的節點上，例如 S 中的(E , 1)這個點對應到 T 中的(E , 9)這個節點上。

所以本論文便針對這點，從資料結構上進行了改進，就是將原本的計數 Prüfer 序列（NPS），更換成 LSN 序列。這個 LSN 序列所存放的值，是每個節點以自己為根的子樹所構成的後序追蹤序列的第一個節點，其在原本樹的後序追蹤序列中的前面一個節點，如果不存在時則以-1 帶入。圖 4.9 是個說明的例子。

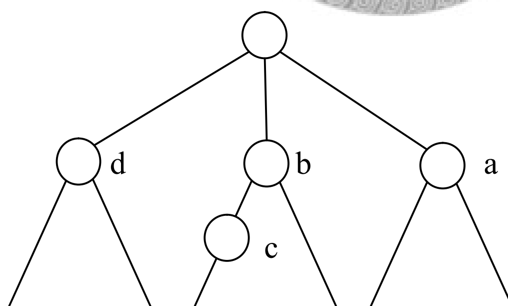
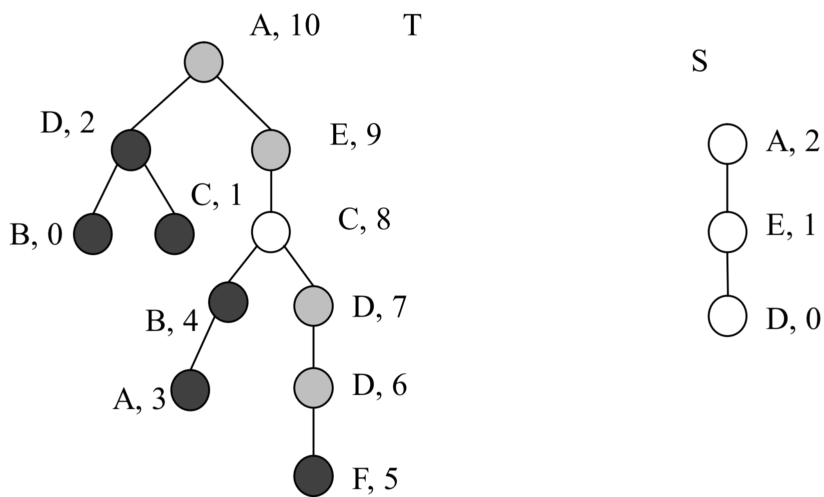


圖 4.9：新資料結構的特性說明

在圖 4.9 中，對 a 點來說就是 b 點，對 b 點來說就是 d 點，而對 c 點的話也是 d 點。另外 d 點則不存在，所以就放入-1。這樣的話，再加上嵌入列表後，就可以很快標定出 TRIPS 演算法 2 中，在資料結構上所要進行搜尋的區間了。



LSN:	-1	0	-1	2	2	4	4	4	2	2	-1
LS :	B	C	D	A	B	F	D	D	C	E	A
PON:	0	1	2	3	4	5	6	7	8	9	10
							E_1			E_1	E_1
								E_2		E_2	E_2

從 2 往左邊搜尋

S 在 T_1 中的嵌入列表

10	-1	9	-1	7	6	-1
-1	-2	0	-2	2	2	-2

S(A , 2)的前一個點(E , 1)

圖 4.10：對嵌入、有序子樹的改良做法說明

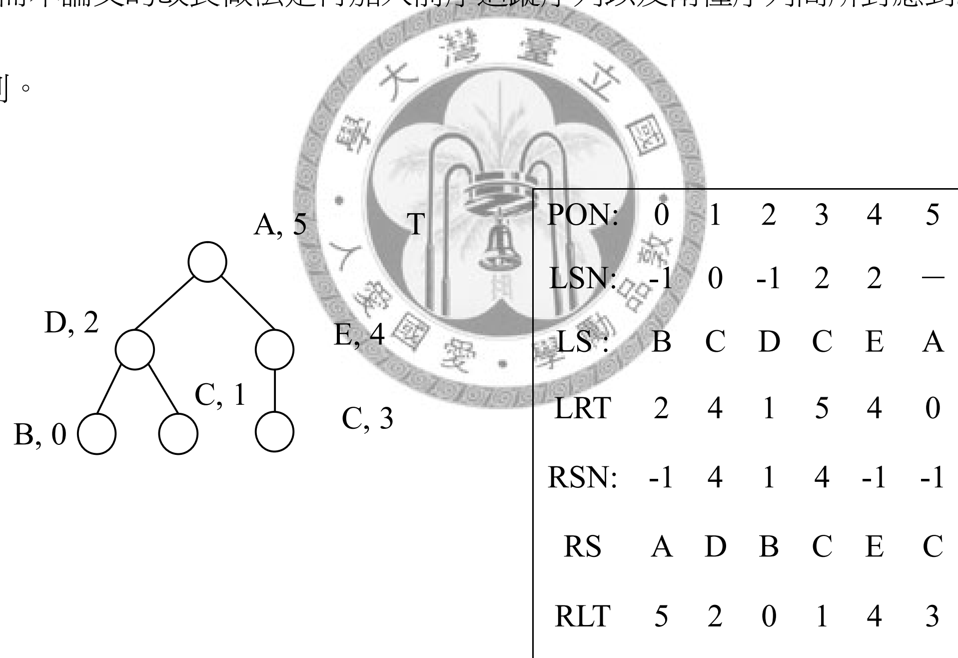
在圖 4.10 中，如果新加入的節點(l , pos) = (B , 2)，那就透過嵌入列表中，尋得 S 的(A , 2)的前一個點(E , 1)所對應到在 T 中的(E , 9)，然後再查詢 LSN 值為 2，所就直接從 PON = 2 的部分往左搜尋即可。

運用新的資料結構後，可以在演算法 2 的提升執行時間，因為可以直接標定

所要尋找的節點區間，並且省去 TRIPS 演算法中確認節點連接的工作，而在下一章中我們會列出針對實際資料的實驗結果。另外，新資料結構的計算，同樣可以透過 LS 和 NPS 事先得到，所以只要計算一次便可。

4.2.4 針對嵌入、無序子樹的改進做法

原本的 TRIPS 演算法在某些步驟上稍微改寫一下的話，仍然是可以探勘頻繁、有根、嵌入、無序子樹的，不過因為後序追蹤序列的特性，所以會比較花時間。而本論文的改良做法是再加入前序追蹤序列以及兩種序列間所對應到的點之序列。



PON:	0	1	2	3	4	5
LSN:	-1	0	-1	2	2	—
LS :	B	C	D	C	E	A
LRT	2	4	1	5	4	0
RSN:	-1	4	1	4	-1	-1
RS	A	D	B	C	E	C
RLT	5	2	0	1	4	3

圖 4.11：對嵌入、無序子樹的改進資料結構例圖

在圖 4.11 中，RS 為前序追蹤序列；RSN 則是與 LSN 的意義相同，不過是針對 RS 序列儲存的資料；LRT 為 LS 序列中節點對應到 RS 序列時的位置；RLT 則是 RS 序列中節點對應到 LS 序列的位置。

