國立臺灣海洋大學

資訊工程學系

碩士學位論文

指導教授：張雅惠 博士

利用 ELCA 技術對圖型資料庫搜尋 R-clique

Using the ELCA Technique to Find R-cliques in a Graph Database

研究生：張祐愷　撰

中華民國 102 年 8 月

利用 ELCA 技術對圖型資料庫搜尋 R-clique

Using the ELCA Technique to Find R-cliques in a Graph Database

研 究 生：張祐愷　　Student：Yu-Kai Chang

指導教授：張雅惠　　Advisor：Ya-Hui Chang

國 立 臺 灣 海 洋 大 學
資 訊 工 程 學 系
碩 士 論 文

A Thesis
Submitted to Department of Computer Science and Engineering
College of Electrical Engineering and Computer Scienece
National Taiwan Ocean University
In Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science and Engineering
August 2013
Keelung, Taiwan, Republic of China

中華民國 102 年 8 月

摘要

圖型資料庫(Graph database)能夠明顯表現出資料之間的相關性，而關鍵字搜尋提供使用者便利的查詢工具。但是因為圖型資料庫的結構比傳統的資料庫來的複雜許多，造成搜尋的時間複雜度遠高於傳統的資料庫搜尋。因此，如何在圖型資料中有效率的進行關鍵字搜尋便成為一個重要的議題。

[KA11]提出了 R-clique 定義，其為一群擁有所有查詢關鍵字且對應節點之間的距離小於一定值的節點叢集。[KA11]以此定義作為對圖形資料庫作關鍵字搜尋的一種規則，並提出對應的演算法以搜尋出一個圖形資料庫中所含有的所有 R-clique。本論文的目的為探討如何將複雜的圖形資料庫轉換成結構較單純的樹狀資料庫，以利用快速的樹狀搜尋方法搜尋出所要的結果。

我們提出了數種將圖型資料庫轉換成樹狀資料庫的策略，並利用 ELCA 的搜尋方式對轉換的資料進行快速搜尋。最後用實驗結果比較各種轉換方式的優劣，並和原方法比較，我們發現以廣度搜尋演算法建立並盡量產生較多分枝的樹，在效率和召回率的整體表現上最佳。

關鍵詞：圖形資料庫、關鍵字查詢

Abstract

Graph databases have been applied to represent complex data, and keyword search on graph databases is a convenient mechanism for users. However, the time complexity for keyword searching on a graph database is much higher than searching on tree databases. Therefore, it is an important issue to improve the efficiency on searching graph databases.

[KA11] proposed the idea of ``R-clique'' as the result of keyword search on graph databases, where an R-clique contains every query keyword and the distance among every keyword match is under a given threshold R. [KA11] has also proposed the Branch and Bound algorithm to quickly find all R-cliques in a graph database. In this thesis, we study how to convert a graph database to a tree database, and use tree search methods to generate the R-cliques in a graph database.

We propose several strategies to convert a graph database into a tree database. We then use the ELCA search method to get search results efficiently. Finally, we design a series of experiments to compare the different tree converting strategies and analyze the pros and cons of each method. Experimental results show that the "Most-branched" tree has best performance in terms of efficiency and recall.

Keyword: graph database, keyword search

誌謝

首先,感謝指導教授張雅惠博士,對於本論文給予相當鼎力的協助,且在論文研究期間不時地共同討論，以及悉心指正與釋疑，並彌補學生於專業學識上的不足。由於老師的耐心督導，使本論文得以順利地完成，同時也讓學生獲益匪淺，在此向敬愛的老師致上衷心的謝忱。

除此之外，感謝口試審查委員柯佳伶博士與林川傑博士，細心審稿以及論文修正，並對於本論文不吝提供寶貴的建議，使本論文更趨於嚴謹完善，在此深表謝意。

最後，感謝實驗室的的同學與學弟妹，陪伴我渡過多采多姿的實驗室生活。感謝大學部的專題生與其他實驗室的學長與同學們，在論文完成過程中給予我許多鼓勵及幫助。感謝親愛的家人，於學習成長階段給予許多支持與鼓勵。在此一併致上心中無限的感激，謝謝你們。

目錄

圖目錄

表目錄

第一章 緒論

我們在此章說明本論文的研究動機、目的和研究的方法，以及提出本論文的貢獻，並介紹相關的研究，最後說明本論文的架構以及各章節的內容。

1.1 研究動機與目的

近年來針對在圖形化資料進行關鍵字搜尋的研究越來越多，而圖形化資料不僅能以資料本身的資訊來表達意義，更能靠著資料與資料間的關係來呈現如相關度之類本身資料所沒有含有的資訊。此外，許多傳統資料庫在經過特定的規則後也能轉化成圖型資料庫，如關連式資料庫能憑藉資料的 foreign key 建立與其他表格資料之間的關係，進而轉換成圖形資料。

許多人提倡以簡易的關鍵字查詢對圖型資料進行查詢，而[KA11]提出了 R-clique 的搜尋方式對圖型資料進行關鍵字查詢。在一個 R-clique 中，針對所有關鍵字至少包含一個對應的節點，且所有對應節點之間的距離皆小於一個限制值 R。然而在圖型資料庫能夠呈現資料關係的背後，同時也必須處理因圖型資料庫的結構太過複雜而使查詢效率較慢的問題。傳統資料庫如關連式資料庫和樹狀資料庫因為結構較單純，加上近年來已經有相當多的研究提出了許多成熟的改良方法，在搜尋的效率上都已經有相當的改善。但是圖型資料庫在結構先天上處理的時間複雜度已經比傳統資料庫的結構來的複雜許多，若再考慮資料的大小較大的問題，圖型資料庫的搜尋效率將會遠低於傳統資料庫的搜尋。

[KA11]提出 Branch and Bound 演算法來搜尋一個圖所含有的全部 R-clique。首先我們說明該演算法的運作方式。一開始使用者必須給定一個限制值 R，之後 Branch and Bound 演算法會從欲搜尋的圖中取出所有符合搜尋關鍵字的節點，並依照關鍵字對每個節點作分群。接著從第一個關鍵字的節點群作為第一群，對第二個關鍵字的節點群一一檢查，若有第一群的節點和第二群的某個節點距離小於 R，則將這兩個節點結合成一個節點集合，作為後補叢集放在一個後補叢集集合(Candidate set)內，在這兩群的所有節點檢查完之後，再將候補叢集集合中所有的後補叢集與第三個關鍵字的節點群一一比較節點距離，若後補叢集所有的節點和第三群的某個節點距離皆小於 R，則將此後補叢集和第三群中被檢查的節點形成新的後補叢集並插入到新的後補叢集集合中。Branch and Bound 演算法會不斷進行這個檢查動作直到所有的關鍵字集合皆被檢查過，最後留下的後補叢集集合中便是符合查詢所要回傳的 R-clique 集合。但是 Branch and Bound 演算法的時間複雜度過高，若假設查詢句有 m 個關鍵字且 C_{MAX} 為擁有最多對應節點的集合，Branch and Bound 演算法的時間複雜度則為 $O(m^2 |C_{MAX}^{m+1}|)$，使我們想試著以較低時間複雜度的演算法來找出 R-clique。

因此，本論文希望能找出能將圖型資料有效轉換成樹狀資料的方法，並以效率較好的搜尋方式來改善圖型資料庫的搜尋。

1.2 研究方法與貢獻

在樹狀資料中，使用關鍵字搜尋時，我們可以找出各個包含所有對應節點的子樹來表示各個對應節點之間的關係，而針對關鍵字搜尋我們可以使用 LCA (Lowest Common Ancestor)搜尋來找出一組對應節點的組合所形成的子樹，該子樹的根節點便為該組關鍵字組合的 LCA 節點。但是 LCA 搜尋所找到的結果太多，需要進一步的篩選較有意義的 LCA 節點。目前經常使用的定義有 SLCA (Smallest Lowest Common Ancestor) [XP05] 與 ELCA (Exclusive Lowest Common Ancestor) [PX07] 兩種搜尋方式。SLCA 是對所有 LCA 樹中，篩選找出形成最小的樹的 LCA 節點作為 SLCA 節點的搜尋結果。而 ELCA 的定義是針對各個所有 ELCA 節點所對應的對應節點到該 ELCA 樹的根節點中的路徑中，不能包含有任何一個 LCA 節點。比較 SLCA 與 ELCA 會發現，ELCA 搜尋所找出的 ELCA 節點會包含所有的 SLCA 節點，而 SLCA 搜尋所能找到的結果總量一定不會超過 ELCA 所找到的結果數量，在一個樹中，ELCA 找到的解數量必大於或等於 SLCA 的解數量。所以我們想試著針對一個資料圖，利用特定的方法產生轉換樹之後，試著用 ELCA 搜尋快速的搜尋出在原始圖中的 R-clique 結果。

本論文提出將圖型資料轉換成樹狀資料的策略，並以[ZBWL+12]所提出的 ELCA 搜尋方式對轉換過的樹狀資料進行搜尋，最後再針對 R-clique 的規範對 ELCA 的搜尋結果進行修正。並提出多種將圖型資料轉換成樹狀資料的策略以取得較精準的搜尋結果。

針對本論文主要貢獻，總結如下所示：

1. 我們利用樹狀資料查詢計算量較圖型資料低的特性，提出五種將圖型資料轉換成樹狀資料的轉換策略，並提出 EFFR (ELCA Fast Find R-clique) 演算法使在查詢時不需要考慮複雜的圖型結構的問題處理，並能快速從轉換過的樹中找出 R-clique 的方法。

2. 我們提出 EAFR (ELCA Accurate Find R-clique) 演算法，經由修改 EFFR 演算法中檢查 R-clique 的部分來提高查詢的召回率 (Recall)。

3. 我們以針對圖型搜尋的 Branch and Bound 演算法，以及我們所提出的五種產生轉換樹的策略，分別對 EFFR 演算法以及 EAFR 演算法進行查詢召回率和效率的比較。發現在效率上對轉換樹查詢比直接對圖型資料庫作查詢來的較佳。查詢召回率方面在 R 的大小不超過一定程度的狀況，針對轉換樹的查詢結果能夠保持相當的召回率。

1.3 相關研究

首先我們討論針對資料圖進行關鍵字查詢的相關研究。[DLQW+07]提出了對關連式資料庫的資料轉換成資料圖的方法,並從其中找出 Top-k 的 Steiner tree。研究[LOF+08]提出稱作 EASE 的方法，以便對 Unstructured、Semi-structured 和 Structured Data，皆可做關鍵字的查詢。EASE 的作法也是將 data 表示成 graph，然後利用 adjacency matrix，解決 r-radius Steiner graph 的問題。研究[TJM+08]提出讓使用者在不需要知道複雜的 schema 以及 query language 的情況下，能建構出查詢句的 template。該論文首先將 keyword match 到 schema graph 上，並附上每個資料來源的權重值，然後在 schema graph 上找到 top k 的 Steiner tree，以產生 top k 的 query。該系統會進一步針對所產生的查詢句提供答案以及對應的資料來源，以便使用者提供 feedback 給系統，讓系統可以學習並改變權重值。研究[YCHH12]則探討針對結構化資料關鍵字查詢改寫的問題，這些改寫後的查詢提供了代替原輸入的描述，進而可以更好的捕捉使用者的資訊需求。改寫過程分為在 off-line 階段產生 Term Augmented Tuple Graph，並加入一個新的 preference vector 參數以抓取具有相關性的 term，到 on-line 階段時使用一有效的機率性產生模組，來產生改寫的查詢句。

和本論文所探討的問題最為近似的是研究[QYCT09]和[KA11]。[QYCT09]提出了對關連式資料庫搜尋 community 的作法，所謂的 community 有一個中心點，然後每一個對應節點到中心點的距離皆小於一個給定的值。為了改善 Graph database search 搜尋結果的召回率以及資料的可讀性,[KA11]進一步提出 R-clique 的定義作為查詢的結果，並針對 R-clique 的搜尋提出兩種演算法。第一種是利用 Branch and Bound 演算法一一檢視可能形成答案的節點組合以取得所有 R-clique，另一種是利用分割搜尋空間來尋找近似 R-clique 的搜尋結果,同時也可以利用各個子搜尋空間的排序來支援 top-k 的搜尋。

至於特別針對 XML 文件進行關鍵字搜尋的部分,[XP05]提出 SLCA 的定義，受到廣泛的重視與引用。[PX07]則修正其定義而後提出 ELCA 的定義，以獲得一些 SLCA 無法取得的資訊。[ZBWL+12]針對關鍵字查詢的 SLCA 以及 ELCA 提出了比以往更快速的搜尋方法,其方式是分別對關鍵字產生相對應直接或間接擁有對應節點 ID 的 inverted list，基於 intersection operation 提出了全新完整有效的演算法，使得問題即是在 inverted lists 中尋找符合一定條件的節點，實驗結果顯示其方法優於其他研究的做法一至二個 order。論文[TPSS11]則提出對 XML 文件指定一節點，找出含有查詢關鍵字且最接近該節點的節點搜尋方式，更將此搜尋方式延伸到 XPath 上來改善 XPath query 的效率，同時提出能有效配合上述搜尋方式的索引格式。

最後，我們討論將圖轉換成樹在演算法領域的相關研究。研究[LR98]提出對

一張資料圖找出一個近似最多葉節點的轉換樹(spanning tree)的策略，而[CW04]的專書則提出了數種不同的轉換樹轉換方法，並分析各種轉換樹的特徵以及最佳化。除了將圖轉成轉換樹，研究[CCW13]也提出將資料圖轉換成 BFS+ tree，以保留所有的邊來解決 “edge partition”的問題。該作法會先對原始圖產生 BFS tree，再將尚未加入到樹中的邊以新增小孩節點的方式通通加入到樹中。

1.4 論文架構

本論文其餘各章節的架構如下：第二章介紹本論文所會使用到的相關定義，包括圖型資料的表示法、圖型資料所建成的索引格式、R-clique 的定義及輸出結果，藉以對本論文所期望解決的問題做詳細的定義，並介紹本論文所欲改善的對象，也就是研究[KA11]所提出的 Branch and Bound 演算法。在第三章說明我們轉換圖型資料至樹狀資料的方法、以及說明如何結合研究[ZBWL+12]的搜尋方式來達成回傳 R-clique 的方法。在第四章會說明提升搜尋品質的改良方式。在第五章我們會以實驗來比較每一種轉換策略對搜尋的影響，並比較搜尋效率的快慢及品質的優劣。最後在第六章提出結論以及本論文未來的研究方向。

第二章 相關定義

我們在此章說明相關的定義，包括圖型資料的表示法、圖型資料所建成的索引格式、R-clique 的定義及輸出。接著，我們定義了本論文所針對的問題，並介紹其他研究針對相同問題的演算法 Branch and Bound。

2.1 圖型資料 (data graph) 表示法

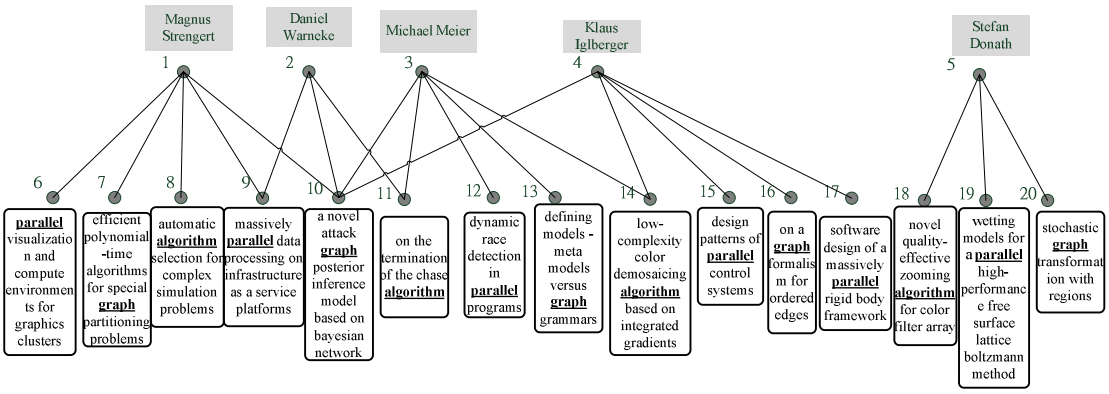

圖 2.1 範例搜尋圖

首先，我們介紹本論文所使用的圖型資料的呈現方式。在一個資料圖中含有多個節點，每個節點包含著各自擁有的關鍵字資訊，且每個節點至少含有一條與圖中的其他節點連接的邊，每條邊上記錄節點間的距離，距離值可依照既有的 scoring function 去計算，本論文中是依照研究[KA11]的定義，個別計算出每個邊的距離。

為了可以方便辨認圖上特定的節點，我們會對圖中的節點作編號，各個節點依照被程式讀入的順序從 1 開始一一作編號。如圖 2.1 中的節點，可分為作者和論文標題兩大類，而每條邊連結一篇論文到對應的作者。圖中每個節點旁的編號便為該點的編號，而圖中編號為 13 的節點擁有關鍵字”graph”，利用節點編號來指定節點便不會遭遇到關鍵字相同導致無法確定是那個節點而產生混淆的情形。

由於一個資料圖可能不是一個有完整連接的圖，所以我們會對資料圖 先作處理讓所有的點都可藉由經過其他節點作連接。我們的作法是新增一個編號為 0 的點作為 pseudo root (PR)，並連接該點至編號為 1 的點，然後開始遍歷(traverse)搜尋可到達的節點，如果發現有新的節點無法被連接，則新增由此點到 PR 的連線。我們以範例圖 2.1 為例子，可看到 PR 連線到節點 1 後可以再走到節點 6, 7, 8,

9, 10，而由節點 9, 10 又可以走到節點 2, 3.....，之後會發現節點 5 沒有辦法連接，此時我們將節點 5 新增一個連線到 PR，之後再從節點 5 開始遍歷剩下的節點，完成後圖會變成如圖 2.2。我們以這樣處理過的圖作為我們的搜尋圖。同時為了簡化起見，之後的範例圖中對應每篇論文的節點我們只列出其中具代表性的關鍵字，而作者節點則省略其關鍵字。

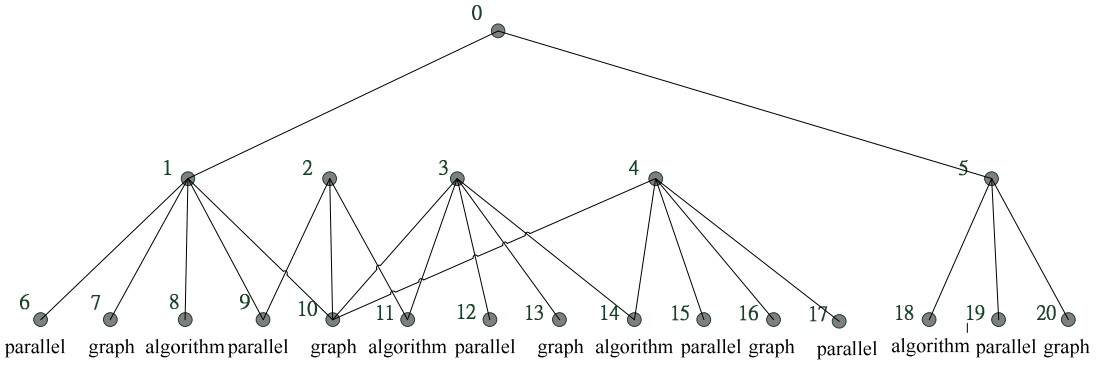

圖 2.2 處理過有完整連接的搜尋圖

對照之前計算距離的公式，為了後續的解釋方便，我們在本論文的範例中先將距離計算公式簡化成 $v_{deg} + u_{deg}$，以避免數值太小。同時為了讓如圖 2.2 中新增加的連結邊與原圖的邊有明顯區隔，不要影響到 R-clique 的搜尋結果，我們會將其距離設為一個極大的值，讓演算法在計算 R-clique 時會直接剔除這些邊。對應的距離值如圖 2.3 所示。