底下我們會有個簡短的證明，說明採取混合的方式會比單一序列還要快。

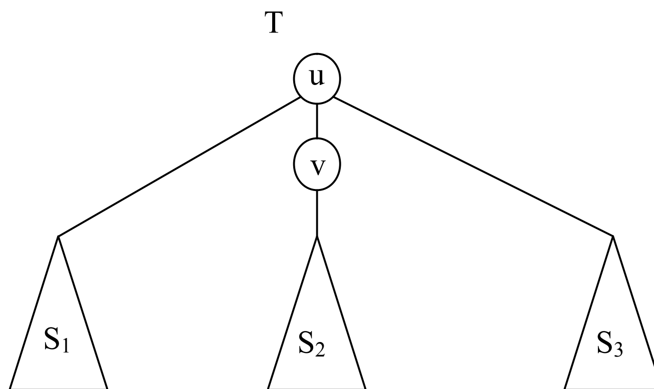


圖 4.12：對嵌入、無序子樹的證明例圖

證明：

在圖 4.12 中，假設 u , v 是在嵌入列表中所對應到的子樹，則當我們要搜尋新加入的節點時，S_1 和 S_3 是可以互換，S_2 因爲是接在 v 底下，所以不需考慮無右搜尋的話，序子樹的影響。而 T 的後序追蹤序列爲 $S_1S_2vS_3u$，所以如果從 S_2 的根往右掃瞄的話，就必定會碰到 v 點。另外如果是從 u 點往前掃瞄的話，更會多經過 v 和 S_2。但如果事前序追蹤序列 $uS_1vS_2S_3$，從 S_2 的根往右馬上就接到 S_3 了，所以便可以省去進行掃瞄到的節點是不是在 S_3 中的判斷。

另外，我們在進行針對無序子樹的探勘時，在產生候選子樹的步驟上，依舊要像 SLETCH 演算法一樣加入是否符合標準式延伸（Canonical Extension）的判斷，例如從上面的例子中，我們在 S_1 和 S_3 中進行搜尋時，合法能加入的節點其標記一定要大於或等於 v 節點。

以上的改進做法，其實與 TRIPS 演算法的一些概念相符，是在建立儲存一棵樹的資料結構時，把一些額外對演算法有幫助的資訊，事先儲存起來，這樣在之後執行演算法時，便可以利用這些以儲存的資訊減少計算步驟。而這樣的做法在第一次建立樹的資料結構時，雖然會花費比較多的時間，不過之後如果樹探勘演算法的步驟會重複多次執行的話，就可以節省不少時間了。在下一章中，我們會放上實驗數據來驗證這個改進的結果。

第五章

實驗結果與分析

在本章中，主要會列出本篇論文進行實驗的環境與數據，還有針對實驗結果進行分析。

5.1 實驗環境

本論文的實驗平台爲 Windows XP 作業系統，撰寫程式的環境爲 Microsoft Visual Studio 2005，使用的語言爲 C++。硬體平台爲普通的 PC，而 PC 的規格如下：

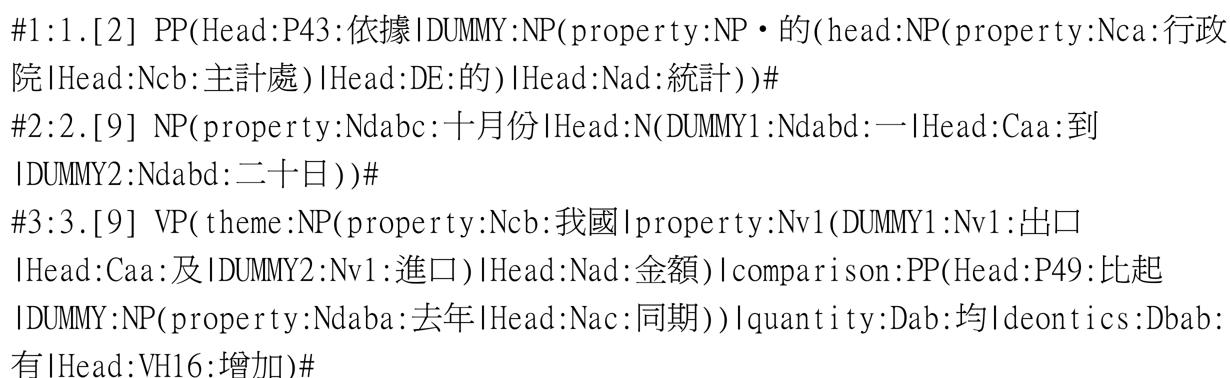
中央處理器	AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ 2.01GHz
記憶體	512MB DDR-400 SDRAM
硬碟	160GB SATA HDD

本實驗所使用的測試樹資料庫，主要是選擇自三個地方，分別是中央研究院中文詞知識庫小組的中文結構樹資料庫、華盛頓大學的 XML 研究資料集合裡的 TREEBANK 和 Zaki 的 CSLOGS。然後底下我們會提出根據這三樣實驗數據進行的實驗方法和所執行的結果。

5.2 實驗方法與結果

5.2.1 中文結構樹資料庫

本實驗中所使用到的中文結構樹資料庫，是取用自中央研究院中文詞知識庫小組所建好的樹資料庫，所使用的資料庫中總共有 36932 棵中文樹結構，然後經過轉換程式轉變成實驗所使用的結構，然後再執行樹探勘演算法的程式。底下是擷取自中央研究院中文詞知識庫小組所建立好原始樹資料的一部份內容：



```
#1:1.[2] PP(Head:P43:依據|DUMMY:NP(property:NP‧的(head:NP(property:Nca:行政院|Head:Ncb:主計處)|Head:DE:的)|Head:Nad:統計))#
#2:2.[9] NP(property:Ndabc:十月份|Head:N(DUMMY1:Ndabd:一|Head:Caa:到|DUMMY2:Ndabd:二十日))#
#3:3.[9] VP(theme:NP(property:Ncb:我國|property:Nv1(DUMMY1:Nv1:出口|Head:Caa:及|DUMMY2:Nv1:進口)|Head:Nad:金額)|comparison:PP(Head:P49:比起|DUMMY:NP(property:Ndaba:去年|Head:Nac:同期))|quantity:Dab:均|deontics:Dbab:有|Head:VH16:增加)#
………………………..
```

其中，由兩個「#」所包圍起來的部分便代表一棵樹，一個節點底下用括號括起來的部分則是代表子節點的部分，而以「|」區隔開的則是兄弟節點，以 : 區隔開的則只是一些詞性說明等部分。

底下是經過轉換後所輸出的結果。

```
T1,PP,P43,$,NP,NP,NP,Nca,$,Ncb,$,$,DE,$,$,Nad,$,$
T2,NP,Ndabc,$,N,Ndabd,$,Caa,$,Ndabd,$,$
T3,VP,NP,Ncb,$,Nv1,Nv1,$,Caa,$,Nv1,$,$,Nad,$,$,PP,P49,$,NP,Ndaba,$,Nac,$,$,$,Dab,$,Dbab,$,VH16,$
T4,VP,Cbca,$,Daa,$,NP,NP,NP,NP,Ndabc,$,$,Caa,$,NP,Ndabc,$,Ndabd,$,$,$,DE,$,$,VH12,$,Nad,$,$,Dbb,$,PP,P49,$,
NP,Ndaba,$,Nac,$,$,$,VH16,$,Di,$,NP,Neqa,$,$
................
```

因爲樹探勘工作應用在語言學的研究上，主要是使用在詞句結構的分析等方面上，所以這裡針對中文樹結構轉換出來的結果，是挑選出詞類的標記，並且去除掉其他不必要的部分。經過轉換之後，每棵樹都是以前序追蹤的順序編碼，然後其中加上回溯符號「$」。

樹探勘工作應用在中文樹結構的詞句結構分析上，主要所會探勘的資訊是屬於有序樹的方面，因爲文句的順序在結構中是不能隨便掉換的。底下我們便有針對歸納、有序樹和嵌入、有序樹執行的實驗結果與分析。另外，也有用在測試上來針對嵌入、無序子樹執行的實驗結果。