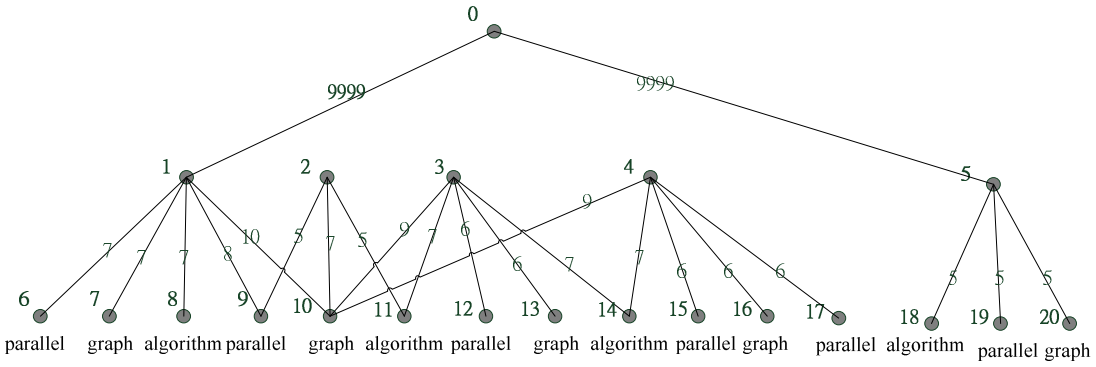

圖 2.3 有完整連接且計算出距離的搜尋圖

2.2 關鍵字查詢之輸出結果

在此節中，我們依照研究[KA11]的定義，將本論文會使用到的定義說明如下：

[定義 2.1]： 對應節點(match) ： 給定一個資料圖 G 或樹 T 以及查詢句 Q，在圖 G 或樹 T 中的節點 n 若含有能對應查詢句 Q 的關鍵字，我們稱 n 為 Q 對圖 G 或樹 T 中的對應節點(match)。

[定義 2.2]： R-clique ：給定一個資料圖 G 及限定距離大小值 threshold R，查詢句 $Q = \{k_1, k_2, \ldots, k_l\}$，一個 R-clique 代表在 G 中的一組節點集合，針對查詢句 Q 中的所有關鍵字皆有對應節點，且每一節點至 R-clique 中另一個節點的距離皆小於 R。在 R-clique 中的節點距離為，在圖 G 中兩個節點的最短距離。

[範例 2.1] ：考慮圖 2.3，我們希望找出與 parallel、graph、algorithm 這三個關鍵字有一定相關程度的論文，對查詢句關鍵字下"parallel", "graph", "algorithm"，R 值定為 14 ，七組 R-clique 分別為節點{6, 7, 8}，{9, 10, 11} ，{11, 12, 13}，{12, 13, 14}，{14, 15, 16}，{14, 16, 17}，與節點{18, 19, 20}，如圖 2.4，在該圖中兩個節點的距離皆為圖 2.3 中兩節點所求得的最短路徑距離。注意到很多組合因為節點間的距離超過 14 而不能回傳，如{10, 14, 15}。

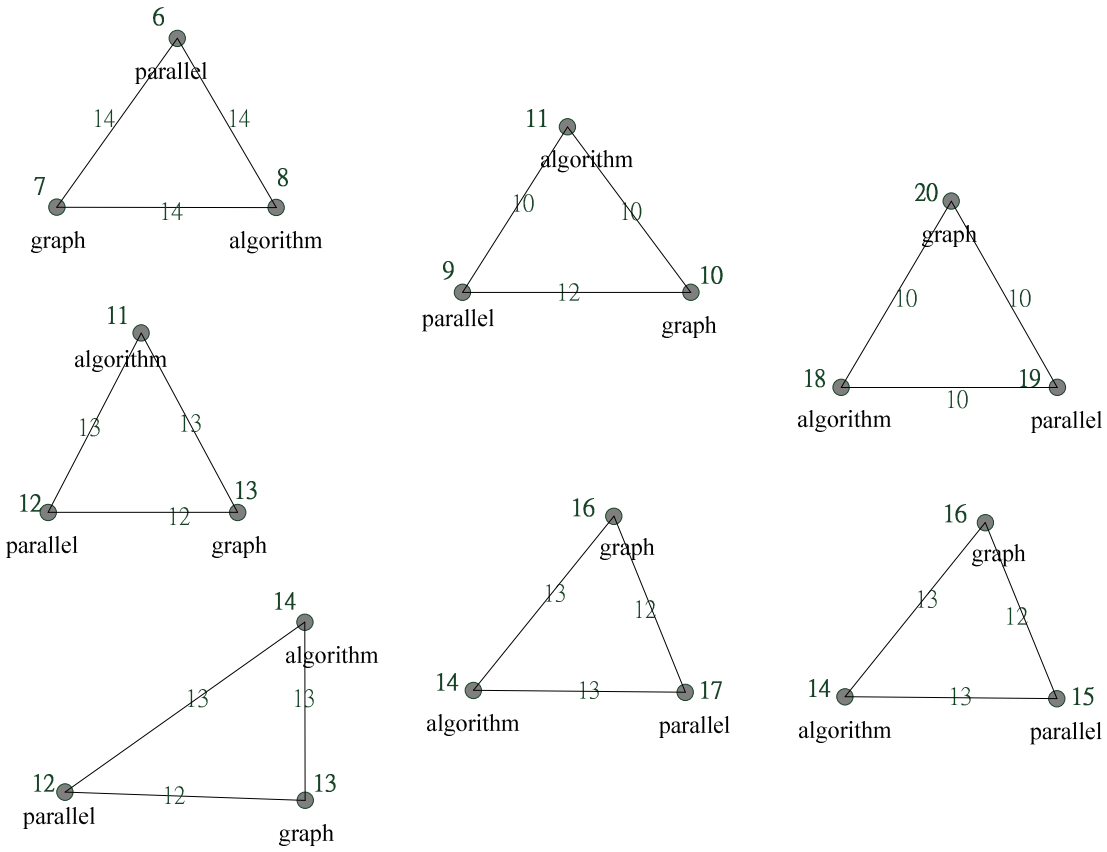

圖 2.4 範例 R-clique

[定義 2.3]：Minimal Steiner tree (MST) ：給定一個資料庫 G 及一個 R-clique C，在圖 G 中一個包含所有 C 的節點的 sub-tree 稱作 Steiner tree，而其中距離總和最小的稱為 Minimal Steiner tree，簡稱 MST。

[範例 2.2]：考慮圖 2.3、圖 2.4 及圖 2.5，針對一個 R-clique，在原圖中可找到一個邊距離總和最小且包含該 R-clique 的所有關鍵字節點的 MST，例如 R-clique{14, 15, 16}對應回圖 2.3 的原圖可得到圖 2.5 中的 MST {4, 14, 15, 16}，其距離總和為 7+6+6=19。

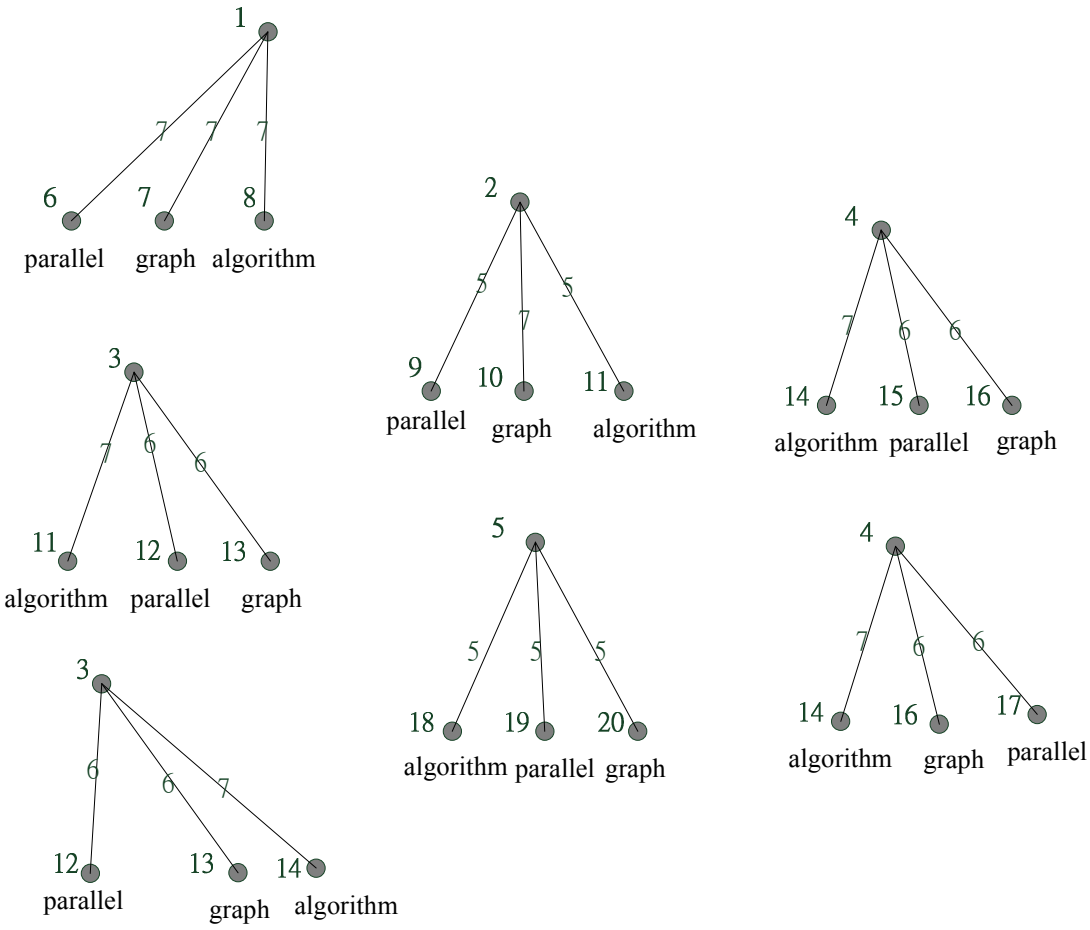

圖 2.5 範例 Steiner tree

2.3 Branch and Bound 演算法介紹

[KA11]針對其所定義的輸出結果，提出了 Branch and Bound 演算法來找出圖中所有符合的 R-clique，並輸出每個 R-clique 對應的 Steiner tree 作為結果。在 Branch and Bound 演算法中，在圖中所有符合關鍵字的節點會依照關鍵字分群，之後再一一檢查距離是否小於 R 能否形成 R-clique，在找出所有的 R-clique 之後再針對各個 R-clique 呼叫 DrawSteinerTree 演算法將每個 R-clique 對應的 Steiner tree 輸出。

演算法名稱：Branch and Bound Main

輸入：資料圖 G, 查詢句{k_1 , k_2 , ... , k_m}共 m 個關鍵字, 距離限制 R 輸出：所有符合搜尋的 R-clique 的 Steiner tree
L01: rList ← Branch and Bound(G, {k_1 , k_2 , ... , k_m}, R) L02: foreach RC in rList do L03: SteinerTreeSet.add(DrawSteinerTree(RC, G)) L04: return SteinerTreeSet

圖 2.6 Branch and Bound Main 演算法[KA11]

在[KA11]的 Branch and Bound 方法中，全部的流程可整理成圖 2.6 的 Branch and Bound Main 演算法，而主要步驟分成兩個階段，第一個階段是在 L01 呼叫 Branch and Bound 演算法得到所有的 R-clique 並存入到 rList 中，第二個階段是在 L02~L04 將 rList 中所有的 R-clique 依照原始資料圖 G 畫成 Steiner tree，最後再回傳所有的 Steiner tree。

演算法名稱：Branch and Bound 輸入：資料圖 G, 查詢句{k_1 , k_2 , ... , k_m}共 m 個關鍵字, 距離限制 R 輸出：所有符合搜尋的 R-clique
L01: for i ← 1 to m do L02: C_i ← the set of nodes in G containing k_i L03: rList ← empty L04: for i ← 1 to size(C_1) do L05: rList.add(C_i^1) L06: for i ← 2 to m do L07: newRList ← empty L08: for j ← 1 to size(C_i) do

```
L09:            for k ← 1 to size(rList) do
L10:                if ∀ node ∈ rList_k dist(node, C_i^j) ≤ r (where rList_k is the kth
                                                              element of rList) then
L11:                    newCandidate ← C_i^j .concatenate(node)
L12:                    newRList_k .add(newCandidate)
L13:        rList ← newRList
L14:    return rList
```

圖 2.7 Branch and Bound 演算法

演算法名稱：DrawSteinerTree

輸入：圖型資料庫 G, 一個 R-clique RC

輸出：R-clique RC 的 steiner tree

L01: Let G_1 be r-clique RC.

L02: Find the minimal spanning tree T_1 of G_1

L03: Create graph G_2 by replacing each edge in T_1 by its corresponding shortest path in G.

L04: Find the minimal spanning tree T_2 of G_2.

L05: Create a Steiner tree from T_2 by removing the leaves (and the associated edges) that are not in the r-clique.

L06: return T_2

圖 2.8 DrawSteinerTree 演算法

Branch and Bound 主要步驟如下，如圖 27 所示。首先，在 L01 和 L02 將每個節點依照 m 個關鍵字分成 m 群，在 L03 建立 rList 作為存放後補叢集的集合，接著在 L04 和 L05 將第一個關鍵字的節點群存入 rList 後，在 L06 開始進入迴圈，在 L07 建立空的 newRList 用來存放接下來運算結束所得到的新的後補叢集的集合。第一次進入迴圈後會將 rList 中的所有節點和第二個關鍵字的所有節點比較

距離，一但距離小於 R，便將檢查到的 rList 中所有的節點和第二群中檢查到距離小於 R 的節點組合成新的後補叢集，再將這個後補叢集放入 newRList 中，再將第二群的所有節點檢查完之後，把 rList 的內容取代成 newRList 所存放的後補叢集集合。再進行下一次的迴圈，在迴圈中檢查完所有的關鍵字集合後得將所求得的 R-clique 存入 rList，在迴圈結束後從最後回傳 rList 取得所有 R-clique。

為了顯示每個 R-clique 中的點在原圖中的關係，[KA11]針對各個 R-clique 對應圖 G 呼叫 DrawSteinerTree 演算法畫出對應的 Steiner tree，最後回傳所有 Steiner tree 的集合 SteinerTreeSet 為搜尋結果。找出 Minimal Steiner Tree 是個 NP 的問題，[KA11]提出一個找近似解的方法，如圖 2.8 所示。其步驟主要分成下列的幾個部分，首先必須先對輸入的 R-clique RC 畫出 Minimal spanning tree T_1，之後將 T_1 的所有邊替換成邊上的兩個點在圖 G 上的最短路徑，建構出圖 G_2，再對圖 G_2 畫出 G_2 的 Minimal spanning tree T_2，最後再針對 T_2 將所有葉節點不屬於 RC 的點從 T2 中剔除，直到所有葉節點皆為 RC 的點為止最後回傳 T_2 即為 RC 的 Steiner tree。

在時間複雜度上，假設所有節點的距離皆小於 R 且 C_{MAX} 為擁有最多對應節點的集合，由於在圖 2.5 中 L06 需要作 m-1 次，L08 在最差的狀況下必須作$|C_{MAX}|$次，而 L09 到 L12 部分 rList 內的數量會不斷增加，最後會變成有|C1|*|C2|*…*|Cm|個，在最差的狀況會達到$|C_{MAX}|^m$個，另外在 L10 檢查 dist 的部分由於有 m 個關鍵字故最多必須檢查 m 次，故 Branch and Bound 演算法的時間複雜度為 $O(m^2 |C_{MAX}{}^{m+1}|)$。DrawSteinerTree 演算法的部分我們假設 RC 的點為 S，圖 G_1 的點為 V，則可以在 $O(|S||V|^2)$的時間建出 G_1，而畫出 Minimal spanning tree T_1 可以在 $O(|S|^2)$完成，而建構 G_2 可以在 O(|V|)的時間完成，畫出 Minimal spanning tree T_2 只需要 $O(|V|^2)$，最後剔除非 R-clique 節點的步驟可以在 O(|V|) 完成，故 DrawSteinerTree 演算法的時間複雜度為 $O(|S||V|^2)$。

Branch and Bound 演算法需要兩個索引，一個是針對節點關鍵字和節點編號建成的反轉索引(inverted list)，反轉索引的一筆資料為一個關鍵字與含有該關鍵字的節點編號，利用反轉索引可以快速的找出哪些節點有查詢關鍵字並快速的分群。另一個是記錄所有節點的最短距離的鄰居索引，該索引記錄所有節點到其他節點的最短距離與最短路徑。

2.4 ELCA 相關定義

[定義 2.3]： LCA 節點：對樹 T 進行關鍵字查詢 Q = {k_1, ..., k_m}，子孫中包含一組 k_1 到 k_m 對應節點的層數最低的節點，稱為 Q 對 T 的 LCA 節點。

[定義 2.4]： SLCA 節點：對樹 T 進行關鍵字查詢 Q = {k_1, ..., k_m}，在 Q 對 T 的 LCA 節點中若其子孫不包含其他 LCA 節點，則稱為 Q 對 T 的 SLCA 節點。

[定義 2.5]： ELCA 節點：對樹 T 進行關鍵字查詢 Q = {k_1, ..., k_m}，假設某個 LCA 節點 e，在以 e 為 root 的子樹中排除其他 SLCA 的子樹後，對{k_1, ..., k_m}仍各自包含對應節點，則稱此 LCA 節點 e 為 Q 對 T 的 ELCA 節點。

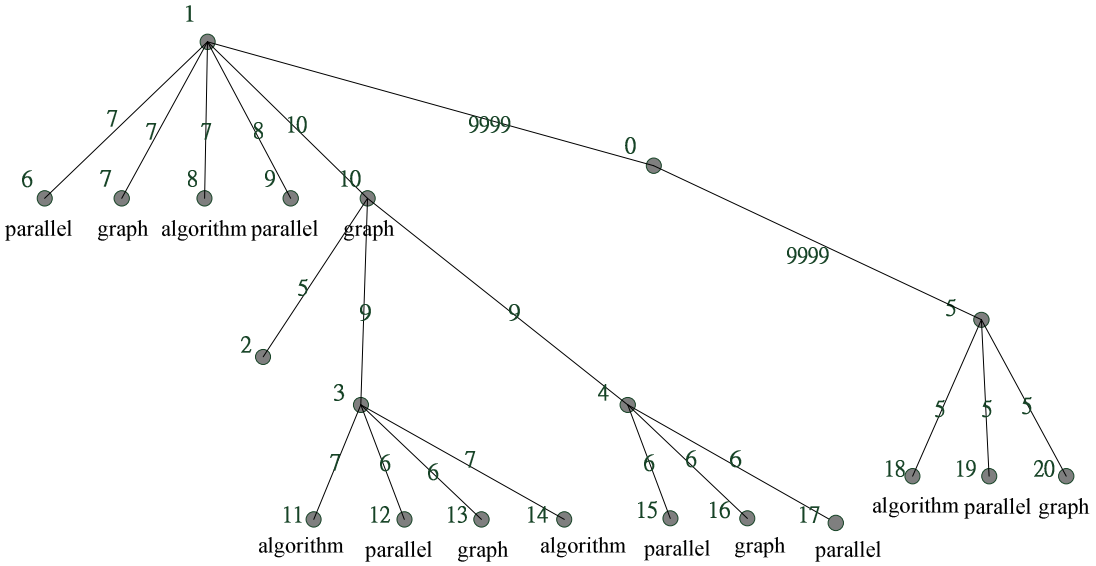

圖 2.9 範例查詢樹

[範例 2.2]： 以圖 2.9 的樹作查詢，查詢關鍵字為"parallel", "graph", "algorithm"，則可以得到 LCA 節點為 1,3,5，進一步依照 ELCA 節點定義可發現節點 1 排除節點 3 之下的對應節點，仍有節點 6,7,8,9,10,15,16 等七個節點滿足查詢，而節點 5 有節點 17,18,19 三個節點滿足查詢。最後可以得到節點 1, 3, 5 此三個節點為 ELCA 節點。