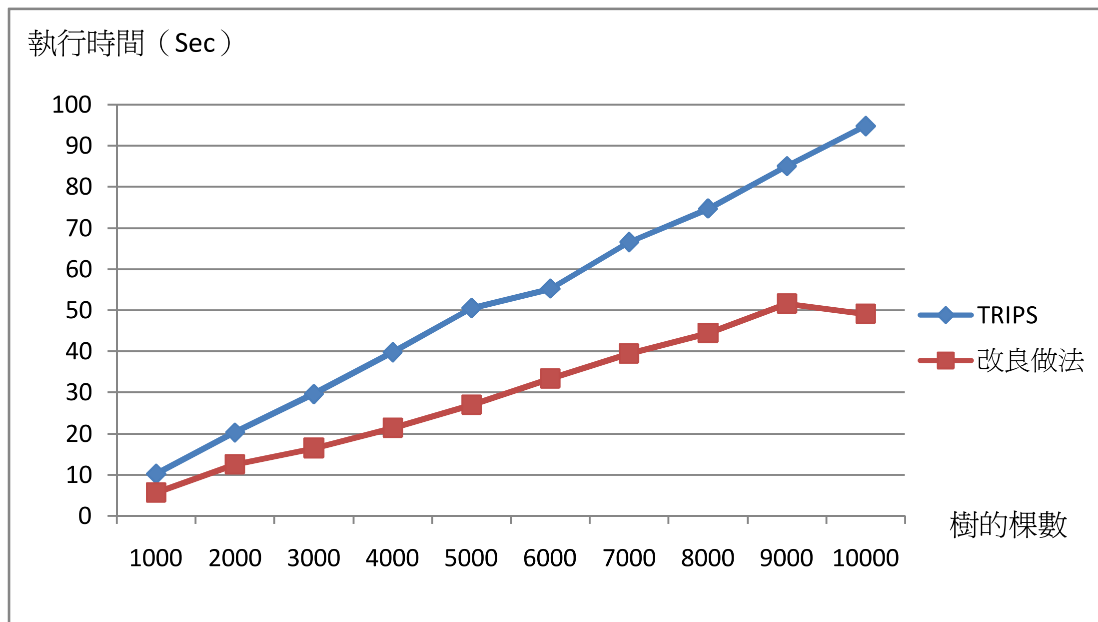


圖 5.1：支持度在 10%下不同棵數的樹對歸納有序子樹之執行時間比較

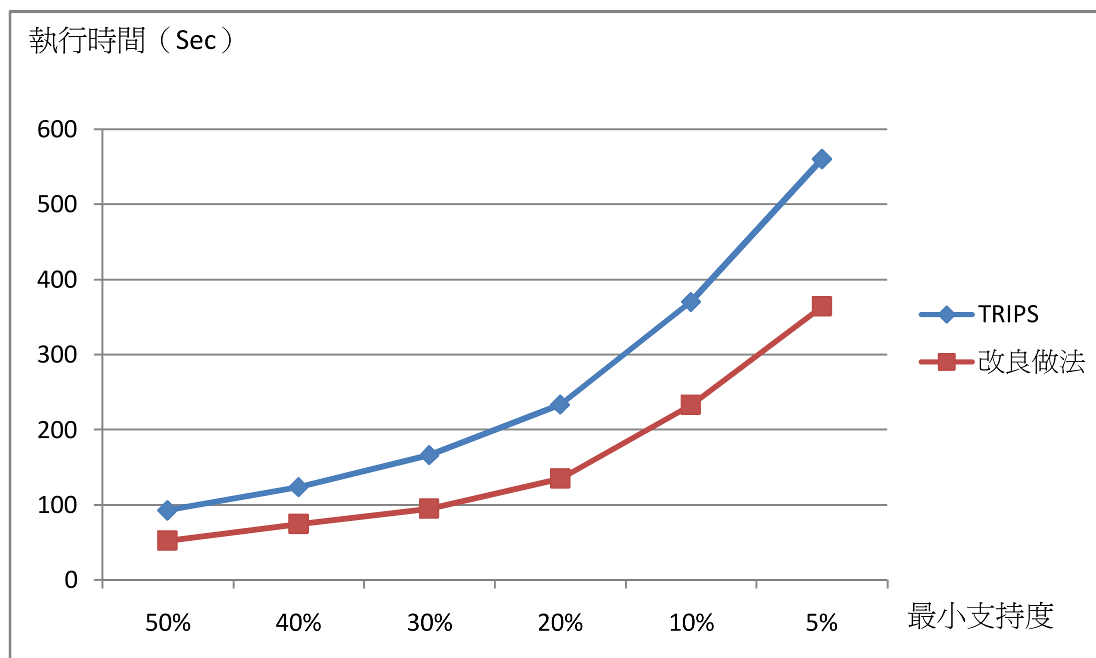


圖 5.2： 35000 棵樹對歸納有序子樹在不同最小支持度下的執行時間

圖 5.1 和圖 5.2 是針對歸納、有序子樹做出來的結果。圖 5.1 是在固定最小支持度底下，變更不同棵樹所做出的結果。圖 5.2 是針對中文樹結構資料庫的 35000 棵樹，進行不同支持度下執行時間的長短評估。由於新做法中預先計算了一些資訊而減少了執行樹探勘工作的計算量，所以在執行時間上比舊的做法要快上一些，而在圖 5.1 中 10000 棵樹在 10%最小支持度底下的執行時間會快過 9000 棵樹在 10%最小支持度底下的執行時間，可能原因應該是 10000 棵樹時，所產生的候選子樹數量比較少，所以造成搜尋空間減少使得執行時間反而快過 9000 棵樹時的情況。