[定義 2.6]： ELCA 樹: 對一個樹進行關鍵字查詢 Q，取得 ELCA 節點後將該 ELCA 節點所含有所有的對應節點到該 ELCA 節點中間的路徑取出合成一個新的樹，我們稱此樹為一個 ELCA 樹。

[範例 2.3]： 以圖 2.9 的樹作查詢，查詢關鍵字為"parallel", "graph", "algorithm"，

依照範例 2.2 我們可得知 3 個 ELCA 樹分別為{1, 6, 7, 8 , 9 , 10, 15, 16, 17}, {3, 11, 12, 13, 14}, {5, 18, 19, 20}，如圖 2.7 所示。

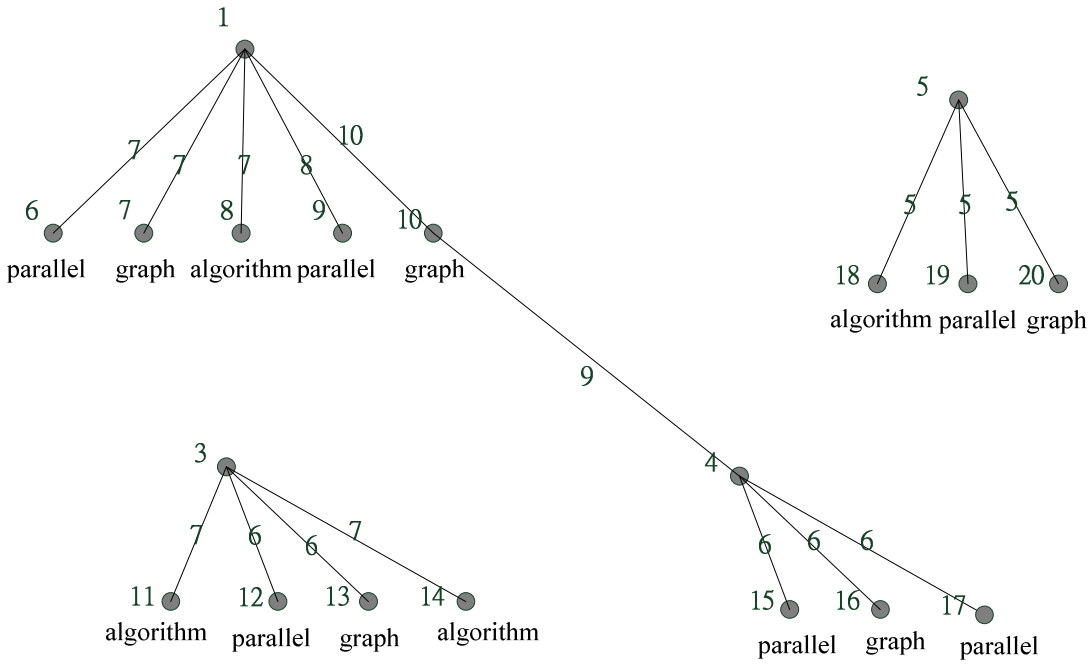

圖 2.10 範例 ELCA 樹

[定義 2.7]： ER tree: 對一個樹進行關鍵字查詢 Q，取得 ELCA 樹後，若該樹中一個以 ELCA root 為根節點的 subtree，包含每個查詢關鍵字正好一次，且節點間的實際最短距離皆小於 threshold R，也就是該組對應節點形成一個 R-clique，則我們稱此樹為一個 ER tree。

[範例 2.3]：參考圖 2.10 及圖 2.11，查詢關鍵字為"parallel", "graph", "algorithm"，且 R 值為 14，由節點 1 為根節點的 ELCA 樹我們可以發現節點{6, 7, 8}對應查詢句能夠組合成 ER tree，而節點{8, 9, 10}也滿足查詢句能組合成另一個 ER tree，但是節點{8, 9, 10}之間的距離大於我們限制的 R，所以在這個範例中不應該被回傳。在圖 2.8 我們列出由圖 2.10 所能產生的 ER tree。

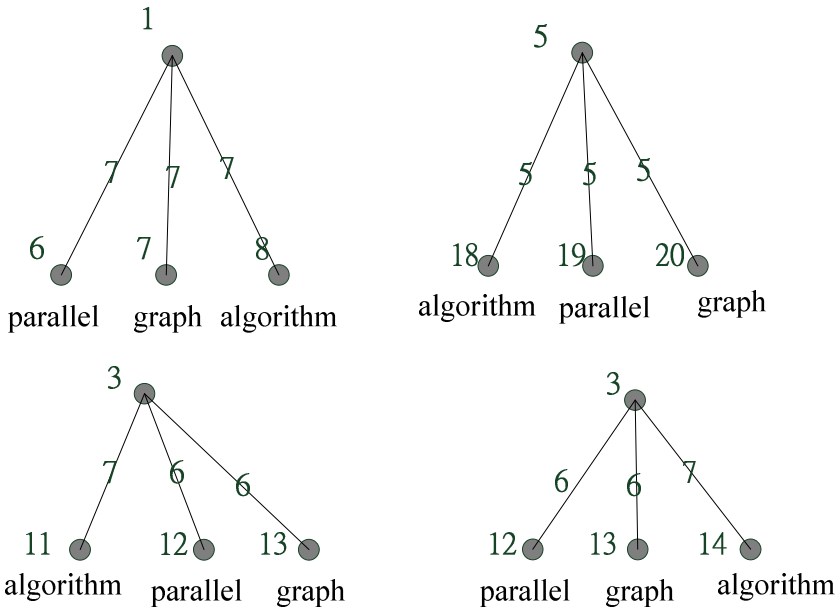

圖 2.11 範例 ER tree

本論文的作法，基本上是將原始資料圖轉換成資料樹後，針對查詢句中的關鍵字找出 ELCA 樹，再檢查樹中對應節點間的距離是否小於 R，然後以 ER tree 的形式回傳，我們在之後的章節會有詳細的說明。相較於[KA11]的做法，由於我們檢查的範圍現在在每個 ELCA tree 下，所以效率會提升，但可能無法全數找到所有的 R-clique，另外，兩種方法回傳的樹也不盡相同，我們也會在實驗章節進行比較。

第三章 演算法說明

在本章中，我們說明轉換資料圖的理由以及介紹如何將資料圖轉換成資料樹並用[BCGL+12]提出的 ELCA 搜尋方式來搜尋結果。

由於對資料圖的搜尋效率比起資料樹的搜尋方式的效率較差，加上針對資料樹的搜尋研究已經相當成熟，所以我們採用將本來的資料圖建出樹後，再利用資料樹的搜尋方法快速找出搜尋結果。在資料樹的研究上許多論文是以 SLCA 搜尋方式作為搜尋結果，但是 SLCA 搜尋方式會過濾掉許多其它有可能也需要被回傳的資訊，並不是相當適合我們的查詢方式。參照圖 2.6，由於節點 1 底下有節點 3 為 SLCA 節點，故節點 1 無法以 SLCA 的搜尋方式被找到。除了 SLCA 搜尋方式之外，論文[XP07]提出了 ELCA 搜尋方式，比起 SLCA 搜尋方式 ELCA 搜尋方式能夠保留部分被 SLCA 搜尋方式過濾掉的需要資訊，故我們選擇 ELCA 搜尋方式作為我們將資料圖轉換成資料樹之後的搜尋方式，並且選擇目前所知是最快的方法[BCGL+12]作為本論文使用的搜尋方法。

3.1 轉換資料圖成資料樹

如同 Branch and Bound 演算法，我們會根據原始資料圖(如圖 2.1)建立所有節點的鄰居索引，以記錄節點間的最短距離。目前我們是利用 Dijkstra 演算法求出指定節點到圖中其他節點的最短距離與最短路徑，而且距離太遠的點不予記錄以節省空間。

接下來，我們將原始圖轉換成樹，以進一步利用新建出的轉換樹（transformed tree）作 ELCA 查詢來篩選初步的結果。以下分節討論我們所使用的轉換策略。

3.1.1 Highest Degree Tree (HD tree)

這個策略優先選擇 Degree 較高的點來建樹來建出較平坦的樹，希望能以較平坦的樹產生層數較低且總數量較少的 ELCA 樹，盡量減少原本屬於同一個 R-clique 的對應節點因分散到不同 ELCA 樹下而無法被回傳。由於在建樹過程中需要知道每個節點的 Degree，故在建樹之前我們會先對資料圖中所有節點先計算 Degree 的資料，並用一個陣列記錄這些節點資料，同時在陣列中記錄各個節點是否已經被加入到樹中。建樹的步驟基本為：以最高 Degree 的節點為根節點，記錄根節點所連接到的所有節點作為根節點的小孩節點，再以剛剛所記錄的節點中擁有最高 Degree 節點的點繼續搜尋可連結到的節點，直到圖中所有節點皆被加入轉換樹中。以下我們列出建立 HD tree 的演算法，注意到我們是以如同圖 2.2 的連接圖（connected graph）作為輸入。

演算法名稱：Highest degree Tree(HD tree) :

輸入： Connected graph CG = { V , E }

輸出：spanning tree T

區域變數：陣列 L：記錄節點編號及其 Degree ，以及是否被加入到樹中

L_{count}：記錄未加入樹中的節點數量

L01: L ← ∅

L02: foreach vertex V in CG

L03: save V and V's degree into L and L_{count}++

L04: Root ←the vertex which has the highest degree in L

L05: set R as T's root node; mark Root as "added into T" and L_{count}--

L06: Q ← an empty priority queue sorted by degree (high to low)

L07: foreach edge E adjacent to Root

L08: if E's destination node V_d is not in T

L09: add E and V_d to T

L10: mark V_d as "added into T" and L_{count}--

L11: push V_d into Q

L12: end if

L13: end foreach

L14: V_T ← Q.removeTop()

L15: while(L_{count} != 0)

L16: foreach edge E adjacent to V_T

L17: if E's destination node V_d is not in T

```
L18:            add E and V_d to T
L19:            mark V_d as "added into T" and L_count--
L20:            push V_d into Q
L21:          end if
L22:       end foreach
L23:       V_T ← Q.removeTop()
L24:    End while
L25:    return T
```

圖 3.1 HD tree 演算法

在 L01~L03 我們建立陣列 L 將圖 G 中所有節點 V 全部儲存到 L 中，且記錄各個節點的 Degree，並用變數 L_{count} 記錄尚有幾個節點未被加入到轉換樹中，接著在 L04 從 L 中選出有最高 Degree 的點 R，在 L05 以 R 作為轉換樹 T 的根節點並將 R 從 L 中取出。L06 我們建立一個 priority queue Q，在後面 Q 會將存入的節點以 Degree 由高到低的順序排序，確保 Q 的頂端回傳的點是 Q 中擁有最高 Degree 的節點。從 L07 到 L13 我們將對 R 有連結的節點加到轉換樹 T 中，並將這些節點從 L 中取出並存到 Q 中。在 L14 從 Q 取出有最高 Degree 的節點，並在 L15~L24 中重複檢查和該節點有連接的其他節點是否要放入轉換樹 T。在 L16~L22 檢查有連接的節點是否已經在轉換樹中，而在 L17 的檢查可以藉由陣列 L 的記錄資料知道該節點是否已經被加入到樹中，若尚未被加入到轉換樹中則將該節點加入轉換樹中並放入 Q 排序。在 L23 會取出 Q 中 Degree 最高的點，並回到 L16 繼續建構轉換樹，直到 L_{count} 為零代表 L 中所有節點皆被加入到轉換樹中便完成轉換樹 T。

ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Degree	2	6	3	5	5	3	1	1	1	2	4	2	1	1	2	1	1	1

ID	18	19	20
Degree	1	1	1

表 3.1 各節點的 Degree

ID	1	3	4	10	2	5	0	9	11	14	6	7	8	12	13	15	16	17
Degree	6	5	5	4	3	3	2	2	2	2	1	1	1	1	1	1	1	1

ID	18	19	20
Degree	1	1	1

表 3.2 各節點的 Degree (依 Degree 高到低排序)

接著我們用圖 2.3 作為範例來解釋建構 HD tree，表 3.1 及表 3.2 列出了各個節點的 Degree。首先會從圖中選出 Degree 最高的點作為 HD tree 的根節點，在圖中節點 1 為最高的點，將節點 1 作為 HD tree 的根節點並加入和節點 1 有連結的點{0, 6, 7, 8, 9, 10}作為子節點，並將這些節點依照 Degree 的高低放入一個佇列中排序，接著從佇列中取出 Degree 最高的點作為下一個要加入轉換樹的點，在此處取出的點為節點 10，並將和節點 10 有連接且尚未被加入到轉換樹的點加到轉換樹中並放入佇列排序，之後重複此動作直到圖 2.3 的所有節點皆被加入到轉換樹中，最後會得到圖 3.2 的轉換樹。

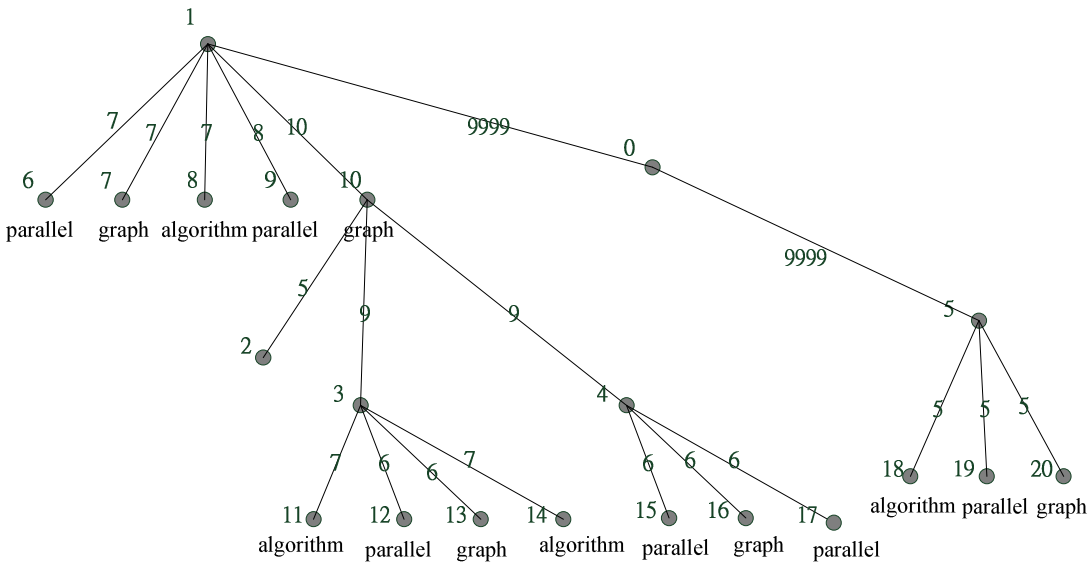

圖 3.2 範例查詢圖所生成的 HD tree

3.1.2 Lowest Degree Tree (LD tree) ：

由於原始圖中邊的距離是由點的 Degree 來決定，我們以選擇低 Degree 節點為優先建樹的方法，期望可以畫出整體距離較低的樹。建樹的步驟基本上是：以最低 Degree 的節點為根節點，記錄根節點所連接到的所有節點作為根節點的小孩節點，再以剛剛所記錄的節點中擁有最低 Degree 節點的點繼續搜尋可連結到的節點，直到圖中所有節點皆被加入轉換樹中。

LD tree 的演算法類似 HD tree 的演算法，參考圖 3.1，主要是在 L04 改為選擇 Degree 最低的點，而 L06 的 Priority queue 的排序改成由 Degree 低到高排序，剩下的步驟則和 HD tree 演算法一樣。

接著我們用範例圖來解釋建構 LD tree，參考圖 2.3，首先從圖中選出 Degree 最低的點作為 LD tree 的根節點，在圖中有數個節點的 Degree 皆為最低，我們以節點編號較小的點優先選擇，所以選擇節點 6。將節點 6 作為 LD tree 的根節點並加入和節點 6 有連結的點節點 1 作為子節點，之後將和節點 1 連接的節點放入佇列中依照 Degree 低到高排序，最後不斷重複此動作可以得到圖 3.4 的轉換樹即為圖 2.3 的 LD tree。比較圖 3.2 與圖 3.3，圖 3.2 樹高 4 層，較為扁平，圖 3.3 樹高 7 層，較為瘦長。

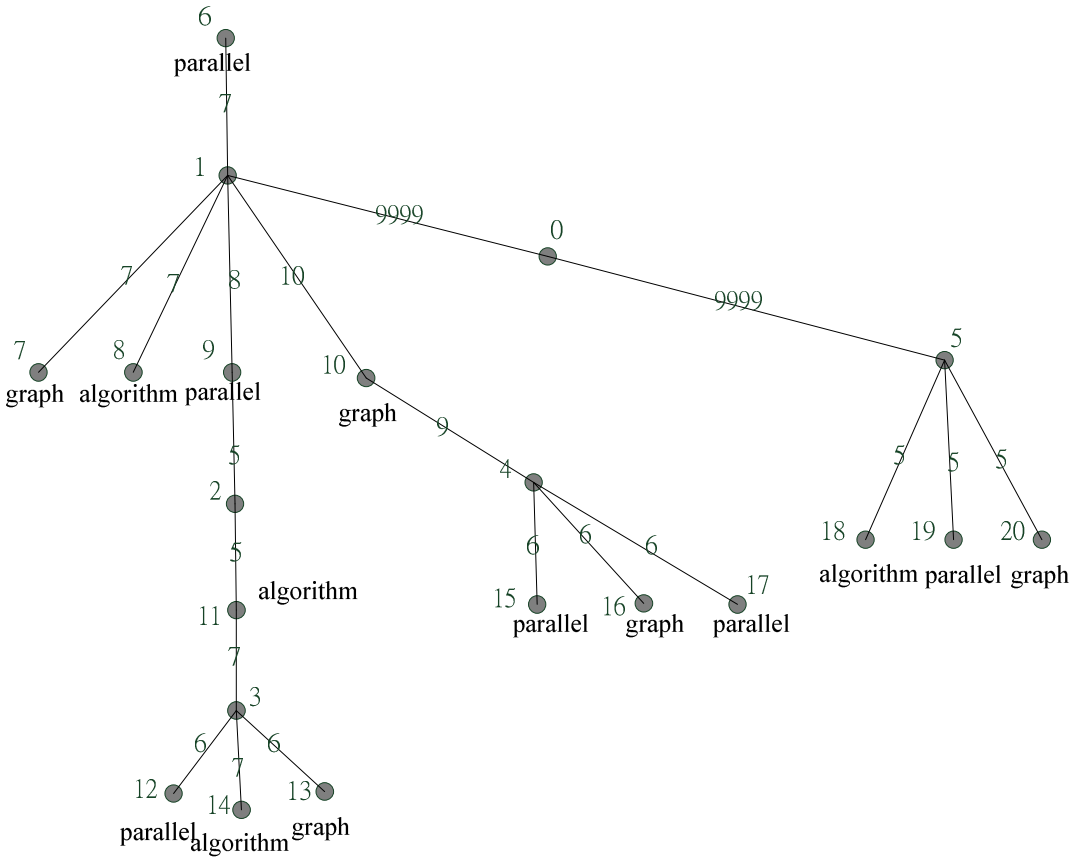

圖 3.3 範例查詢圖所生成的 LD tree

3.1.3 Highest Low Degree Tree (HLD tree) ：

如同前述邊的距離是由節點的 Degree 決定，由於 HD tree 會優先選擇 Degree 較高的節點，導致建構出來的樹中邊的距離也會較高，但是 LD tree 優先選擇 Degree 低的點，又會讓一個點下面的子節點偏少，故我們結合前兩種方法如下：