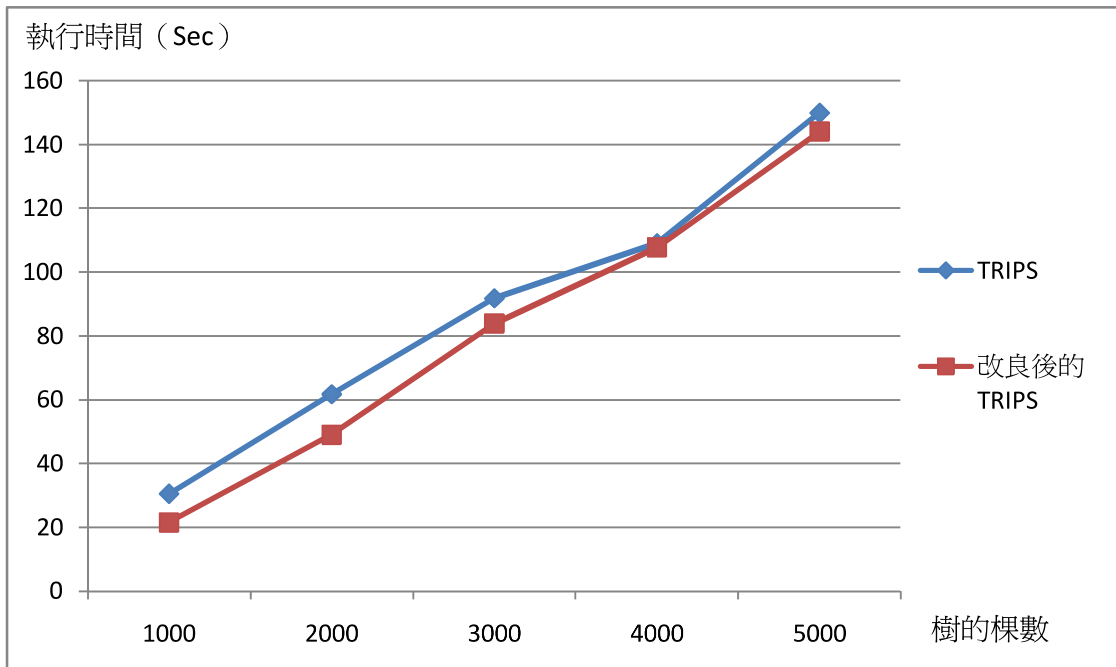


圖 5.3：在支持度 10%下各數量的樹對嵌入有序子樹之執行時間比較

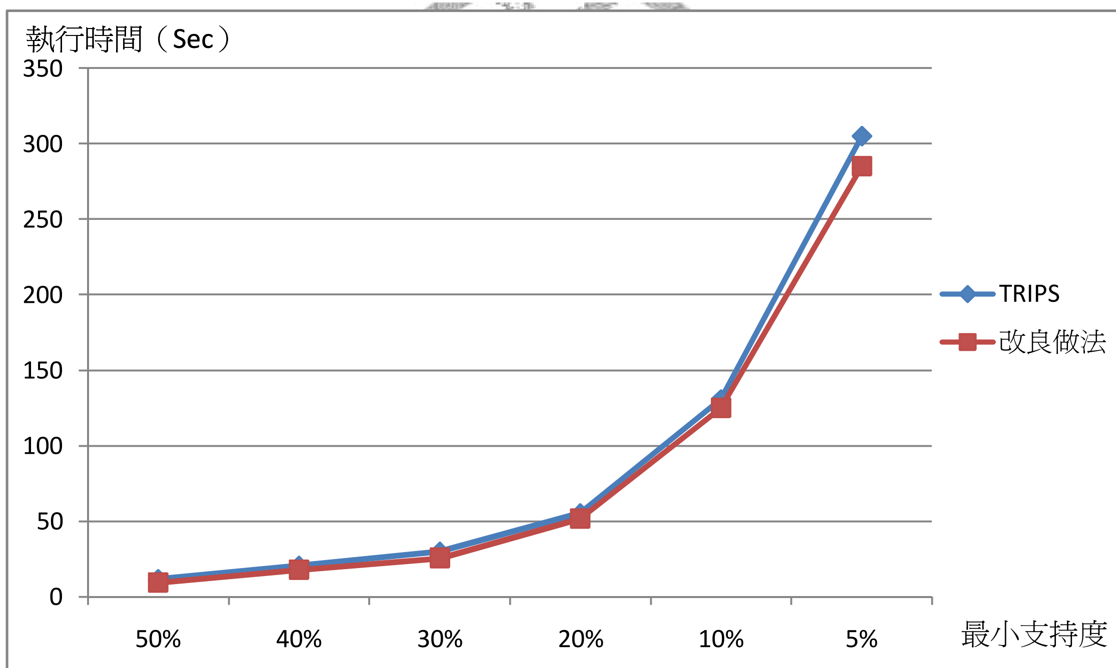


圖 5.4：5000 棵樹在不同最小支持度下對嵌入有序子樹的執行時間

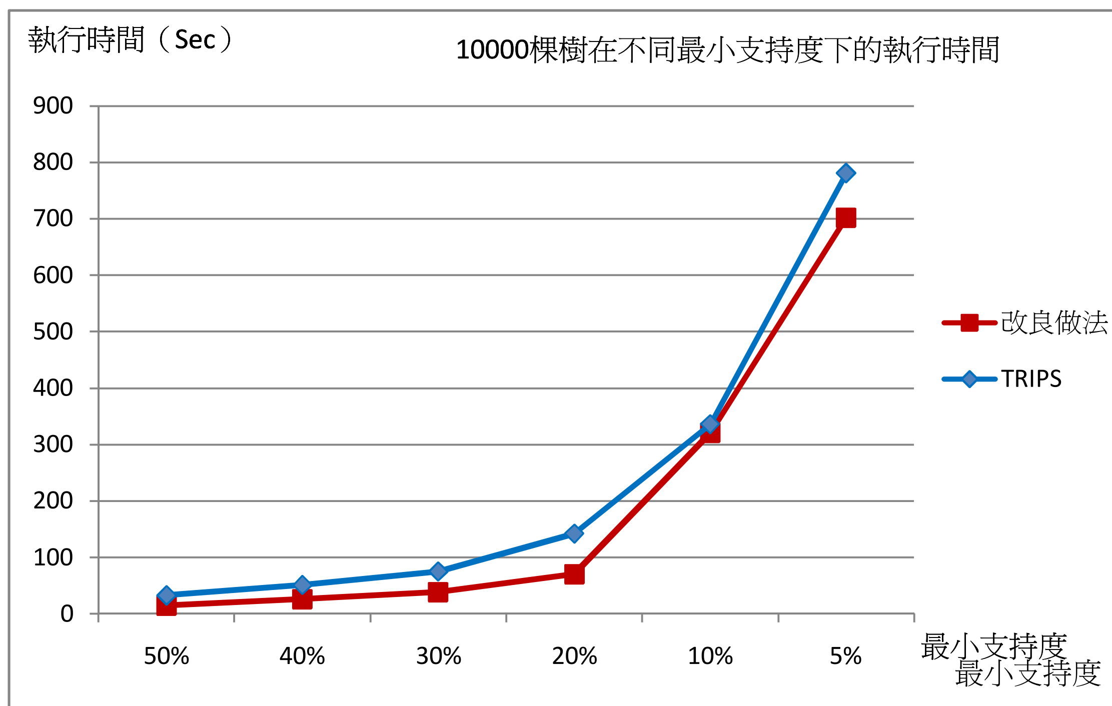


圖 5.5：10000 棵樹在不同最小支持度下對嵌入有序子樹的執行時間

圖 5.3、圖 5.4 和圖 5.5 是針對嵌入、有序子樹做出來的結果。圖 5.3 是在固定最小支持度底下，改變樹的數量所做出的結果。圖 5.4 和圖 5.5 是分別針對中文樹結構資料庫的前 5000 棵樹和前 10000 棵樹，進行不同支持度下執行時間長短的實驗。跟上面相同由於新做法中預先計算了一些探勘時的必要資訊，所以減少了執行樹探勘工作的計算量，因此執行時間比舊做法要快上一點。

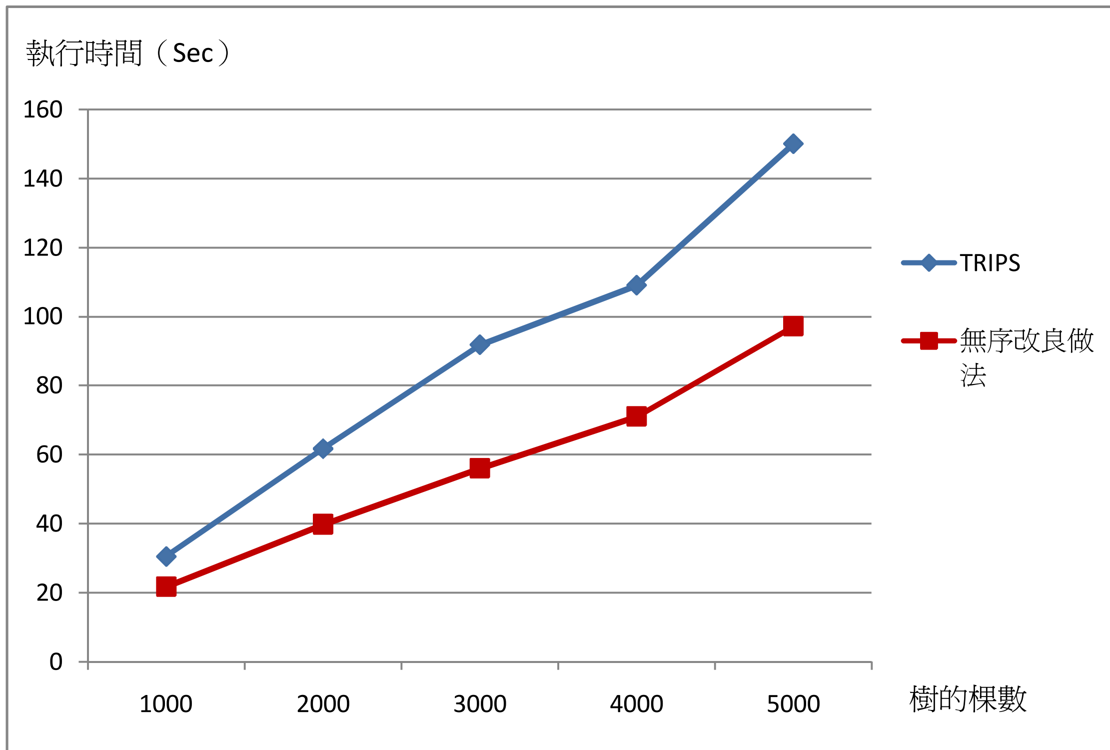


圖 5.6：支持度在 10%時不同棵數的樹對嵌入無序子樹之執行時間比較

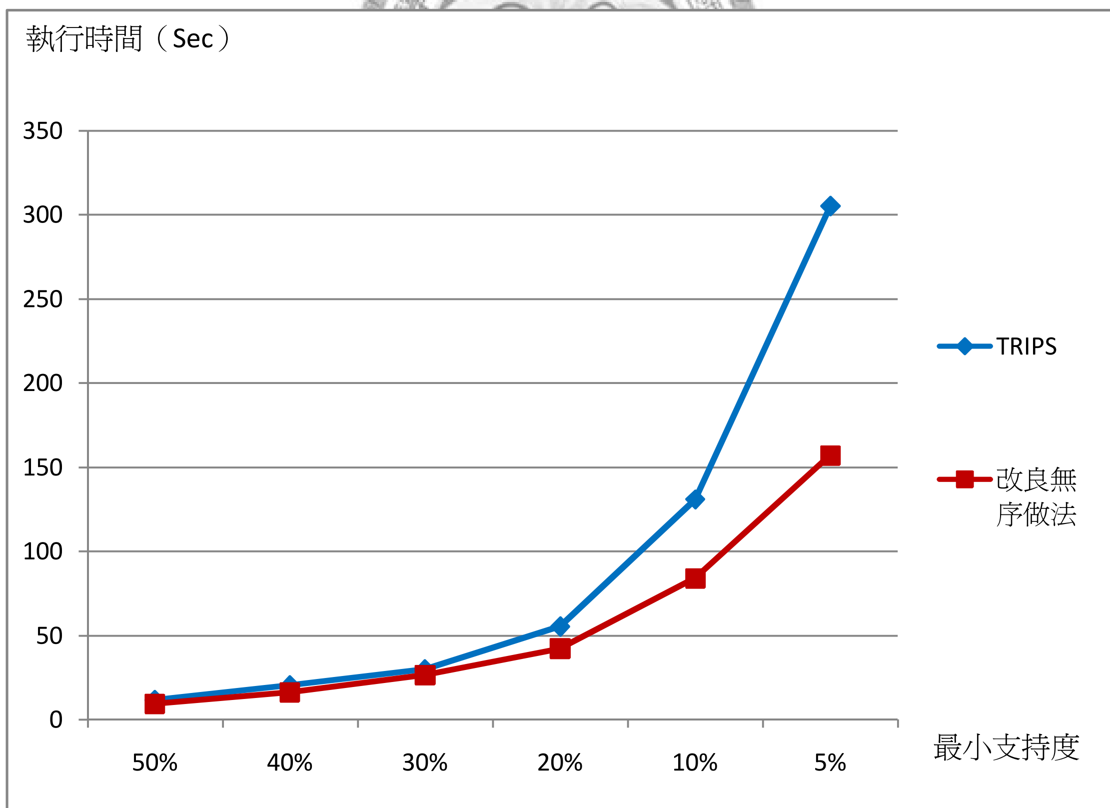


圖 5.7：5000 棵樹在不同最小支持度下對嵌入無序子樹的執行時間

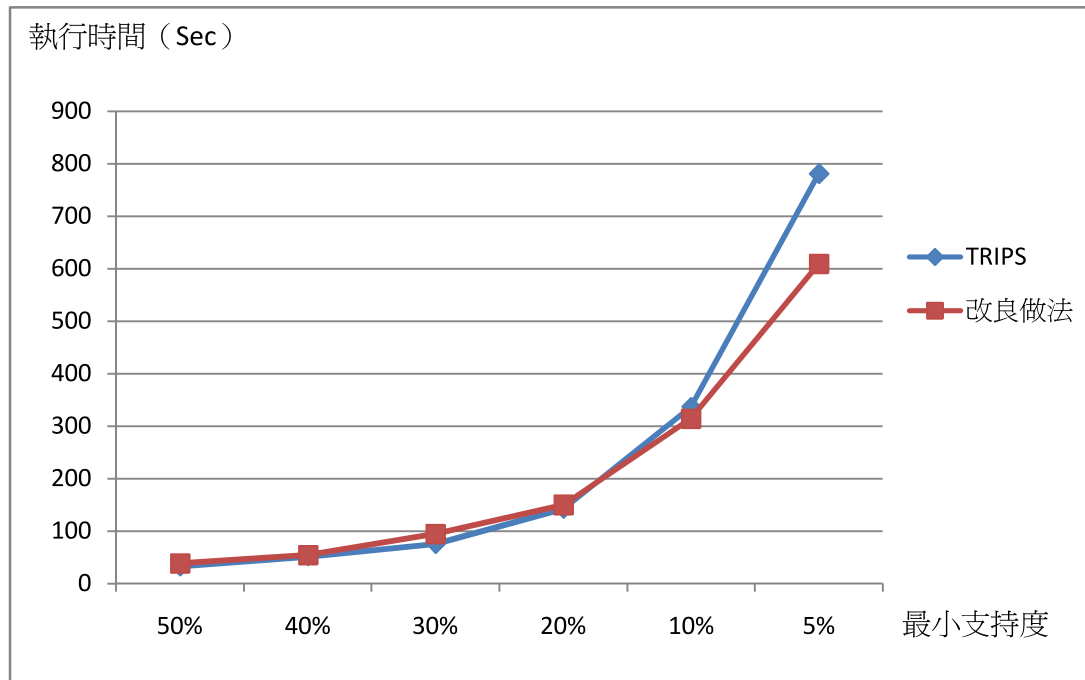


圖 5.8：10000 棵樹在不同最小支持度下對嵌入無序子樹的執行時間

圖 5.6、圖 5.7 和圖 5.8 是針對頻繁、有根、嵌入、無序子樹的改良 TRIPS 做法。其中圖 5.6 是在支持度 10%下不同數目棵數的實驗結果，圖 5.7 和圖 5.8 分別是在下，不同最小支持度下的實驗結果。因爲在[5]中，TRIPS 並沒有實作出針對無序子樹的做法，所以這裡的實驗結果是跟原本 TRIPS 對有序子樹下執行時間比較。而如何判斷這個做法在針對嵌入、無序子樹下是否比較快，主要是依據[5]以及[6]中的實驗結果。在[6]中 SLEUTH 演算法針對嵌入、無序子樹的結果差不多是略慢於 TreeMiner 演算法針對嵌入、有序子樹的結果。而在[5]中，TRIPS 演算法針對嵌入、有序子樹的結果則是快於 TreeMiner 演算法針對嵌入、有序子樹的結果。所以依此推論本論文的改良做法會比 SLEUTH 演算法還快。