以最高 Degree 的節點為根節點，記錄根節點所連接到的所有節點作為根節點的小孩節點，再以剛剛所記錄的節點中擁有最低 Degree 節點的點繼續搜尋可連結到的節點，再選擇 Degree 最高的節點繼續搜尋可連結到的節點，之後持續交錯著選擇 Degree 高與 Degree 低的節點，直到圖中所有節點皆被加入轉換樹中。

演算法的部分和 HD tree 大部分相同，不同的是在 L06 的 Priority queue 的部分由於必須交錯選擇 Degree 高的節點或 Degree 低的節點，故在選擇節點時第一次會從頭選擇 Degree 最高的節點，第二次選擇 Degree 最低的節點，第三次再選擇 Degree 高的節點，之後就以此種規則交錯選取 Priority queue 的頭或尾的節點，剩下的步驟則和 HD tree 演算法一樣。

我們再以圖 2.3 作為範例說明建立 HLD tree，一開始同 HD tree 我們先

選出 Degree 最高的點也就是節點 1，並將有連接的點，如節點 0,6,7,8,9,10 加入到轉換樹中並將這些點放入佇列中排序，但是在這邊排序的條件和 HD tree 不同，是以 Degree 低到高來對節點排序，由於 Degree 最低的點也就是 Degree 為 1 的點（如點 6, 7, 8）無其他相鄰的點，所以考慮 Degree 次低的點，也就是節點 0 與節點 9，由於兩個節點的 Degree 相同，我們優先選擇編號較低的點，在此選擇節點 0 加入到轉換樹中並將和節點 0 有連接且尚未被加入到轉換樹中的點 5 放入佇列中排序，接下來再從佇列中選擇 Degree 最高的點，之後重複一樣的動作最後就會得到圖 3.4 的 HLD tree。

比較圖 3.4 的 HLD tree 與圖 3.2 的 HD tree，我們發現由於演算法篩選條件的不同，讓節點 10 底下的子節點被移到節點 9 下，改變了一個點底下所含的子節點數量，後面我們會在實驗比較兩種樹的差異對查詢的召回率造成的影響。

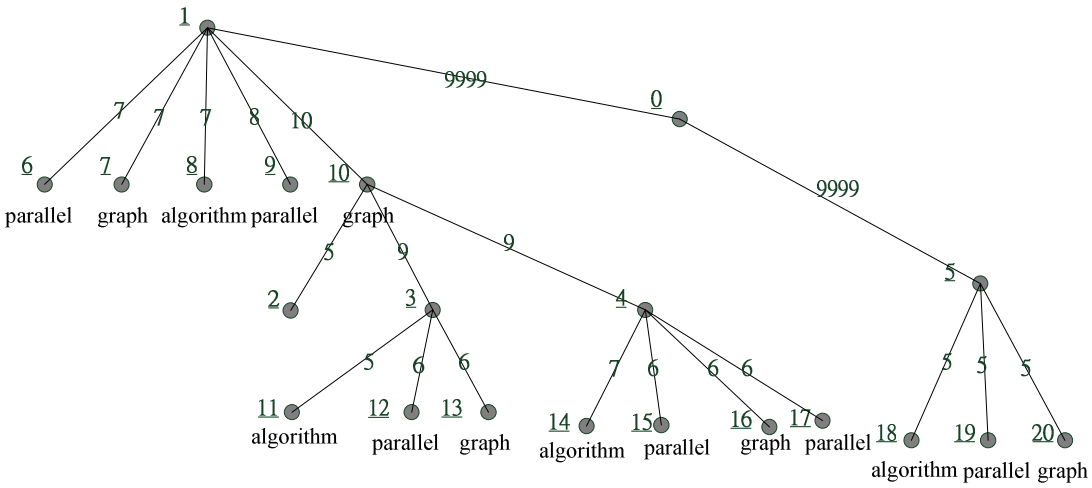

圖 3.4 範例查詢圖所生成的 HLDtree

3.1.4 Most-branch Tree (MB tree) ：

我們的作法是針對每個 ELCA 下的 match 決定其是否能成為 R-clique，我們希望提高分支的個數來提高 ELCA 樹的數量，以提升 Recall，同時希望 match 平均分配到各個 ELCA tree 下可以降低檢查 match 之間距離的時間。以下為大略的建樹步驟：

先對欲搜尋圖作 Breadth first search 建出一轉換樹，再將圖中剩下未使用的邊試著替換轉換樹的邊，如果能產生較多分支節點的樹則以新邊取代舊邊，同時以新轉換樹為基準繼續檢查是否有可替換邊能產生更多分支節點。直到產生該圖的最多分支轉換樹。以下列出建樹的演算法：

演算法名稱：Most-branch Tree (MB tree) :

輸入：connected graph CG

輸出：spanning tree T

```
L01:   T ← BFS(CG) // get BFS tree from CG
L02:   Eset ← the edges which are in G but not in T
L03:   foreach edge E in Eset
L04:      Tn ← add E into T
L05:      C ← the cycle found in Tn
L06:      foreach edge E' in C
L07:         T' ← the new tree which remove E' from Tn
L08:            if BranchCount(T') > T
L09:               Replace T with T'
L10:            end if
L11:      end foreach
L12:   end foreach
L13:   return T
```

圖 3.5 MB tree 演算法

參考圖 3.5，我們在 L01 對圖 CG 進行 Breath first search 建構一個轉換樹 T，之後在 L02 將在 CG 中沒有被 T 包含的邊存入邊集合 Eset 中，接著在 L03 到 L12 對 Eset 中的每一個邊進行檢查，在 L04 與 L05 將選中的邊 E 放入 T 中產生有一個環的新圖 Tn，並將產生的新的環(cycle)放入 C 中，在 L6 到 L11 中對 C 的邊一一檢查將每個邊去除產生的新轉換樹 T'是否能比 T 有更多分支節點，若有則將 T 以 T'取代並繼續判斷，此判斷條件可以讓我們得到有較多分支的轉換樹。直到原始圖 G 所能產生最多分支節點的轉換樹為止。注意到，L06-L11 的迴圈會針對每個 cycle 保留具有最多分支的樹，而 L03-L12 的迴圈則是檢查每個一開始

沒被建立到樹裡面的邊。

在此我們一樣以圖 2.3 作為範例說明 MB tree 的建立，首先我們必須先對原始圖進行 Breadth-first search 建構出 BFS tree，如圖 3.6 所示，此時分支的數量為 6，接著我們一一檢查沒有被加入到轉換樹內的邊，先選一個邊加入到目前的轉換樹中產生有一個環(cycle)的圖，接著再將環上的每個邊檢查，看拿掉一個邊之後的轉換樹是否比本來的轉換樹擁有更多分支點(Branch node)，若有則將有更多分支點的生成術取代本來的轉換樹，之後再繼續檢查其他的邊，直到可以檢查完所有邊為止，比較圖 3.6 及圖 3.7 我們可以發現邊{2, 10}被加入到轉換樹中而邊{1, 10}被剔除，導致節點 2 也變成分支節點。最後即可得到擁有圖 3.7 分支數是 7 的轉換樹。

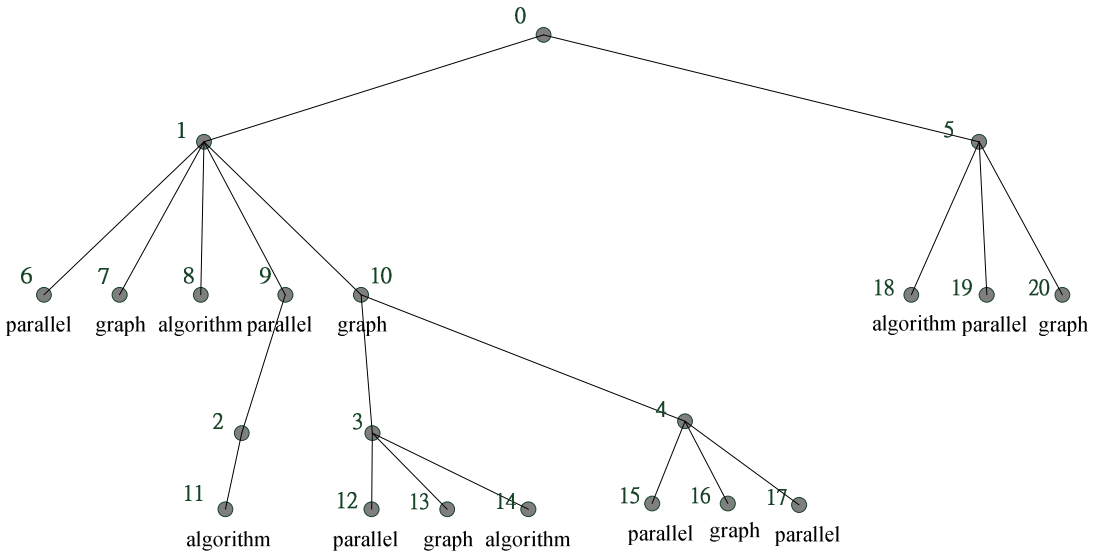

圖 3.6 範例查詢圖所生成的 BFS tree

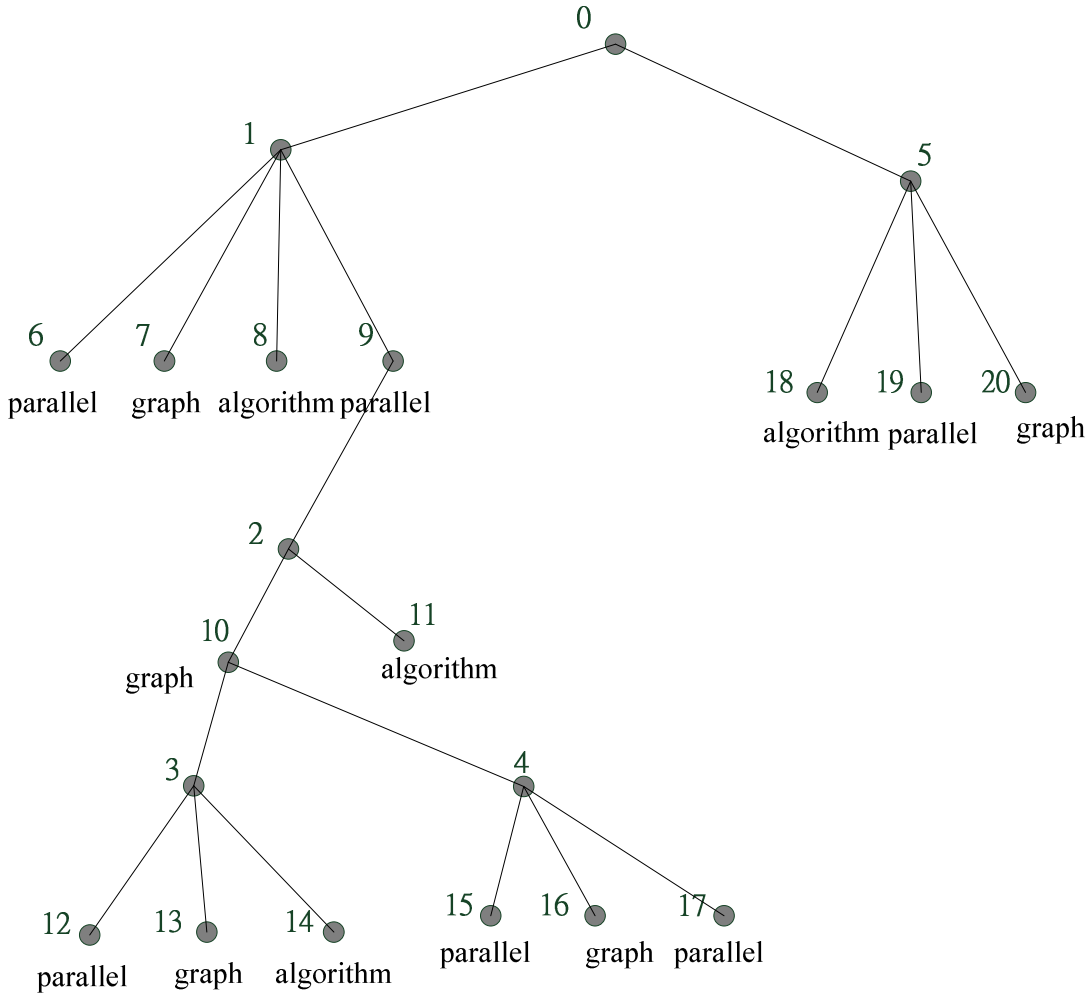

圖 3.7 範例查詢圖所生成的 MB tree

3.1.5 Breath First Search+ Tree (BFSP tree) ：

前述的四種方法只有考慮讓轉換樹包含原始圖的所有節點，但是沒有考慮到剩下沒有被畫入轉換樹的邊，為了避免在搜尋時無法考慮到沒有被畫入轉換樹的邊而找不到 R-clique，我們希望加入沒有被畫入轉換樹的邊並產生新的節點以解決這個問題。以下為建樹的大略步驟：

先對欲搜尋圖作 Breath first search 建出一轉換樹，再將圖中剩下未使用的邊依照節點編號的大小，找到該條邊在轉換樹中編號較小的節點，在該點加上編號較大的點作為新的子節點。以下我們列出演算法：

演算法名稱：Breath first Search+ Tree (BFS+ tree) : 輸入：connected graph CG 輸出：transformed tree T

```
L01:  T ← BFS(CG) // get BFS tree fromC G
L02:  Eset ← the edges which are in G but not in T
L03:  foreach edge E in Eset
L04:      add E into T to create new child node
L05:  end foreach
L06:  return T
```

圖 3.8 BFS+ tree 演算法

參考圖 3.8，在 L01 對圖 CG 進行 Breadth first search 建構一個轉換樹 T，之後在 L02 將在 G 中沒有被 T 包含的邊存入邊集合 Eset 中，在 L03 到 L05 將沒有使用到的邊依照節點編號的大小，在 T 中先找到節點編號較小的點作為父節點，並將編號較大的點在該點的位置新增新的子節點。重複此動作後至演算法結束。注意到轉換樹 T 的節點數量會變成圖 CG 的邊總數+1。假設圖 G 有 n 個節點 k 個邊，在演算法 L01 產生的 T 會有 n 個節點，而在演算法結束後會新增到 k+1 個節點。

我們再以圖 2.3 作為範例，一開始一樣必須先作 Breadth-first search 建構出圖 3.6 的 BFS tree，之後將沒有被加入到 BFS tree 中的邊以加入新子節點的方式一一加入到 BFS tree 中，最後可以得到圖 3.9 的 BFS+ tree，在圖 3.9 中，標示虛線的邊即為一開始沒有被加入到 BFS tree 中的邊。此種建立方式讓 BFS+ tree 能夠補強另外四種轉換樹無法考慮到未加入到轉換樹中的邊資訊的狀況。由於將圖 2.3 中所有的邊都加入到樹中，圖 3.9 的 BFS+ tree 的節點數量比起圖 2.3 的資料圖多出了 3 個。

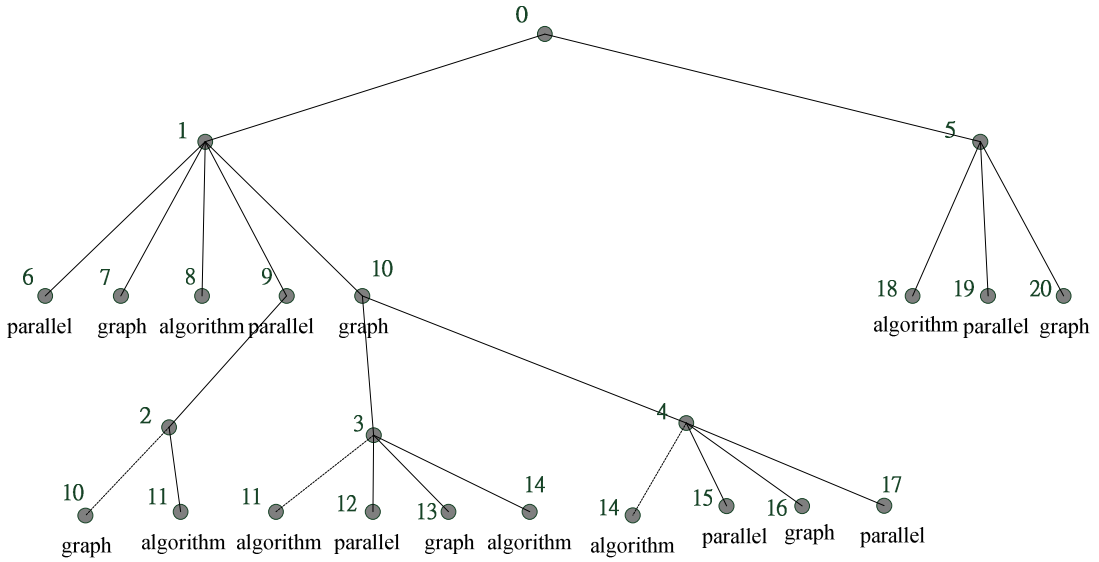

圖 3.9 BFS+ tree

總合而言，根據上述的五種策略，我們可以產生五種不同特性的轉換樹：(1) HD tree 以高 Degree 為優先建出較平坦的樹，(2) LD tree 以最低 Degree 為優先建出邊的距離較低的樹，(3) HLD tree 以最高 Degree 作為根節點，之後以低 Degree 為優先建出較平坦且第二層之後邊的距離較低的樹，(4) MB tree 建出分支最多的樹以期待能找出最多組 ELCA 結果，(5) BFSP tree 將前述四種策略未考慮到的邊再加到轉換樹中，避免轉換樹無法考慮到未加入邊的資訊。

3.2 查詢資料樹

將資料圖轉成轉換樹之後，我們必須對各個轉換樹建構[BCGL+12]提出的 ELCA 搜尋方法所會用到的編碼和 ELCA 索引，在此我們以圖 3.9 的 BFS+ tree 作為範例說明。