5.2.2 CSLOGS

Zaki 的 CSLOGS 包含了壬色列理工學院（Rensselaer Polytechnic Institute）計算機科學系收集超過 1 個月的 web logs 資料，想要取得時可以連到網頁 http://www.cs.rpi.edu/~zaki/software/中下載。CSLOGS 的資料構成方式如下：

```
........................................................
493 493 13 228 318 816 1927 -1 1928 -1 -1 -1 798 -1 764 -1
494 494 20 228 318 816 1927 -1 1928 -1 -1 -1 798 -1 764 -1 303 315 816 -1 -1 306 -1
495 495 23 228 318 816 1927 -1 1928 -1 -1 -1 798 -1 764 -1 303 315 816 -1 -1 306 -1 772 798 -1
496 496 26 228 318 816 1927 -1 1928 -1 -1 -1 798 -1 764 -1 303 315 816 -1 -1 306 -1 772 798 -1 773 798 -1
497 497 31 228 318 816 1927 -1 1928 -1 -1 -1 798 -1 764 -1 303 315 816 -1 -1 306 -1 772 798 -1 773 798 -1 774 798 -1 777 -1
498 498 3 1158 7537 -1
499 499 6 1158 7537 -1 2571 8771 -1
500 500 9 1158 7537 -1 2571 8771 -1 2577 4438 -1
501 501 14 1158 7537 -1 2571 8771 -1 2577 4438 -1 2583 9000 9015 -1 -1
502 502 23 1158 7537 -1 2571 8771 -1 2577 4438 -1 2583 9000 9015 -1 -1 2586 4594 -1 4593 -1 4592 -1 4547 -1
503 503 28 1158 7537 -1 2571 8771 -1 2577 4438 -1 2583 9000 9015 -1 -1 2586 4594 -1 4593 -1 4592 -1 4547 -1 2591 4685 -1 4682 -1
504 504 31 1158 7537 -1 2571 8771 -1 2577 4438 -1 2583 9000 9015 -1 -1 2586 4594 -1 4593 -1 4592 -1 4547 -1 2591 4685 -1 4682 -1
8096 8101 -1
……………………………….
```

前兩個數字代表的是此爲第幾棵存取樹（access tree），之後正的數字是表示連結的網頁代號，可以代表節點的標記，-1 則是回溯符號，因此這些存取樹都是以前序追蹤方式編碼而成，所以我們可以很容易的轉換成我們所需要的輸入資料。CSLOGS 總共包含了 59691 棵樹，節點標記種類爲 13361 種，在 CSLOGS 中一個樹的平均有 12.94 個節點，而最大的樹擁有 428 個節點。底下是我們針對 CSLOGS 的實驗結果。

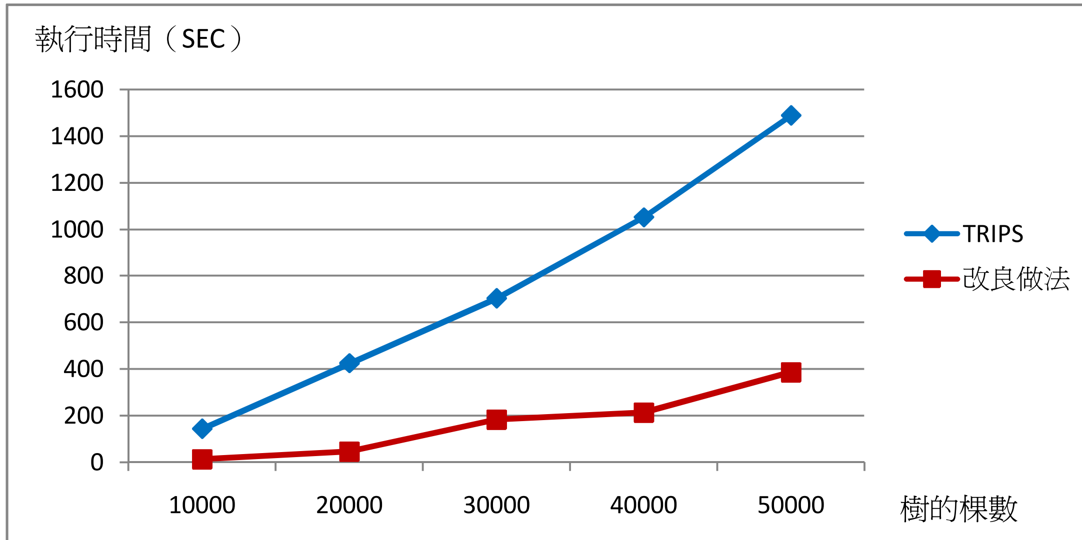


圖 5.9：支持度在 10%時不同棵數的樹對歸納有序子樹之執行時間比較

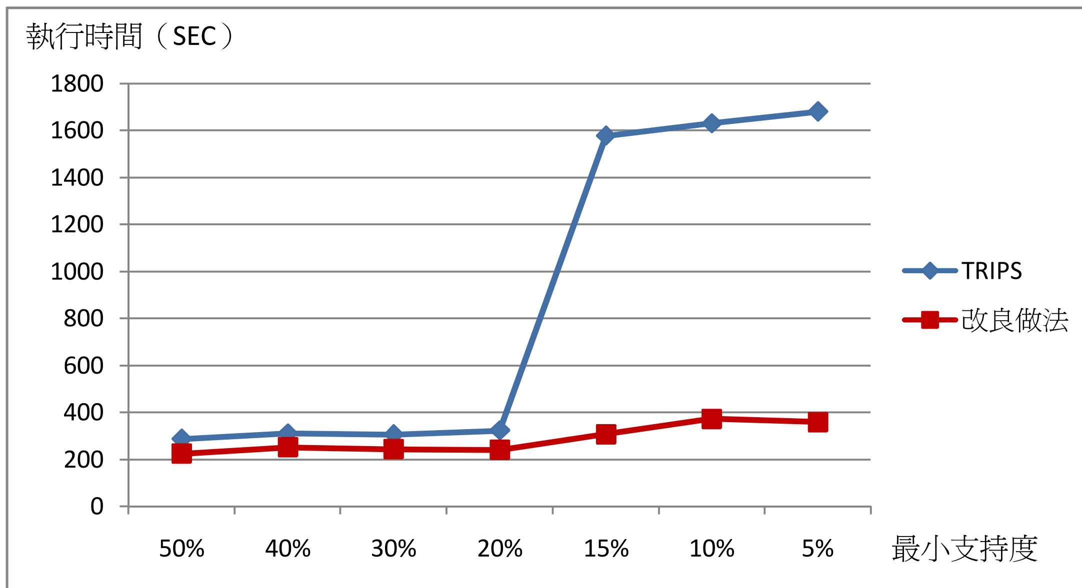


圖 5.10：50000 棵樹在不同最小支持度下對歸納有序子樹的執行時間

圖 5.9 和圖 5.10 是針對歸納、有序子樹做出來的結果。在圖 5.10 中，最小支持度 20%～15%間 TRIPS 的變化相當大，原因是因爲在這兩個最小支持度間，由探勘出來的結果可以得知在接近 15%時會有比較多的候選子樹出現，造成需要搜尋的子樹量較多。

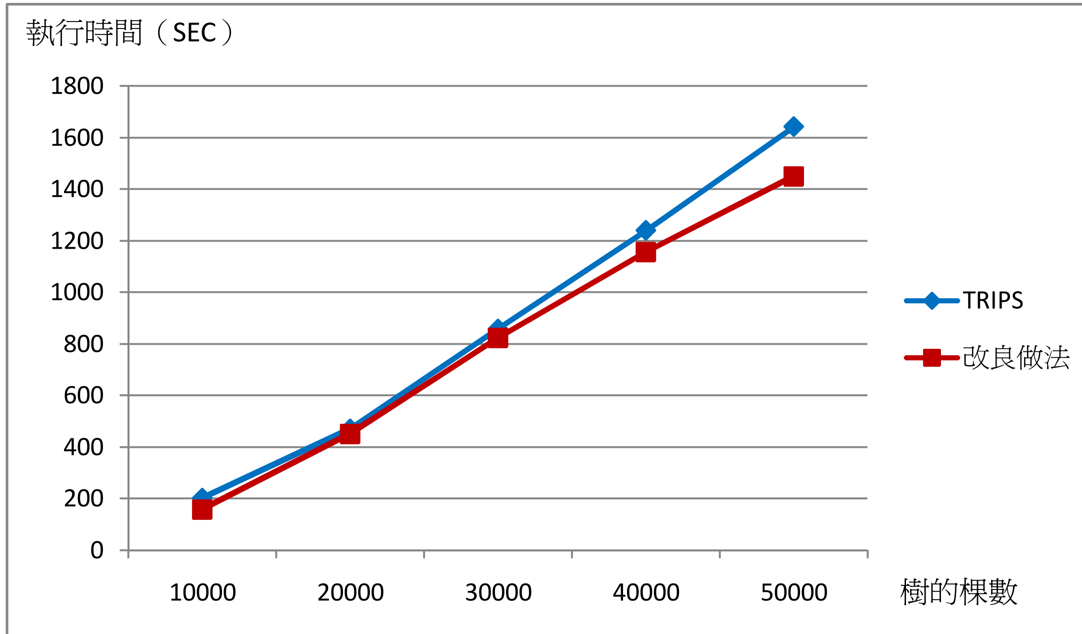


圖 5.11：支持度在 10%時不同棵數的樹對嵌入有序子樹之執行時間比較

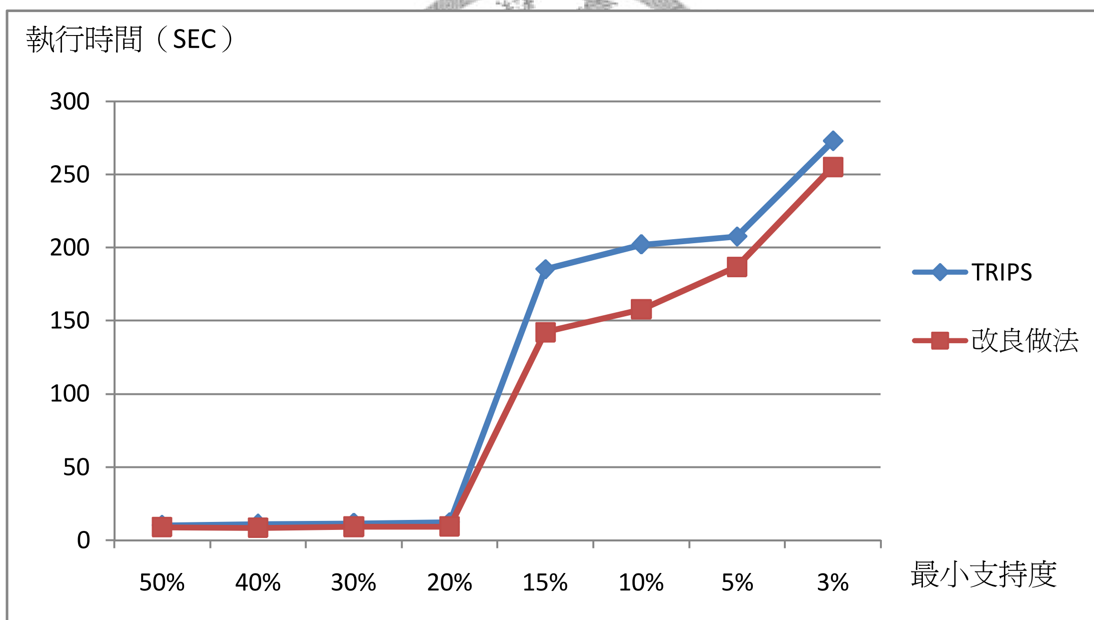


圖 5.12：10000 棵樹在不同最小支持度下對嵌入有序子樹的執行時間

圖 5.11 和圖 5.12 針對嵌入、有序子樹做出來的結果。同樣的在最小支持度小於 15%左右的附近執行時間增加許多，因爲在這裡探勘出的子樹種類增加許多，特別是小於 5%時，更是多了不少。

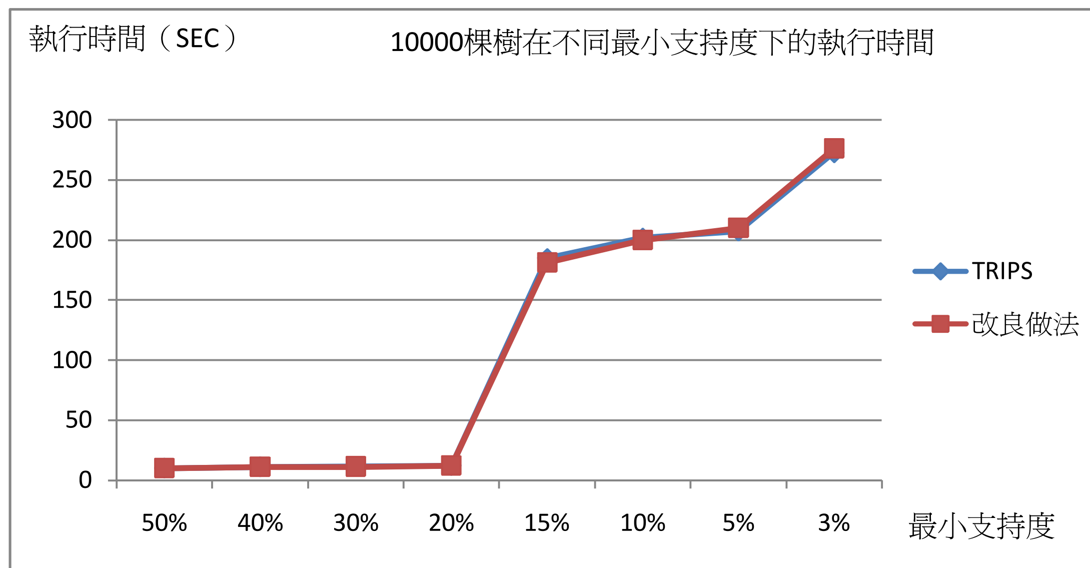


圖 5.13：10000 棵樹在不同最小支持度下對嵌入無序子樹的執行時間

圖 5.13 和圖 5.6、圖 5.7 和圖 5.8 的狀況相同，只有改良做法是針對嵌入、無序子樹的進行探勘，然後根據[5]和[6]來比較。而這裡的結果也和之前圖 5.7 和圖 5.8 的結果類似。

對 CSLOGS 資料庫所進行的實驗，也由於 CSLOGS 資料庫的情況又與之前中文樹資料庫的情況差很多，特別是 CSLOGS 資料庫中節點標記的種類多出許多，所以常常要在很小的最小支持度底下，才能探勘出比較多的子樹，這點也對生成候選子樹的搜尋空間帶來很大的影響。另外還有就是 CSLOGS 資料庫中，樹的結構也比較不同，許多存許樹都是扁平或高瘦，比較少有接近完滿樹（complete tree）的情況，這也對實驗結果造成了影響。