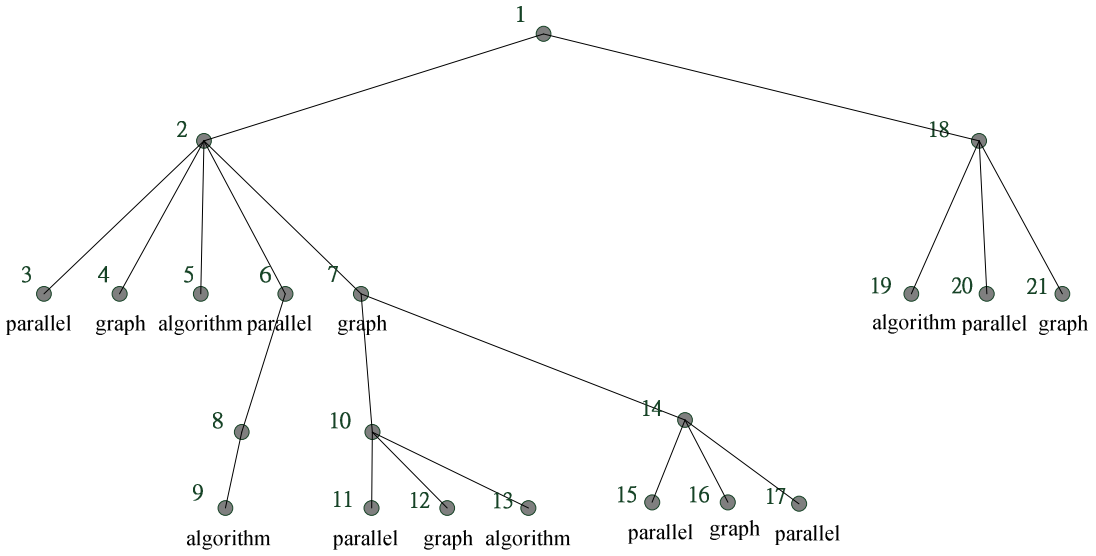

圖 3.10 重新編碼的 BFS tree

在[BCGL+12]提出的 ELCA 搜尋方法，樹的編碼必須以 Pre-order 順序編碼，故我們會對我們所轉換的樹重新編碼，並建立原始編碼與新編碼的對照表，以便在之後從樹中找到對應節點需要確定在原始圖中的距離時可以快速從表中取得原始的編號。

另外，ELCA 索引會針對每個查詢關鍵字個別建立出一個表，表中記錄了所有對應節點以及對應節點的祖先資訊。在表中有 4 個欄位，Pos 代表此表的第幾格、ID 為該節點的 Pre-order 編號、PIDPos 為該節點的祖先節點存在表中的第幾格，若是根節點則記為-1，N_{desc} 代表該節點底下含有幾個該關鍵字的對應節點。

我們以圖 3.10 為範例建立 parallel, graph, algorithm 這三個關鍵字的 ELCA 索引：

parallel

Pos	0	1	2	3	4	5	6	7	8	9	10
ID	1	2	3	6	7	10	11	14	15	18	19
PIDPos	-1	0	1	1	1	4	5	4	7	0	9
N_{desc}	5	4	1	1	1	1	1	1	1	1	1

graph

Pos	0	1	2	3	4	5	6	7	8	9
ID	1	2	4	7	10	12	14	16	18	20
PIDPos	-1	0	1	1	3	4	3	6	0	8
N_{desc}	5	4	1	3	1	1	1	1	1	1

algorithm

Pos	0	1	2	3	4	5	6	7	8	9	10
ID	1	2	5	6	8	9	7	10	13	18	21
PIDPos	-1	0	1	1	3	4	1	6	7	0	9
N_{desc}	4	3	1	1	1	1	1	1	1	1	1

表 3.3 範例 ELCA index

參照圖 3.10 及表 3.3，樹中對應關鍵字 parallel 的對應節點共有 5 個，而從根節點開始記錄 ID 為 1 ，PIDPos 設為-1，底下含有對應節點數為 5 故 N_{desc} 記錄 5，接著記錄節點 2，ID 為 2、PIDPos 為表中 Pos 為 0 的 ID1 故記錄 0，底下含有對應節點數為 4 故 N_{desc} 記錄 4，以此類推便可得到圖 3.11 的 ELCA 索引。利用建立好的 ELCA 索引，[BCGL+12]提出的 ELCA 搜尋方法 FwdELCA 會將需

要查詢的對應資料塞入一個堆疊中，再依照[BCGL+12]中所證明的特性，觀察在堆疊中的 N_{desc}，便可判斷該節點是否為 ELCA 節點，然後將該節點下的對應節點取出。

但是，為了要回傳 R-clique 的結果，我們必須對每個 ELCA 樹再作檢查和篩選。針對一個被回傳的 ELCA 樹，樹中的節點必包含所有查詢關鍵字。注意到在樹中節點之間的距離，為由其中一個對應節點到 ELCA 樹的根節點，再由根節點到另一個對應節點，而且我們假設該距離即為其間最短距離。根據 R-clique 的定義，對應節點 1 到對應節點 2 的最短距離必須小於 R，而我們以 R/2 作為瓶頸值來判斷該節點是否可形成 R-clique。也就是，假設一個對應節點 1 到 ELCA 樹的根節點的距離為 $Dist_1$、對應節點 2 到 ELCA 樹的距離為 $Dist_2$，我們的判斷條件為 $Dist_1 < R/2$ 且 $Dist_2 < R/2$，我們由此可推出 $Dist_1 + Dist_2$ 必小於 R ，所以符合 R-clique 兩點之間距離必須小於 R 的限制。

以下列出我們的演算法：

演算法名稱： EFFR 輸入：資料樹 T(V, E), 查詢句{k_1 , k_2 , ... , k_m}共 m 個關鍵字, 距離限制 R 輸出：所有符合查詢句的 R-clique
L01: $E_S \leftarrow$ FindELCAset(T, k_1, …, k_m) L02: rList ← empty L03: foreach ELCA tree E_A in E_S L04: M ← all match keyword nodes in E_A L05: for i ← 1 to size(M) do L06: if dist(M_i, E_A's root) < R/2 L07: RSet[j] ← M_i // if Mi is match to jth keyword L08: end if L09: RCset ← CombineRclique(RSet[1], Rset[2], …, Rset[m]) L10: foreach R-clique RC in RCset L11: ERtreeSet.add ← GetERtree(RC, E_A) L1: return ERtreeSet

圖 3.11 ELCA Fast Find Rclique (EFFR)演算法

EFFR 演算法主要步驟如下：L01 到 L03 找出在資料樹中的所有 ELCA 樹，在 L04 到 L08，我們檢查在 ELCA 樹下的所有對應節點到該 ELCA 樹的根節點的距離，如果小於 R/2，我們便將該對應節點放入一個集合 RSet，注意在 L05 的 size(M)是指 M 的總個數而非群數。在 L09 中 RSet 已經存放了一個 ELCA 樹下所有可能包含於 R-clique 的節點，依照我們前述的證明我們知道這些節點所組合成的叢集(clique)必為 R-clique，我們呼叫 CombineRclique 函式直接將所有節點依照他們的關鍵字組合成一組組的 R-clique，若有一個節點同時符合多個關鍵字，則將該節點會被放入所有符合關鍵字的節點群中。再將剛剛所組合得到的 R-clique 存放入 RCset，在此我們可以從 RCset 中取得由演算法所找到的一個 ELCA 樹下的 R-clique，之後在 L10 及 L11 我們對 RCset 中的各個 R-clique RC 呼叫 GetERtree 函式取得該組 R-clique 的 ER tree。將得到的 ER tree 存入 ERtreeSet 中，最後回傳 ERtreeSet 作為搜尋結果。在程式中，我們以二元樹 (左子右弟)的結構來儲存 ELCA 樹的所有節點，而在 GetERtree 函式中，我們對 ELCA 樹 E_A 以 R-clique RC 的點一一進行遍歷搜尋，每一次搜尋即可得到一個對應節點到 ELCA 樹根節點的路徑，對 m 個對應節點搜尋之後便可以得到該 R-clique 的 ER tree。在時間複雜度的部分，在 L05 到 L08 必須檢查| M |個邊。在 L09，我們假設 M_{MAX} 為|RSet[1]|到|RSet[m]|中的最大值，則此演算法的時間複雜度為 $|M|+|M_{MAX}{}^{m}|$。在一個 ELCA 樹 E_A 中，假設 E_A 有$|T_E|$個節點，則對一個對應節點搜尋得到路徑的時間複雜度為$|T_E|$，而一個 R-clique 總共有 m 個對應節點，建出一個 ER tree 的時間複雜度為 $m*|T_E|$。

然而單純使用 ELCA 節點與對應節點之間的距離來判斷會導致一部分的解無法由此方法搜尋出，以下我們畫出範例圖分析 EFFR 演算法的缺點：

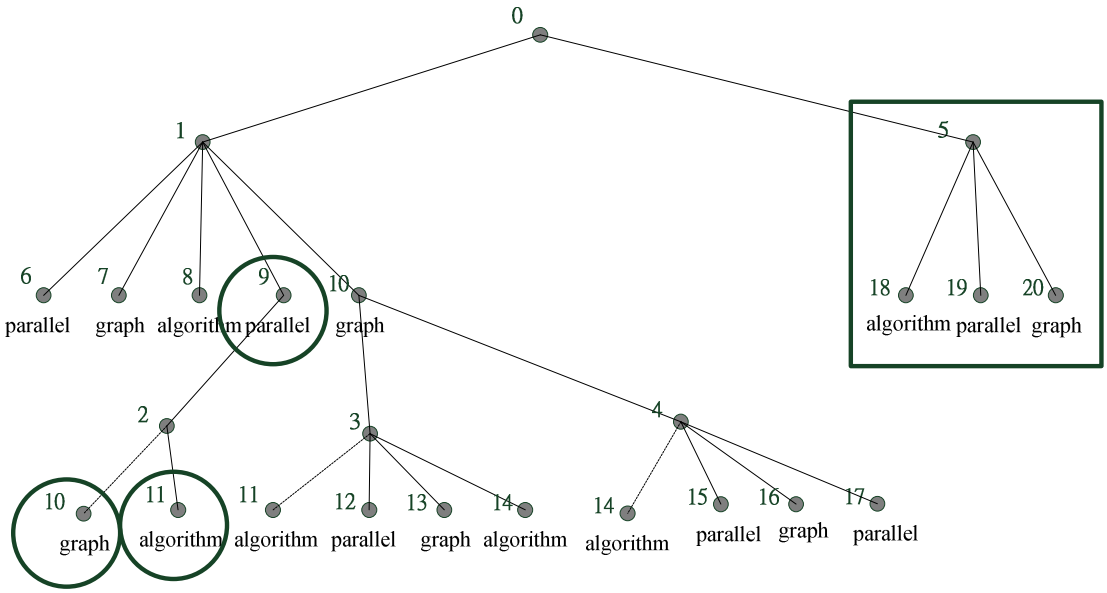

圖 3.12 範例查詢圖的 BFS+ tree 查詢召回範例

參考圖 3.12，該圖為範例查詢圖的 BFS+ tree，我們可發現{9, 10, 11}此組 R-clique 無法回傳，其原因為當存在對應節點到 ELCA 節點的距離超過 R 的一半時，雖然該對應節點實際最短距離和其他對應節點的距離小於R，仍然無法回傳。譬如在此例子中，ELCA 節點為 9，而對應節點 10 到節點 9 在樹上的距離為 12，已經超過範例所定的 R 的一半，故此種狀況無法被 EFFR 演算法回傳。

為了解決上述狀況，我們在下一章提出改善召回率的改良演算法。

第四章 查詢改良方法

前述對 ELCA 樹快速搜尋 R-clique 的方法在搜尋時，必須是各個對應節點的最短路徑正好在對應的 ELCA 樹中才能正確被搜尋到。針對這點，我們提出新的改良方法來提升每一種轉換樹的搜尋 Recall。

這裡我們採用將 ELCA 樹中所有的對應節點依照關鍵字分群之後套用 Branch and Bound 演算法來直接檢查兩點間實際的最短距離。相較於原始的 Branch and Bound 演算法，在一個 ELCA 樹下可以有效減少 Branch and Bound 演算法所需要計算的節點量。假設一個資料圖中 C_{MAX} 為擁有最多對應節點的集合，且假設在資料圖被轉換成樹後產生了兩個 ELCA 樹將所有對應節點平分成兩群，則一個 ELCA 樹下擁有最多對應節點的集合的節點個數會變為 $\frac{1}{2}|C_{MAX}|$，則對兩個 ELCA 樹進行 Branch and Bound 演算法的計算量會變成 $2 * m^2 | (\frac{1}{2} C_{MAX})^{m+1}|$，其值小於原始 Branch and Bound 演算法所需的 $m^2 | (C_{MAX})^{m+1}|$。

在正式介紹改良演算法之前，我們將原始的 Branch and Bound 演算法修改成如下的 BABR 演算法：

演算法名稱：BABR 輸入：對應 m 個關鍵字的對應節點集合 $M_1, M_2,\ldots M_m$, 距離限制 R 輸出：所有符合搜尋的 R-clique
L01: for i ← 1 to size(M_m) do L02: rList.add(M_i^1) L03: for i ← 2 to m do L04: newRList ← empty L05: for j ← 1 to size(M_i) do L06: for k ← 1 to size(rList) do L07: if ∀ node ∈ $rList_k$ dist(node, M_i^j) ≤ r (where $rList_k$ is the kth element of rList) then L08: newCandidate ← M_i^j .concatenate(node) L09: $newRList_k$ .add(newCandidate) L10: rList ← newRList

L11: return rList

圖 4.1 BABR 演算法

BABR 演算法在 L01 到 L11 的步驟皆和 Branch and Bound 演算法的檢查節點部分相同，有差別的是在一開始輸入的節點已經按照查詢句的關鍵字分群完畢，而不像 Branch and Bound 演算法需要從資料圖中搜尋對應節點。

以下列出我們的 EAFR 演算法：

演算法名稱：EAFR 輸入：資料樹 T, 關鍵字{k_1 , k_2 , ... , km}共 m 個關鍵字, 距離限制 R 輸出：所有符合搜尋的 R-clique
L01: E_S ← FindELCAset(T, k_1, …, k_m) L02: rList ← empty L03: foreach ELCA tree E_A in E_S L04: {M_1, M_2,…,M_m} ← all match keyword nodes in E_A L05: RCset.add (BABR(M_1, M_2,…,M_m ,R)) L06: foreach R-clique RC in RCset L07: ERtreeSet.add (GetERtree(RC)) L08: return ERtreeSet

圖 4.2 ELCA Accurate Find Rclique (EAFR) 演算法

ELCA Accurate Find Rclique (EAFR) 演算法大略可分為下列兩個階段：

1. 對轉換樹 T 找出所有的 ELCA 樹
2. 對每個 ELCA 樹，套用 BABR 演算法來尋找所有的 R-clique，並將 BABR 演算法找到的所有 R-clique 集中到一個集合中，之後呼叫 GetERtree 得到每個 R-clique 的 ERtree 作為搜尋結果回傳。

開始處理查詢之後我們在 L01 對轉換樹 T 進行 ELCA 搜尋得到多個 ELCA 樹，並將所有 ELCA 樹存入集合 E_S 中準備作後續的判斷。在 L02 建立一個空的 R-clique 集合 rList 作為儲存找到的 R-clique 集合。從 L03 開始我們對 ELCA 樹集合 E_S 開始作 R-clique 的判斷，在 L03~L05 我們將集合 E_S 中的每個 ELCA 樹

E_A 取出，在 L04 我們將 E_A 中所有符合查詢的對應節點存到對應到每個關鍵字的集合 M_j 中，在 L05 再將所有對應節點集合和限制 R 套用 BABR 演算法搜尋出存在 E_A 裡的 R-clique，再將找到的 R-clique 存入 rList 中，在 L06 及 L07 我們將找到的 R-clique 呼叫 GetERtree 函式得到該 R-clique 的 ER tree 並存入 ERtreeSet 中，最後在 L08 確定已經檢查完所有 ELCA 樹並得到所有 ER tree 之後將 ER tree 集合 ERtreeSet 回傳即可取得搜尋的 R-clique 結果。假設 M_{MAX} 為擁有最多對應節點的集合，我們的演算法的時間複雜度為 $O(m^2 |M_{MAX}^{m+1}|)$。

以下我們畫出範例圖分析 EAFR 演算法查詢結果以及對不同轉換樹的搜尋差異：

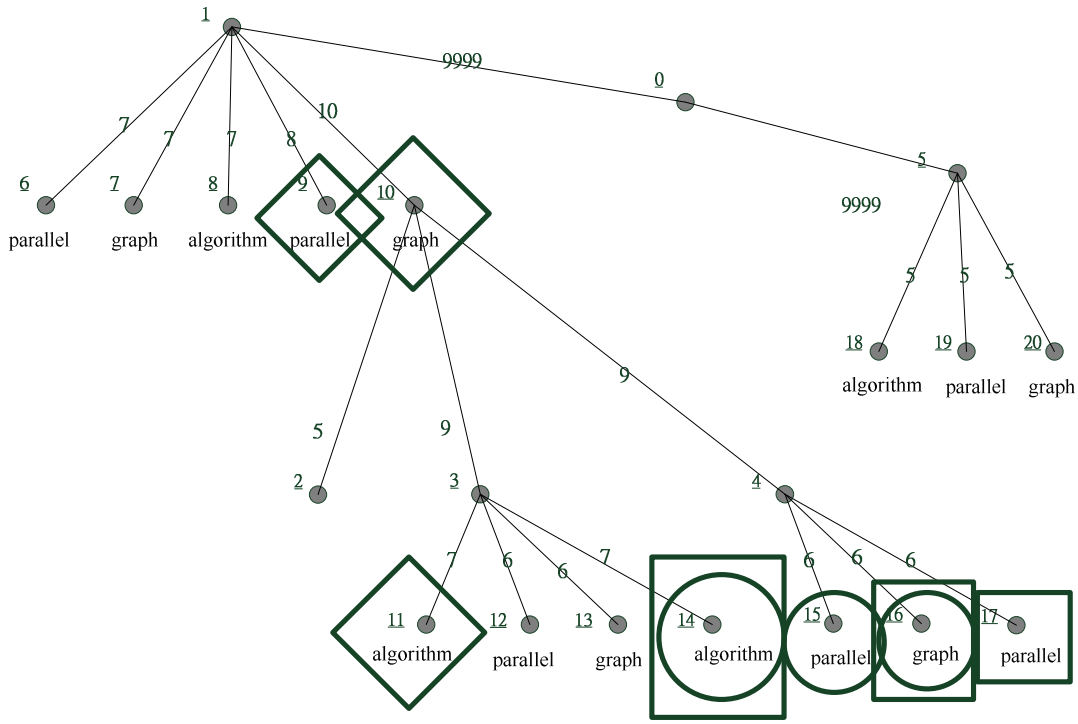

圖 4.3 範例查詢圖的 HD tree 查詢召回

首先是 HD tree，參考圖 4.3，HD tree 的建構條件為優先選擇 Degree 較高的點，而這點使一個點下包含的對應節點數較多。對照圖 2.3 及圖 2.4 以及查詢句 TQ1 可以發現 HD tree 可以回傳{6, 7, 8}，{11, 12, 13}，{12, 13, 14}，{18, 19, 20}四組 R-clique。而無法回傳{9, 10, 11}，{14, 15, 16}，{14, 16, 17}三組 R-clique。圖中被相同圖形框起的節點為無法被回傳的一組 R-clique。這三組 R-clique 無法被回傳主要是因為同一 R-clique 下的對應節點被分配到不同的 ELCA 樹下，導致 EAFR 演算法無法找到這三組組合。