5.2.3 TREEBANK 的 XML 文件

TREEBANK 取自於 http://www.cs.washington.edu/research/xmldatasets 中，TREEBANK 的 XML 文件同樣也是來自於語言方面，主要是根據英文演講稿中的詞句結構所建立出來的。TREEBANK 總共有 52581 個 XML 文件，其中 XML 元素名稱的種類較少並且重複度高，也就是說這代表樹節點的標記種類也會較少且多重複；TREEBANK 中最大的樹有 648 個節點，每棵數的平均節點數為 68.03 個。底下為 TREEBANK 的一部份內容：


```
<EMPTY>
  <S>
   <NP>
    <JJ>Vz+gjytHnSqmoW8pTex59B==</JJ>
    <NN>l333Xdexuj+rLngvm5JwYa==</NN>
    <NNS>0fKDQZ+3ZRIjAo11SUyMQx==</NNS>
   </NP>
   <VBP>ItAvafJCEinspfoHNn8VbR==</VBP>
   <VP>
    <VBN>wtLif6joihOeW/VvC+lIW6==</VBN>
    <NP>
     <NN>CRtAgTzGVllR050ybz8wLM==</NN>
    </NP>
    <S>
     <NP>
      <_NONE_>O9Dgkfn4JDLYMry5bfJG39==</_NONE_>
     </NP>
     <TO>l7qHDSm13CMOXnY+jkrZ6t==</TO>
     <VP>
      <VB>OrwbtC5iVeBNpJFWBVdiLK==</VB>
      <NP>
       <NP>
        <DT>BbGbgbogeaDTvevENHu97s==</DT>
……………………………………………………………
```

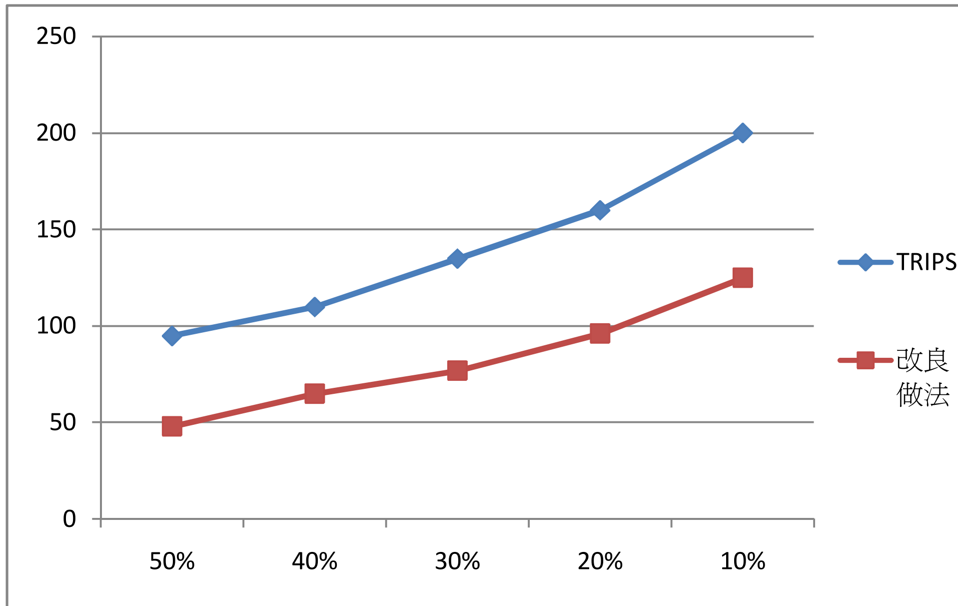


圖 5.14：對歸納子樹的樹探勘結果比較

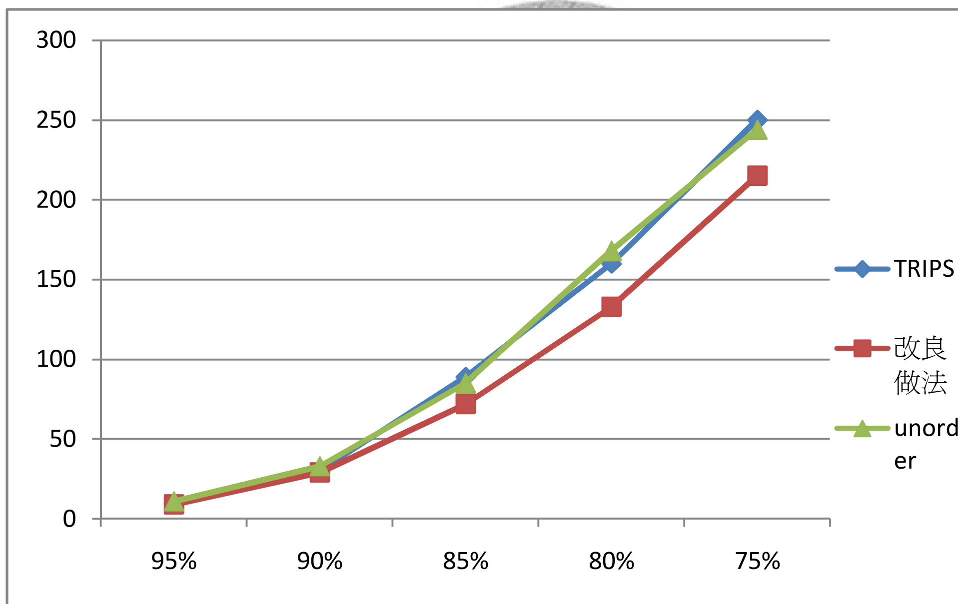


圖 5.15：對嵌入子樹的祍探勘結果比較

圖 5.14 是針對歸納、有序子樹進行的實驗結果。圖 5.15 是針對嵌入、有序與無序子樹的實驗結果。因爲 TREEBANK 的樹結構可能比較接近中文樹結構的狀況，所以實驗結果也比較相近。

第六章

結論及未來工作展望

本文改進的地方，是根據[5]中所提出的 TRIPS 演算法，改良新的資料結構，改善 TRIPS 演算法的執行時間，本論文改善的地方在針對歸納子樹和嵌入、有序子樹的探勘上，速度都有所提升。另外，本論文在針對 TRIPS 原本比較不足的嵌入、無序子樹方面，提出了補足的做法。

由實驗結果看來，本文提出的新做法確實有幫助 TRIPS 演算法提升了速度，並且也能對嵌入、無序子樹進行樹探勘的工作。

在樹探勘演算法的研究方面，未來可能會針對更加複雜或是特殊的結構，來思考不一樣的演算法，或者是針對樹結構的特性，來改進演算法。另外，像在[8]中有提出 TRIPS 的平行演算法，也可以運用本篇論文提供的改良做法進行改進，或者是新研究其他的平行演算法。

參考文獻

[1] M.J. Zaki and C.C. Aggarwal. XRules: an effective structural classifier for XML data. Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 316-325, 2003.

[2] M.J. Zaki. E_ciently mining frequent trees in a forest. Proceedings of the eighth ACM SIGKDD conference on Knowledge discovery and data mining, 2002.

[3] T. Asai, H. Arimura, T. Uno, S. Nakano (2003): Discovering Frequent Substructures in Large Unordered Trees. *Proceedings of the International Conference on Discovery Science*, Japan.

[4] S. Nijssen, JN Kok (2003): Efficient discovery of frequent unordered trees. *Proceedings of the International Workshop on Mining Graphs, Trees, and Sequences*, Croatia.

[5] S. Tatikonda, S. Parthasarathy, and T. Kurc. Trips and tides:New algorithms for tree mining. Technical Report ftp://ftp.cse.ohio-state.edu/pub/tech-report/2006/TR68.pdf, (OSU-CISRC-7/06-TR68), 2006.

[6] M.J. Zaki (2005): Efficiently Mining Frequent Embedded Unordered Trees. *Fundamenta Informaticae* vol. 65, pp. 1-20.

[7] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. Archiv für

Mathematik und Physik, 27:742-744, 1918.

[8] S. Tatikonda, S. Parthasarathy. An Efficient Parallel Tree Mining Algorithm for Emerging Commodity Processors.