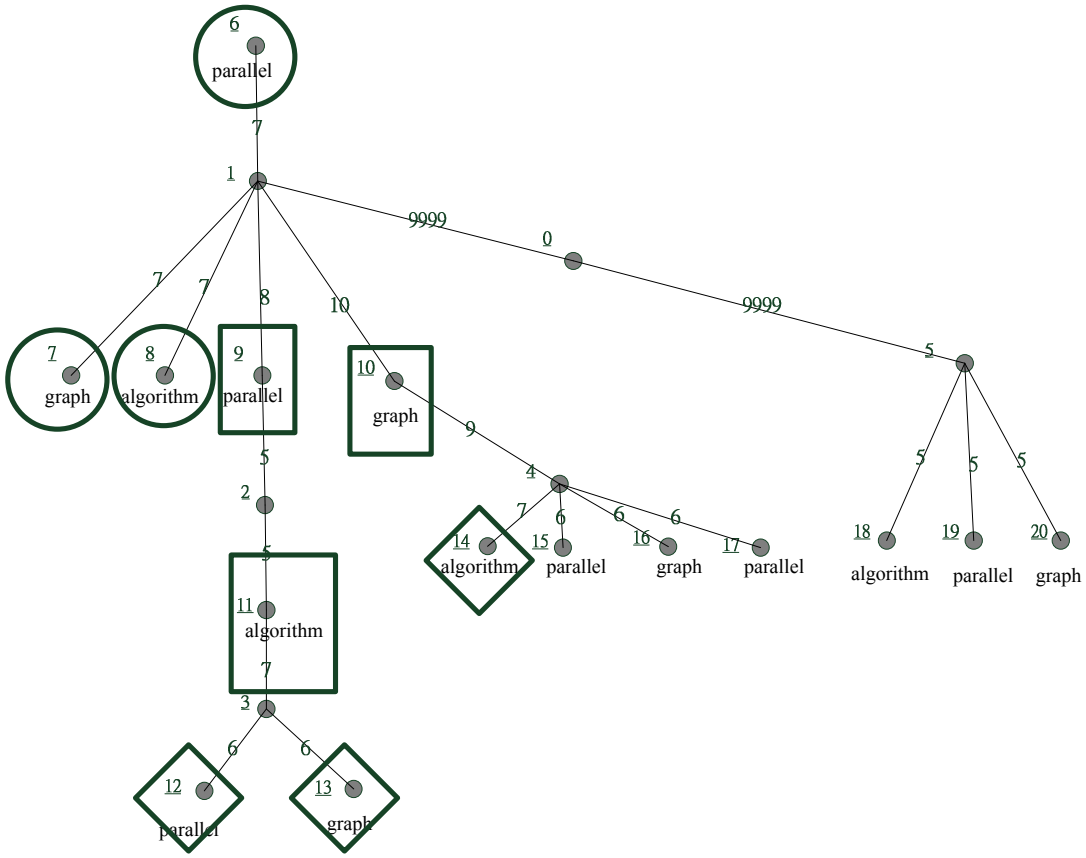

圖 4.4 範例查詢圖的 LD tree 查詢召回

接著我們分析 LD tree，參考圖 4.4 由於 LD tree 是以低 Degree 的點為優先建樹，導致建出來的樹深度較深，且一個父節點所擁有的子節點較少，而使所有 ELCA 樹能包含到的對應節點總數較少。由圖 4.4 可以看到，LD tree 可以回傳{11, 12, 13}，{14, 15, 16}，{14, 16, 17}，{18, 19, 20}四組 R-clique。而無法回傳{6, 7, 8}，{9, 10, 11}，{12, 13, 14}三組節點。圖中以同樣形狀的圖型被框起的節點為同一個無法被回傳 R-clique 的對應節點，參考{12, 13, 14}，由於節點 14 被分到節點 4 的 ELCA 樹下，導致此組 R-clique 無法被回傳，而節點 11,12,13 形成了一個 ELCA 樹，使得{9, 10, 11}此組解無法被找到，而以節點 1 為根節點的 ELCA 樹包含了 7, 8, 9, 10 四個對應節點，導致{6, 7, 8}此組解也無法被找到。由於這些對應節點被分配到不同的 ELCA 樹下，故在 LD tree 無法被回傳。

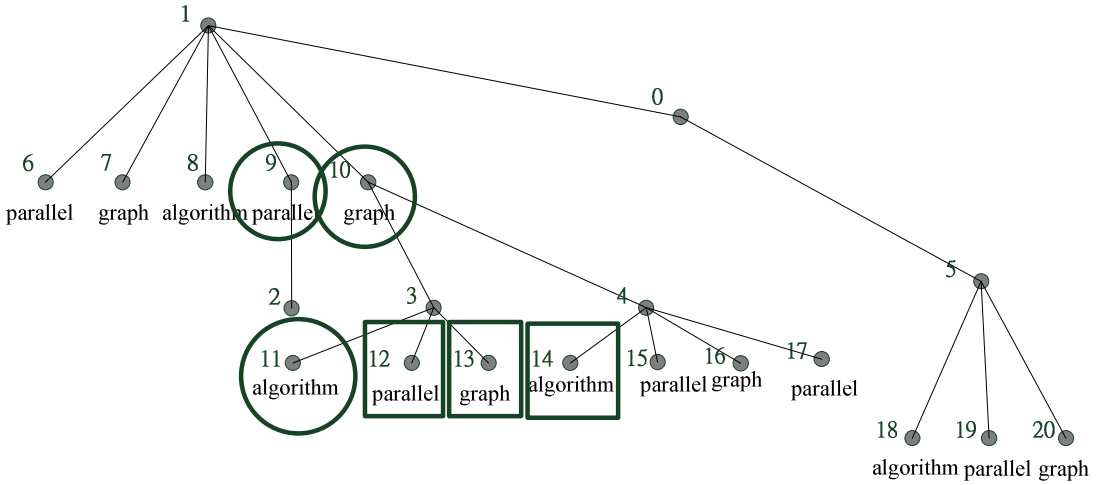

圖 4.5 範例查詢圖的 HLD tree 查詢召回

接著我們分析圖 4.5，比較 HD tree 與 HLD tree，由於 HD tree 的特性是由節點高的點優先建樹而建出較平坦的樹，而 HLD tree 的特性是根節點的 Degree 高，其他節點在建構時會交錯選擇 Degree 較低和 Degree 較高的點，這種特性使節點的分配與 HD tree 不同。而在此例子 HLD tree 可以回傳{6, 7, 8}，{18, 19, 20}，{14, 15, 16}，{14, 16, 17}，{18, 19, 20}五組解，而{9, 10, 11}、{12, 13, 14}這兩組解也因為對應節點被分配到不同的 ELCA 樹下導致無法被回傳。

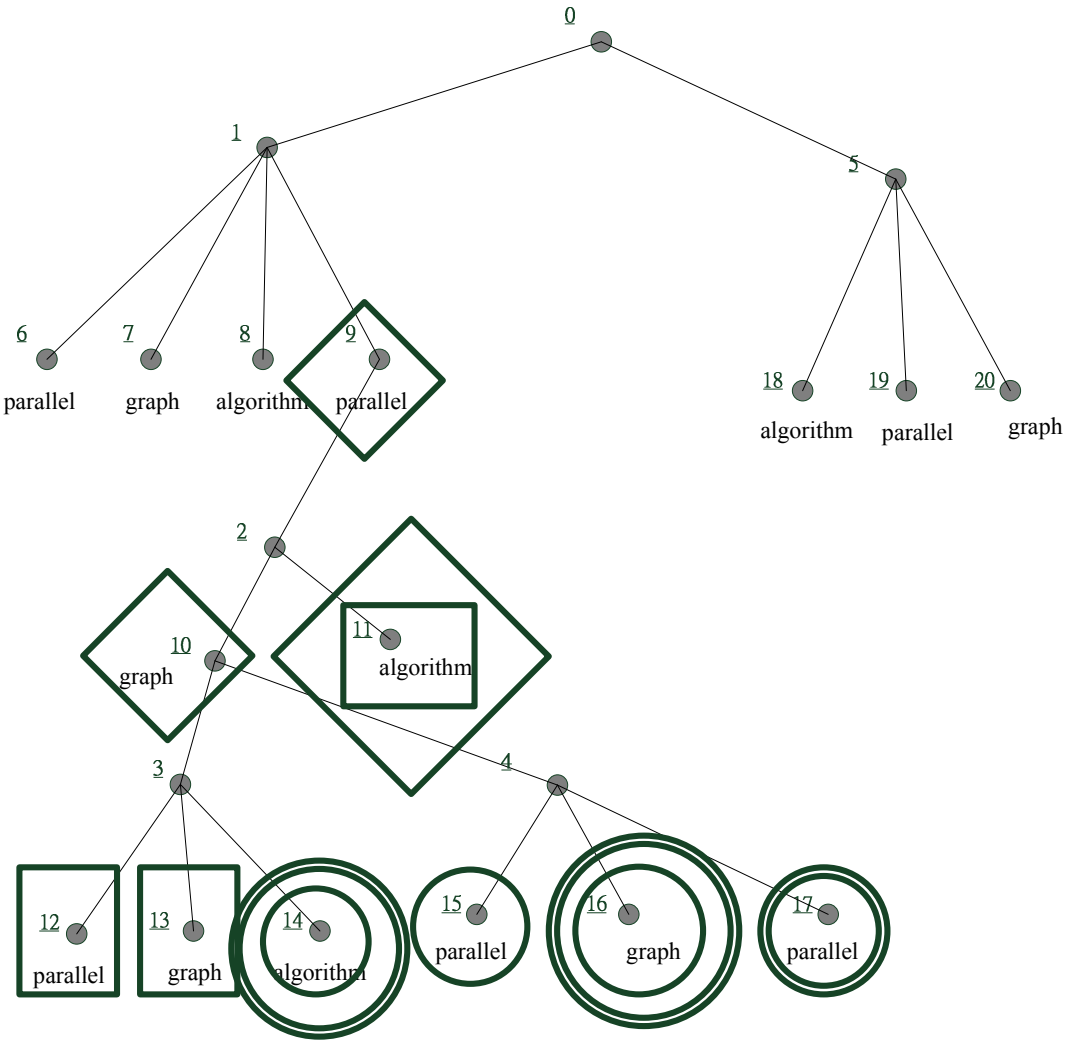

圖 4.6 範例查詢圖的 MB tree 查詢召回

接著我們分析 MB tree，參考圖 4.6，{6, 7, 8}，{12, 13, 14}，{18, 19, 20}三組 R-clique 可以被回傳，然而節點{11, 12, 13}由於節點 11 並不在節點 12,13 的 ELCA 樹下而無法被回傳，而{9, 10, 11}，{14, 15, 16}，{14, 16, 17}三組 R-clique 也是因為同一組 R-clique 的對應節點沒有被分在同一個 ELCA 樹下而無法被回傳。

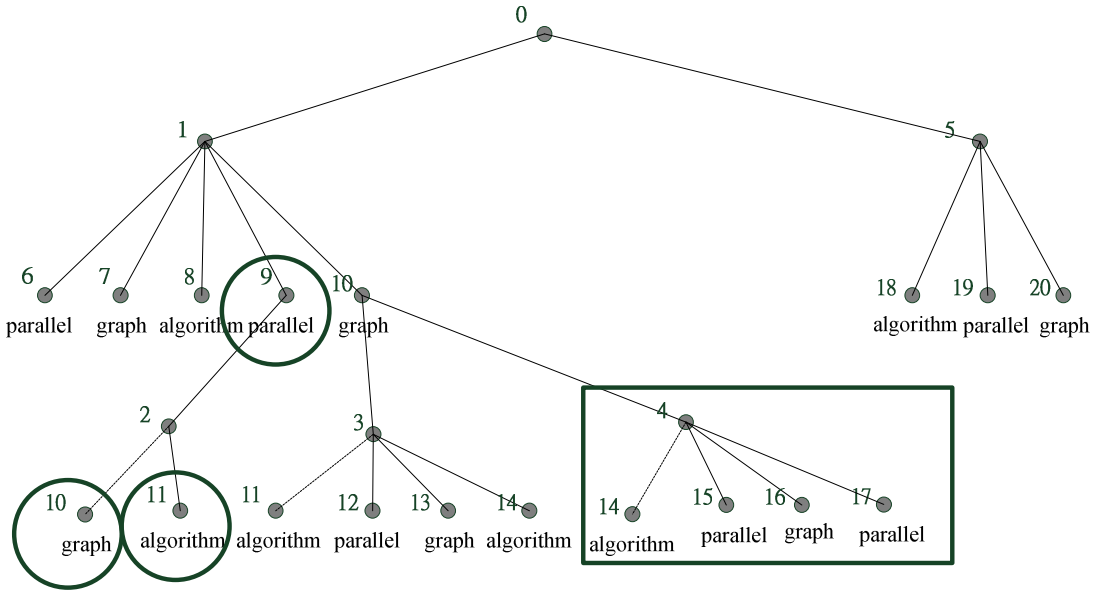

圖 4.7 範例查詢圖的 BFS+ tree 查詢召回

最後我們分析 BFS+ tree，參考圖 4.7，在此範例中 BFS+ tree 恰好能將原始圖的所有 R-clique{6, 7, 8}，{9, 10, 11}，{11, 12, 13}，{12, 13, 14}，{14, 15, 16}，{14, 16, 17}，{18, 19, 20}全部回傳。首先參考圖中被方框框起的部分，R-clique{14, 15, 16}以及{14, 16 ,17}這兩組解在前四種轉換樹經常出現對應節點被分配到不同 ELCA 樹的狀況，但是在 BFS+ tree 中，由於原始圖的所有邊都被加入到樹中，而使節點 14 能夠在兩個不同的 ELCA 樹下出現，而讓此兩組解在 BFS+ tree 能夠被回傳。而圖中被圓圈框起的{9, 10, 11}此組解也因為 BFS+ tree 有加入所有原始圖邊的特性而讓節點 10, 11 出現在節點 9 下形成一個 ELCA 樹，而使此組解能夠被回傳。

第五章 實驗

在本章節中，我們以實驗比較三個演算法 Branch and Bound、EFFR、EAFR 的效率和查詢召回率，以及比較我們提出的五種轉換資料樹。首先，先說明我們進行實驗的環境，我們以個人電腦作為實驗的環境，其 CPU 為四核心的 Intel i7 2600，其中每個核心的時脈為 3.4GHz，記憶體為 16GB，所採用的作業系統為 Windows 7 企業版 SP1。以上述的實驗環境我們以本論文所提出的各種轉換轉換樹的方法與[KA11]所提出的 Branch and bound 演算法進行查詢效率和召回率的比較。針對前述的鄰居索引的部分，為了避免鄰居索引所需要的儲存空間太大以及建置的時間過久，我們只儲存 R<10 的最短路徑資料，當距離大於 10 時我們便不記錄，而在需要時取出邊並利用其 Degree online 計算。在測試資料方面我們選擇以儲存投稿論文的 DBLP 資料集作為實驗用的測試資料。我們選用 DBLP 年份在近五年(2008~2013)的資料，XML 原始檔案大小為 321MB，轉換成圖後節點總數為 678620 個，邊的總數為 3415726 個。轉換圖的方是我們依照[KA11]的方法，將 XML 檔轉換成 author，title，cite 三個表格，並將作者撰寫論文的關係建立成 foreign key (title 和 cite 之間亦建立)，然後將每個 tuple 視為一個節點，一個 foreign key 到另一個 tuple 視為一個邊到另一個節點的連結，而邊距離的計算方式我們參照[KA11]的定義，假設有兩相鄰節點 u、v，則 u 到 v 的距離為$(\log_2(1 + u_{deg}) + \log_2(1 + v_{deg}))/2$，其中 u_{deg} 為 u 的 Degree，v_{deg} 為 v 的 Degree。

以下我們列出實驗使用的查詢句以及關鍵字的出現頻率：

出現頻率	關鍵字
0.0001~0.0003	genes, trees, paths
0.0003~0.0006	protocol, fast, retrieval
0.0006~0.0009	programming , scheme, scheduling, environments, parallel
0.0009~0.0012	framework , modeling, problem, methods, selection, sensor
0.0012~0.015	optimization, dynamic, modeling, problem
0.015~	cloud, network, control, detection, performance

表 5.1 關鍵字出現頻率

編號	資料集	關鍵字
TQ1	DBLP	gene, trees
TQ2	DBLP	programming, parallel, environments
TQ3	DBLP	optimization, dynamic, framework
TQ4	DBLP	network, control, modeling
TQ5	DBLP	network, detection, performance
TQ6	DBLP	protocol, fast, retrieval
TQ7	DBLP	scheme, scheduling, hybrid, communication
TQ8	DBLP	methods, selection, sensor, imperfect
TQ9	DBLP	cloud, modeling, problem

表 5.2 查詢句 TQ1 ~ TQ9

我們在表 5.2 列出我們在實驗所使用的查詢句，查詢句 TQ1~TQ5 是將較有關連性的關鍵字組合成一個查詢句，在後面的實驗作為比較召回率的參考。查詢句 TQ6~TQ9 主要是依照關鍵字的出現頻率來分群，為了不讓查詢輸出的結果太多導致難以分析詳細的資料，我們在 TQ9 加入了一個屬於不同組關鍵字頻率的關鍵字來抑制 TQ9 輸出的結果總數。

以下我們列出以前述的查詢句所作出的實驗結果比較 EFFR 與 EAFR 兩種方法的查詢效率與 Recall，在查詢回傳率的比較上，我們以 Branch and Bound 演算法所找出的 R-clique 中所包含的對應節點作為比較標準，以 Branch and Bound 演算法所找出的 R-clique 總數量作為分母，若我們的方法回傳了一個一樣的 R-clique 節點則將分子加一，比較回傳的搜尋結果能回傳多少比例的 R-clique。

5.1 召回率的評估

我們首先比較 Branch and Bound 演算法與 EFFR 演算法的搜尋召回率，使用的查詢句為 TQ1~TQ5，為了讓查詢結果適合人工判別召回率，我們試著將輸出的答案組數盡量控制在數十組到兩百組之間，故我們將 R 設定大小為 7：

查詢句	B&B	HD	LD	HLD	MB	BFS+
TQ1	100% (0.224 s)	0.00% (0.02 s)	0.00% (0.029 s)	0.00% (0.009 s)	5.88% (0.023 s)	65% (0.312 s)
TQ2	100% (4.737 s)	0.00% (0.208 s)	0.00% (0.256 s)	0.00% (0.097 s)	68.18% (0.241 s)	98% (4.975 s)
TQ3	100% (23.953 s)	0.00% (0.793 s)	0.38% (0.965 s)	0.00% (0.349 s)	13.41% (1.264 s)	77.01% (20.706 s)
TQ4	100% (61.043 s)	0.57% (1.503 s)	0.71% (1.617 s)	0.57% (0.658 s)	10.65% (2.309 s)	73.72% (42.83 s)
TQ5	100% (43.874 s)	0.00% (1.340 s)	0.47% (1.443 s)	0.24% (0.523 s)	9.00% (1.966 s)	81.41% (38.18 s)

表 5.3 EFFR 演算法查詢召回率比較及查詢時間(sec)

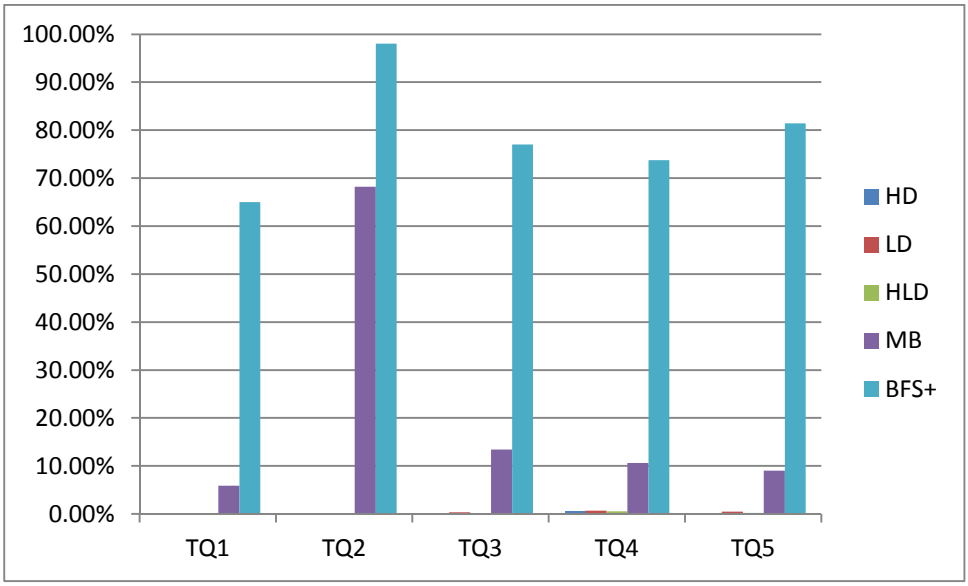

圖 5.1 EFFR 演算法查詢召回率比較

由圖 5.1 及表 5.3 可得知，除了 BFS+ tree 以外其他四種轉換樹的查詢召回率並不高，主要是因為 BFS+ tree 以外的四種轉換樹沒有辦法考慮原始圖中所有邊的資訊，另外在當 R-clique 的節點在轉換樹中的祖先不同時，也會導致 EFFR 演

算法無法找到該 R-clique。而 BFS+ tree 由於在轉換樹時將原始資料圖中所有的邊加入到轉換樹中，任一個點可隸屬於多個 ELCA 樹，故在搜尋結果上較其他四種樹好，但是當對應節點經過 ELCA 樹的根點的距離超過限定值時，該 R-clique 便無法以此種方法回傳。

接著我們比較 EAFR 演算法的召回率，同樣設定 R 的大小為 7：

查詢句	B&B	HD	LD	HLD	MB	BFS+
TQ1	100% (0.224s)	76.47% (0.031s)	35.29% (0.029s)	70.59% (0.042s)	94.12% (0.19s)	100% (0.236s)
TQ2	100% (4.737s)	54.55% (0.442s)	77.27% (0.224s)	84.09% (0.699s)	97.73% (4.34s)	100% (4.482s)
TQ3	100% (23.953s)	38.70% (1.355s)	47.51% (0.720s)	53.64% (2.169s)	89.66% (18.00s)	99.62% (19.389s)
TQ4	100% (61.043s)	47.59% (2.41s)	40.63% (1.236s)	51.14% (4.04s)	89.35% (38.54s)	99.01% (36.849s)
TQ5	100% (43.874s)	43.88% (1.658s)	45.29% (0.847s)	51.53% (2.746s)	90.00% (29.42s)	99.65% (30.402s)

表 5.4 EAFR 演算法查詢召回率比較及查詢時間(sec)

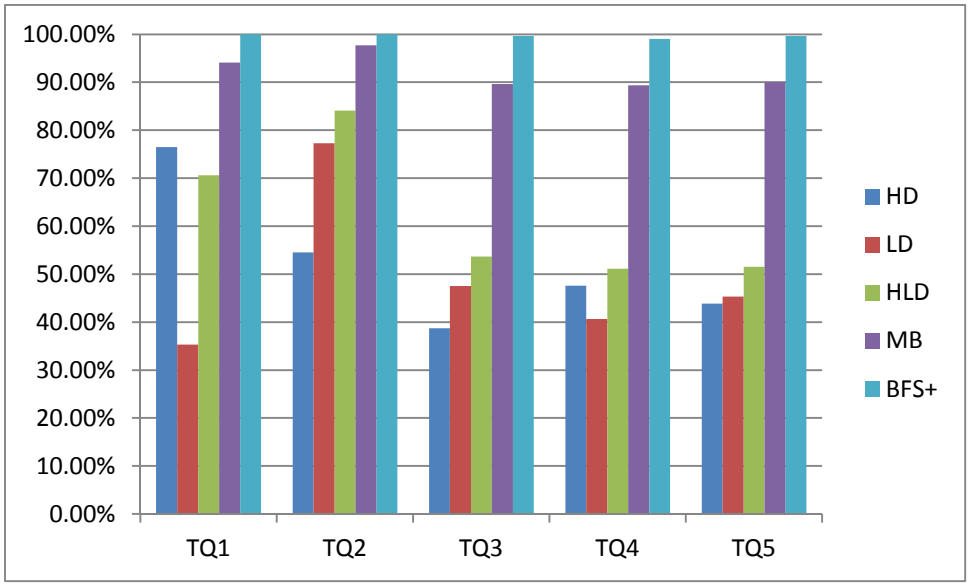

圖 5.2 EAFR 演算法查詢召回率比較

由於 EAFR 演算法在檢查時是針對每個對應節點之間在原始圖的最短路徑進行，這點在檢查 R-clique 時能避免 EFFR 演算法無法找到最短路徑不是經由 ELCA 樹的根節點的缺點，只要當 ELCA 樹下有包含一個 R-clique 的所有對應節點時，該 R-clique 便可以被 EAFR 演算法找到，而有比較高的召回率。

以下我們用一個例子解釋一個 R-clique 在 EFFR 演算法無法被回傳而在 EAFR 演算法可以被回傳的狀況：

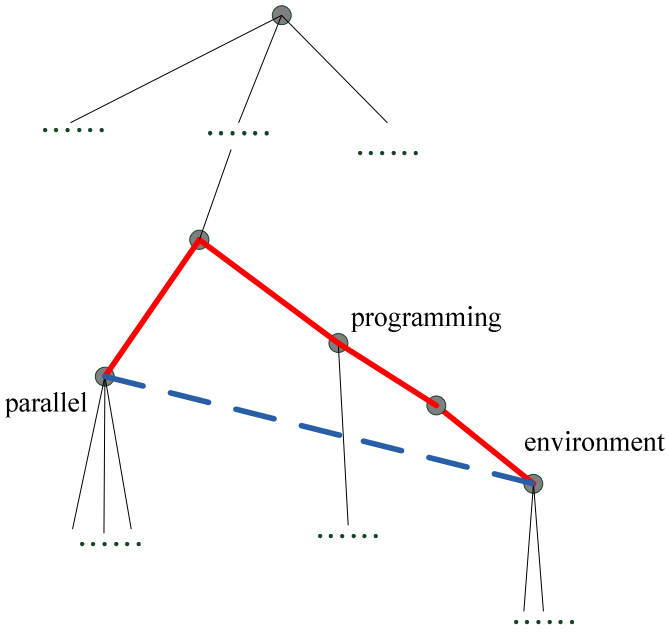

圖 5.3 EFFR 演算法無法回傳的解

參考圖 5.3，圖中 parallel、programming、environment 三個點為同一個 R-clique 的對應節點，圖中粗體實線為三個對應節點在轉換樹中的連接邊，而粗體的虛線代表 parallel 和 environment 兩個節點在原始圖的最短路徑連線。在 EFFR 演算法中只能檢查到圖中粗體實線的邊，而無法考慮 parallel 和 environment 兩個節點在原始資料圖的實際最短距離。但是在 EAFR 演算法中，當檢查到 parallel 和 environment 兩個節點時，判斷距離的邊是考慮原始圖的最短距離而非轉換樹上的連結邊。故當轉換樹上的連結邊距離大於 R 而原始圖的實際距離小於 R 時，EFFR 演算法無法找到此組 R-clique 而 EAFR 演算法可以找到。這點特性使得 EAFR 演算法的召回率較 EFFR 演算法高。

但是 EAFR 演算法仍然有部分無法回傳的解，我們再針對 EAFR 演算法所沒找到的 R-clique 解進行分析發現在以下的狀況出現時，EAFR 演算法無法找到 R-clique，我們以 HD tree 為例：

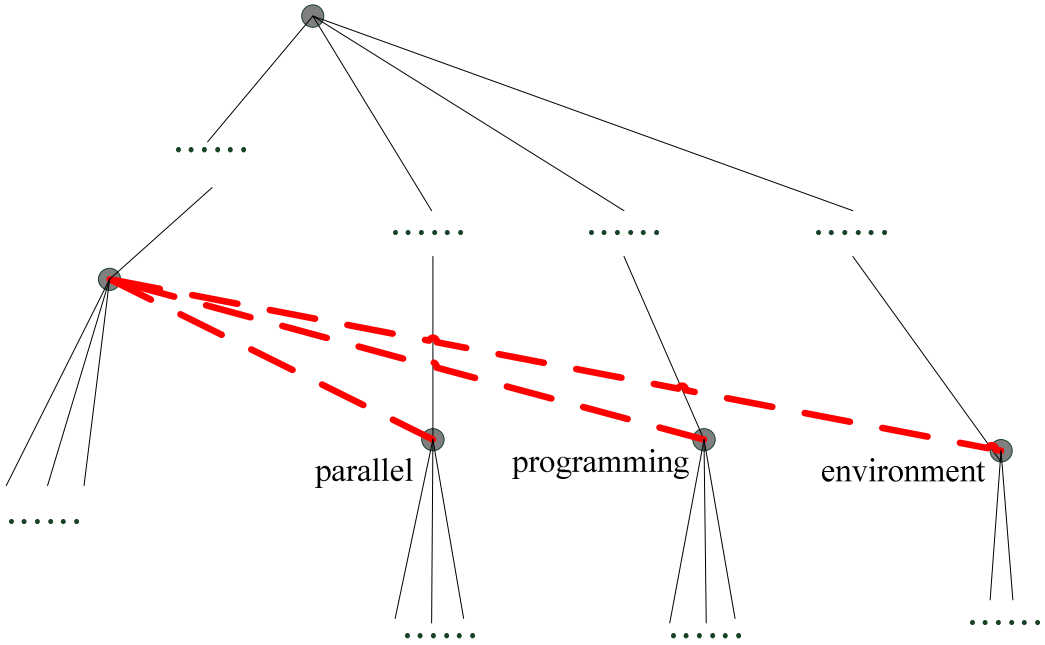

圖 5.4 無法找回的 R-clique

在圖 5.4 中，由於整顆 HD tree 的大小太過於龐大故我們只節錄一個無法被 EAFR 演算法回傳的 R-clique 的部分。高度最高的點為 HD tree 的根節點，圖中 parallel、programming、environment 三個點為同一個 R-clique 的對應節點，而連接這三個對應節點的粗體虛線的點為這三個對應節點之間最短路徑會經過的點，而由此四個節點可組成一個 R-clique。但是在轉換樹中 R-clique 的各個對應節點被分配到不同的 ELCA 樹下，導致 EAFR 演算法無法回傳此類的解。故當一個 R-clique 的對應節點被分散到不同 ELCA 樹底下時，EAFR 演算法便無法找到該 R-clique。

然而比較 BFS+ tree 與其他四種轉換樹，由表 5.4 及圖 5.2 可以看出 BFS+ tree 的召回率明顯高出許多，由於 BFS+ tree 在建立轉換樹時會將原始圖所有的邊都加入到轉換樹中，使得一個節點能夠被分配到多個不同的 ELCA 樹下改善前述的一個 R-clique 的對應節點被分配到不同 ELCA 樹下的狀況。然而 BFS+ tree 即使有加入原始資料圖所有的邊來考慮未加入的邊資訊，但是樹的形狀和原始資料圖仍然不同，加上 ELCA 搜尋先天上的限制無法解決對應節點被分配到不同 ELCA 樹下的問題，故像在 TQ3 及 TQ4 等查詢句仍會出現一部分如圖 5.5 的狀況使 BFS+ tree 無法回傳全部的解。

不過整體來看，除了 BFS+ tree 有接近百分之百的召回率外，MB tree 的作法皆有近 9 成以上的召回率，至於 HD、LD、HLD tree 則各有 1~2 個查詢句達到 7 成的召回率，表示將圖轉換成樹的作法，利用 EAFR 演算法 MB tree 與 BFS+ tree 能保持高的召回率。

5.2 查詢效率評估

接著我們以查詢句 TQ6~TQ9 來比較 EFFR 演算法在不同關鍵字出現頻率的搜尋效率，在這邊我們將 R 分別設為 7，而 keyword frequency 的總數是 TQ6 < TQ7 < TQ8 < TQ9：

查詢句	B&B	HD tree	LD tree	HLD tree	MB tree	BFS+ tree
TQ6	2.728 (26 組)	0.138 (0%)	0.160 (0%)	0.058 (0%)	0.109 (7.69%)	2.472 (65.38%)
TQ7	11.187 (3 組)	0.468 (0%)	0.480 (0%)	0.193 (0%)	0.406 (0%)	8.232 (100%)
TQ8	11.975 (119 組)	0.175 (0%)	0.204 (0%)	0.081 (0%)	0.198 (7.56%)	1.962 (88.24%)
TQ9	4.086 (17 組)	0.441 (0%)	0.500 (0%)	0.161 (0%)	0.526 (5.88%)	9.028 (100%)

表 5.5 EFFR 演算法查詢效率與召回率比較(sec)

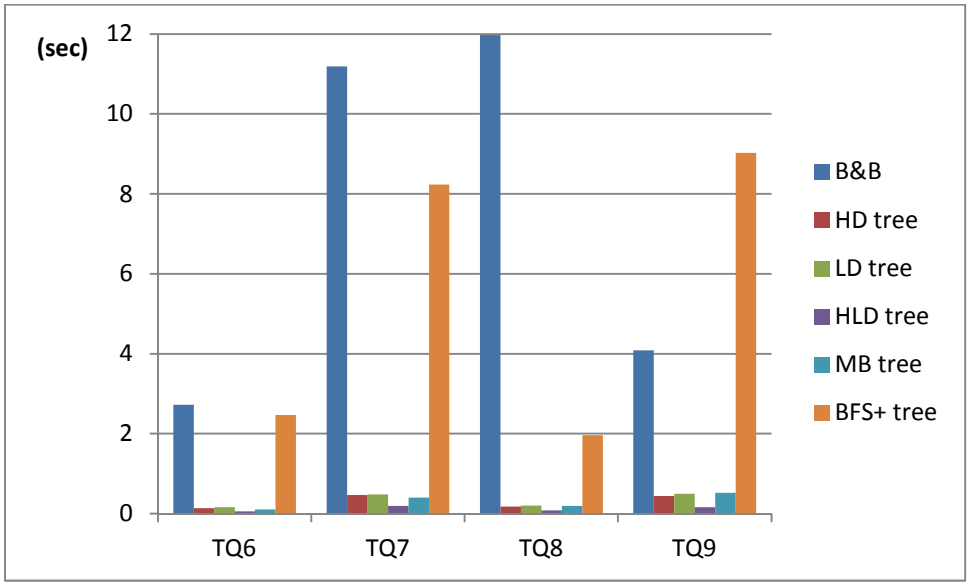

圖 5.5 EFFR 演算法查詢效率比較

參考表 5.5 與圖 5.5，由於第四章前述 Branch and Bound 演算法在搜尋時的查詢時間主要是取決於能組成 R-clique 的節點組合量，故在 TQ6~TQ8 的查詢時間主要和關鍵字的出現頻率成正比。而 TQ9 在一開始的設計會輸出太多組結果，故我們再加入一個關鍵字頻率不同新關鍵字讓輸出的 R-clique 組數減少，因此導

致 TQ9 在檢查時能組合成 R-clique 的節點組合數變少，而使查詢時間較 TQ7 及 TQ8 較低。

而由表 5.5 與圖 5.5 無法直接看出 EFFR 演算法的搜尋需要時間是否和關鍵字出現頻率大小成正比，但是可以看出對轉換樹進行 EFFR 演算法搜尋需要的時間較 Branch and Bound 少，由於 EFFR 演算法的計算量主要取決於每個 ELCA 樹中所有對應節點到 ELCA 節點的距離檢查，針對這點，我們列出每個查詢句在各個轉換樹檢查的對應節點數量總和。

TQ6	對應節點#	時間(sec)
HD tree	3119	0.138
LD tree	3042	0.16
HLD tree	2155	0.058
MB tree	1424	0.109
BFS+ tree	34258	2.472

TQ8	對應節點#	時間(sec)
HD tree	4736	0.175
LD tree	4531	0.204
HLD tree	3580	0.081
MB tree	2082	0.198
BFS+ tree	31178	1.962

TQ7	對應節點#	時間(sec)
HD tree	8806	0.468
LD tree	8380	0.48
HLD tree	6268	0.193
MB tree	6044	0.406
BFS+ tree	92280	8.232

TQ9	對應節點#	時間(sec)
HD tree	8089	0.441
LD tree	8218	0.5
HLD tree	5434	0.161
MB tree	6627	0.526
BFS+ tree	97783	9.028

表 5.6 EFFR 演算法對應節點數量分析

在表 5.6 我們列出每個查詢句在各個轉換樹搜尋時所需要檢查的對應節點數量。由表 5.6 可以看出，在大部分狀況下，當 ELCA 搜尋所需要檢查的對應節點的數量增加時，所需要的查詢時間也會較長。 而 BFSP tree 由於在建樹時必須將原始圖的所有邊都加入到樹中，導致需要檢查的對應節點量大於其他四種轉換樹，而使需要的搜尋時間較長。

接著我們繼續用查詢句 TQ6~TQ9 比較 EAFR 演算法和 Branch and Bound 演算法在不同的關鍵字出現頻率的搜尋效率，在此 R 設定為 7：

查詢句	B&B	HD tree	LD tree	HLD tree	MB tree	BFS+ tree
TQ6	2.728	0.281 (26.92%)	0.175 (69.23%)	0.478 (38.46%)	0.662 (100%)	2.866 (100%)
TQ7	11.187	0.716 (66.67%)	0.500 (100%)	1.156 (100%)	0.971 (100%)	10.537 (100%)
TQ8	11.975	1.746 (38.66%)	0.754 (18.49%)	3.429 (43.50%)	3.122 (92.44%)	13.117 (88.24%)
TQ9	4.086	0.302 (76.47%)	0.156 (35.30%)	0.507 (70.59%)	0.370 (94.11%)	3.707 (100%)

表 5.7 EAFR 演算法查詢效率比較(sec)

參考表 5.7 為 EAFR 演算法在不同關鍵字出現頻率的查詢時間，前面提到在 EAFR 演算法中，當產生較多組 ELCA 樹將對應節點分成多個小群時，由於每群的節點數量會降低而能夠讓 BABR 演算法的總計算量降低，在此我們列出各個查詢句在不同轉換樹查詢時所產生的 ELCA 樹數量來比較查詢效率：

TQ6	ELCA 樹#	時間(sec)
HD tree	122	0.281
LD tree	125	0.175
HLD tree	58	0.478
MB tree	80	0.622
BFS+ tree	646	2.866

TQ8	ELCA 樹#	時間(sec)
HD tree	72	1.746
LD tree	70	0.745
HLD tree	35	3.429
MB tree	43	3.122
BFS+ tree	280	13.117

TQ7	ELCA 樹#	時間(sec)
HD tree	159	0.716
LD tree	148	0.5
HLD tree	74	1.156
MB tree	135	0.971
BFS+ tree	852	10.537

TQ9	ELCA 樹#	時間(sec)
HD tree	136	0.302
LD tree	136	0.156
HLD tree	62	0.507
MB tree	257	0.370
BFS+ tree	816	3.707

表 5.8 EAFR 演算法對應節點數量分析

由表 5.8 可以看出，除了 BFS+ tree 以外當 ELCA 樹的數量較高時，所需的查詢時間通常也會較低，而 BFS+ tree 雖然在所有查詢句皆可以產生最多個 ELCA 樹，但是因為 BFS+ tree 樹的大小取決於原始圖的邊樹，而這點使 BFS+ tree 樹的大小必大於其他四種轉換樹，導致在搜尋時間上無法有較佳的表現。

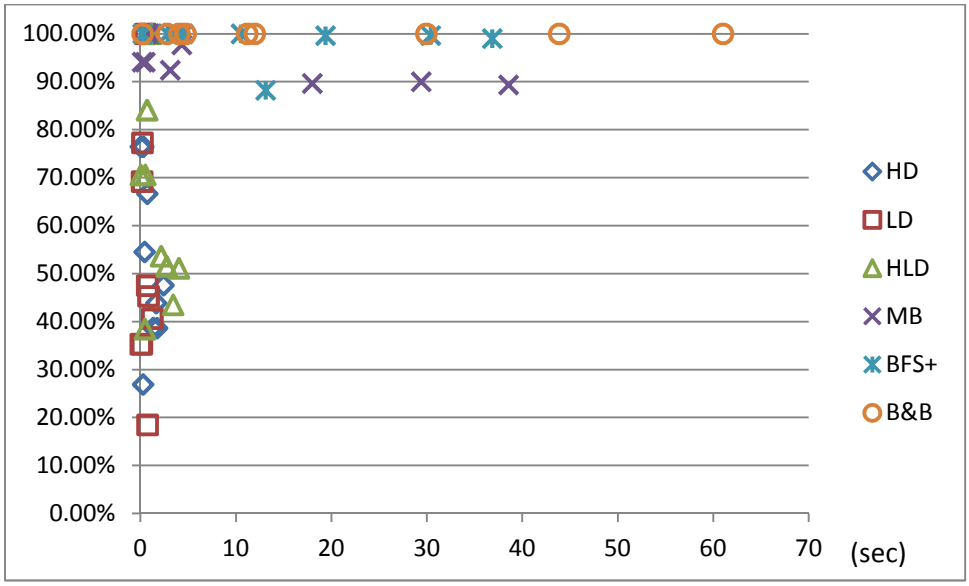

圖 5.6 EAFR 演算法查詢效率與召回率比較

我們由表 5.4 及表 5.7 的資訊 (TQ1-TQ9) 畫成圖 5.6 的散布圖分析，圖中 X 軸為查詢花費時間，Y 軸為查詢的召回率，而在這些資料中召回率高且查詢時間低的結果為較好的方法，也就是圖中越接近左上角的點代表越好的方法。由圖 5.6 發現，MB tree 的點幾乎都集中在左上角，代表 MB tree 所需的查詢時間較低且同時有相當高的召回率；B&B 和 BFS+的分布則部分落於右上角，表示有時需要較多的時間；其他三種（HD、LD、HLD）則多落於左下角，也就是雖然快速但召回率低。故可以看出在 EAFR 演算法中使用 MB tree 查詢可以有最好的表現。

由前述的實驗可知，在 EAFR 演算法中 MB tree 與 BFS+ tree 能保持極高的召回率，而在效率上使用 MB tree 查詢所需的時間也較短。而在 EFFR 演算法中雖然大部分的轉換樹所需的查詢時間較短，但在召回率部分似乎只有 MB tree 與 BFS+ tree 能較有效的回傳解。在此我們仿照圖 5.6 選擇 MB tree 以及 BFS+ tree 對 EFFR 演算法及 EAFR 演算法同時比較查詢效率與召回率，使用的查詢句一樣為 TQ1~TQ9。

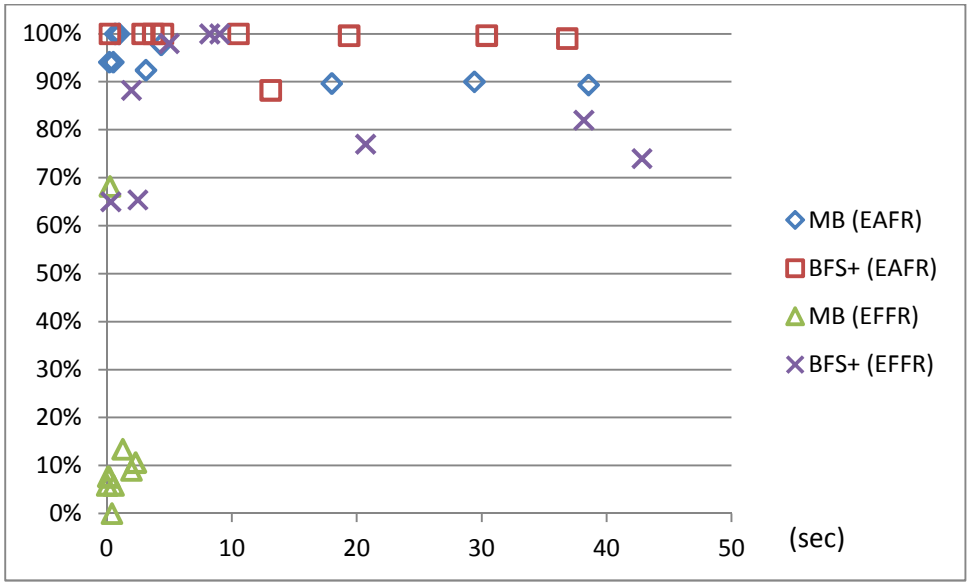

圖 5.7 EFFR 演算法與 EAFR 演算法查詢效率與召回率比較

由圖 5.7 可以看出，在 EFFR 演算法只有在 BFS+ tree 搜尋時才能得到較高的召回率，而 EAFR 演算法中以 MB tree 的點在圖中多次出現在圖的左上角，代表用 MB tree 搭配 EAFR 演算法能同時兼顧較高的召回率與效率。雖然搭配 EFFR 演算法的 BFS+ tree 也有不少點集中在圖的左上角，然而比較 EFFR 演算法的 BFS+ tree 與搭配 EAFR 演算法的 MB tree 發現，在整體的表現上還是搭配 EAFR 演算法的 MB tree 較佳。

5.3 不同 R 值大小對查詢效率比較

我們再以不同的 R 值大小來觀察 R 值變動對 EAFR 演算法查詢召回率的影響，這邊我們選用 TQ2 作為查詢句：

R	HD	LD	HLD	MB	BFS+
R=6	56.52%	90.62%	90.62%	100.00%	100%
R=7	54.55%	77.27%	84.09%	97.73%	100%
R=8	47.12%	53.84%	61.54%	87.50%	100%

表 5.9 EAFR 不同 R 值召回率比較

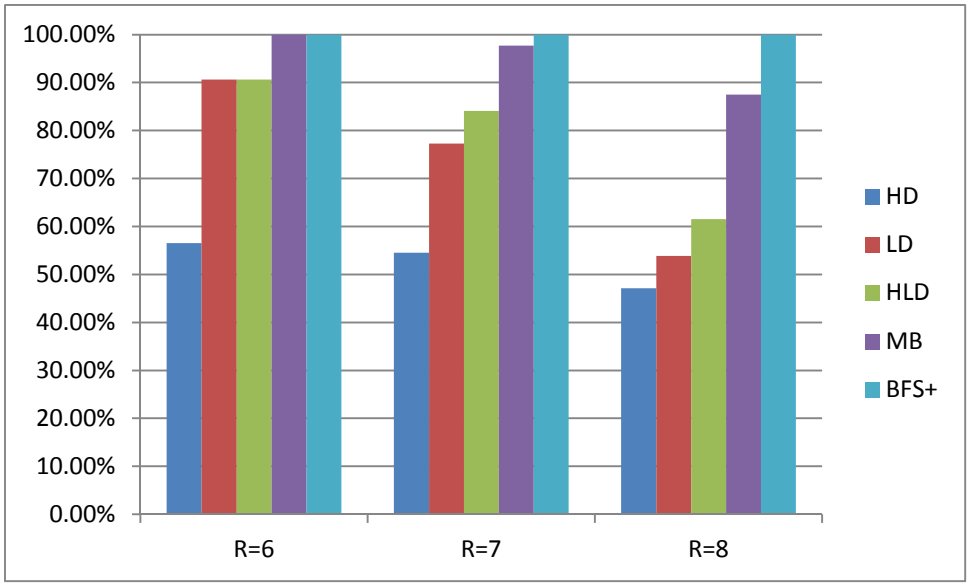

圖 5.8 EAFR 不同 R 值召回率比較

由表 5.9 及圖 5.8 可以發現，當 R 值上升時，各個轉換樹的召回率大部分皆會下降，由於 BFS+ tree 以外的四種轉換樹無法考慮到原始資料圖的所有邊資訊，故在 R 值變大時可能會加重一個 R-clique 的對應節點被分配到不同 ELCA 樹的狀況，而 BFS+ tree 由於在樹中包含了所有原始圖的邊，針對 R 值的變動對召回率似乎沒有大太的影響。

我們在此比較 R 在不同的大小時對 EFFR 搜尋效率造成的影響，使用的查詢句為查詢句 TQ8：

R	B&B	HD tree	LD tree	HLD tree	MB tree	BFS+ tree
R=6	10.053	0.175	0.082	0.078	0.196	2.002
R=7	11.975	0.175	0.204	0.081	0.198	1.962
R=8	16.972	0.187	0.212	0.085	0.194	2.075

表 5.10 EFFR 演算法不同 R 值的效率比較(sec)

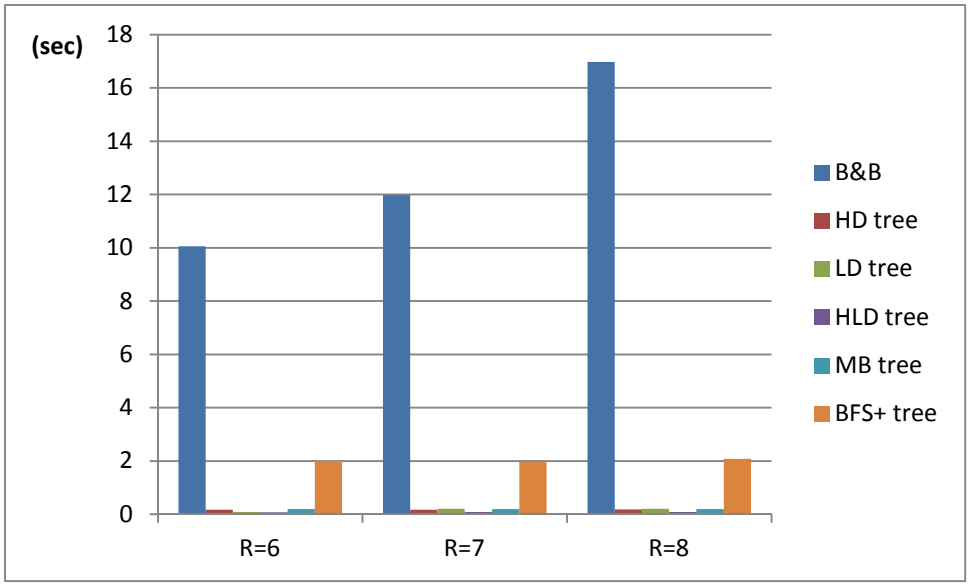

圖 5.9 EFFR 演算法不同 R 值的效率比較

R=6	對應節點#	時間(sec)
HD tree	4736	0.175
LD tree	4531	0.082
HLD tree	3580	0.078
MB tree	2082	0.196
BFS+ tree	31178	2.002

R=7	對應節點#	時間(sec)
HD tree	4736	0.175
LD tree	4531	0.204
HLD tree	3580	0.081
MB tree	2082	0.198
BFS+ tree	31178	1.962

R=8	對應節點#	時間(sec)
HD tree	4736	0.187
LD tree	4531	0.212
HLD tree	3580	0.085
MB tree	2082	0.195
BFS+ tree	31178	2.075

表 5.11　EFFR 演算法在不同 R 值下的查詢比較

由於 R 值大小不會影響一個轉換樹所找到的 ELCA 樹，由表 5.10 可以看出，R 值大小對 ELCA 樹下需要檢查的對應節點數量沒有太大的影響，故 EFFR 演算法在搜尋時間上並不會隨著 R 值的變動有太大的變化。在表 5.11 我們列出不同轉換樹需要檢查的對應節點數量，發現執行時間主要和對應節點數量成正比。

接著我們比較 R 在不同的大小時對 EAFR 演算法搜尋效率造成的影響，使用的查詢句為查詢句 TQ8：

R	B&B	HD tree	LD tree	HLD tree	MB tree	BFS+ tree
R=6	10.053	1.645	3.155	4.305	4.871	11.706
R=7	11.975	1.746	0.745	3.429	7.624	13.117
R=8	16.972	2.122	0.826	4.275	7.523	17.784

表 5.12　EAFR 演算法不同 R 值的效率比較(sec)

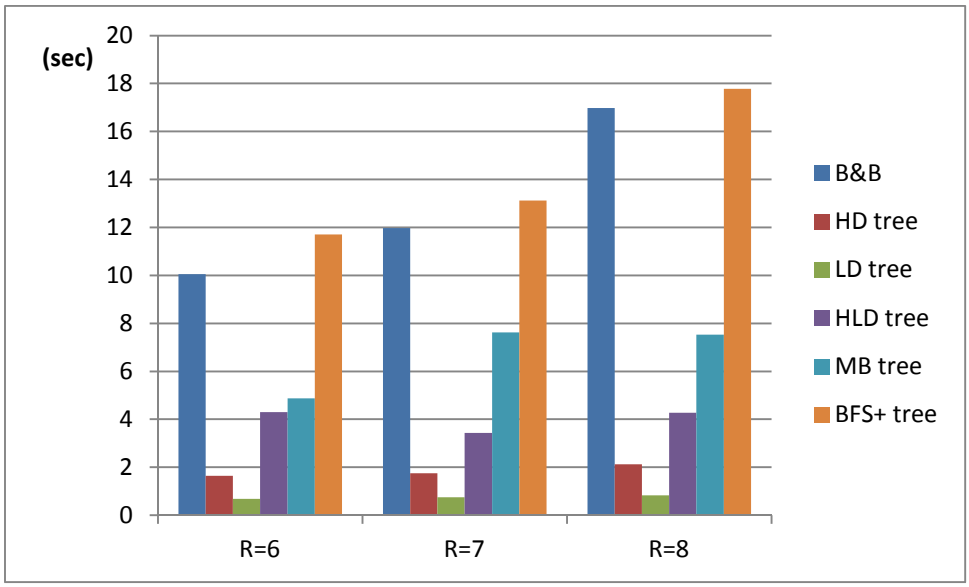

圖 5.10　EAFR 演算法不同 R 值的效率比較

參考表 5.12 及圖 5.10，發現隨著 R 的大小增大，EAFR 演算法所需的時間在大部分的轉換樹也會上升，這是因為 EAFR 演算法所呼叫的 BABR 演算法會隨著 R 變大而使需要計算的叢集組合變多。另外 EAFR 演算法的計算量和 ELCA 樹將 BABR 演算法所需要計算的節點分群的效果有關，而 R 的大小對同一個查詢句並不會影響 ELCA 樹的個數，故我們在此只列出 R=6 時各個轉換樹產生的 ELCA 樹個數來比較時間：

R=6	ELCA 樹#	時間(sec)
HD tree	72	1.645
LD tree	70	0.686
HLD tree	35	4.305
MB tree	43	4.871
BFS+ tree	280	11.706

表 5.13 EAFR 演算法的時間與 ELCA 樹個數的比較(sec)

參考表 5.13，可以發現當 ELCA 樹的數量較多時，通常執行時間會較短，比較 HD tree 與 HLD tree，由於 HD tree 的 ELCA 樹的數量較多，使得全部的對應節點被分成了多群比較小群的叢集，進而降低 BABR 演算法的總計算量。另外參考 BFS+ tree，雖然 BFS+ tree 產生的 ELCA 樹數量最多，但是由於 BFS+ tree 的樹大小較另外四種轉換樹高，故在搜尋時間上需要花最多時間。

第六章 結論及未來方向

在本論文中，我們提出了數種將 R-clique 的查詢方式轉化成由資料圖建出轉換樹的策略，並提出 EFFR 與 EAFR 兩種以 ELCA 的搜尋方式快速搜尋出 R-clique 的演算法。由於樹狀查詢的複雜程度比圖型查詢低，在 EFFR 演算法的部分，雖然大部份轉換樹在查詢的召回率表現較差，但是在 BFS+ tree 進行 EFFR 演算法搜尋時，在大部分的查詢句都能有較好的搜尋效率，且召回率大約都能維持到 6 成到 7 成左右。在 EAFR 演算法的部分，HD tree、LD tree 及 HLD tree 會依照查詢句的不同而在召回率上有不同的表現，但在召回率高的時候可以提升到 8 成左右。而 MB tree 與 BFS+ tree 在召回率表現皆接近或高於 9 成，尤其是 BFS+ tree 幾乎都能達到將近百分之百，但是 BFS+ tree 由於樹的大小較大，導致在效率上有時不及 Branch and Bound 演算法。而在 MB tree 的部分，效率皆比 Branch and Bound 演算法好，而在召回率上的表現也相當高。

本論文未來的研究方向，希望能改善目前的蒐尋演算法，加強目前的蒐尋效率以及蒐尋的召回率，以及在轉換樹方面，希望能提出更好的轉換樹方法來改善查詢效率及召回率。並期望未來能改良到支援 top-k 的搜尋並與其他 top-k 的圖型演算法進行比較。

參考文獻

[CCW13] An-Chiang Chu, Kun-Mao Chao, Bang Ye Wu, “A linear-time algorithm for finding an edge-partition with max-min ratio at most two.” Discrete Applied Mathematics, 161(7-8): 932-943, 2013.

[CW04] Kun-Mao Chao, Bang Ye Wu, “Spanning trees and optimization problems”, hapman & Hall/CRC, 2004.

[DLQW+07] Bolin Ding, Xuemin Lin, Lu Qin, Shan Wang, Jeffrey Xu Yu, Xiao Zhang, “Finding Top-k Min-Cost Connected Trees in Databases”, In Proceedings of the ICDE conference, 2007.

[KA11] Mehdi Kargar, Aijun An, “Keyword Search in Graphs: Finding r-cliques”, In Proceedings of the VLDB Conference, 2011.

[LOF+08] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, Lizhu Zhou, "EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-structured and Structured Data", In Proceedings of the SIGMOD conference, 2008.

[LR98] Hsueh-I Lu, R. Ravi, “Approximating Maximum Leaf Spanning Trees in Almost Linear Time”, J. Algorithms 29(1): 132-141, 1998.

[PX07] Yannis Papakonstantinou, Yu Xu, “Efficient LCA based Keyword Search in XML Data”, In Proceedings of the EDBT conference, 2008.

[QYCT09] L. Qin, J. Yu, L. Chang, Y. Tao, “Querying communities in relational databases”, In Proceedings of the ICDE conference, 2009.

[TJM+08] Partha Pratim Talukdar, Marie Jacob, Muhammad Salman Mehmood, Koby Crammer, Zachary G. Ives, Fernando Pereira, Sudipto Guha,

"Learning to Create Data Integrating Queries", In Proceeding of the VLDB conference, 2008.

[TPSS11] Yufei Tao, Stavros Papadopoulos, Cheng Sheng, Kostas Stefanidis, “Nearest Keyword Search in XML Documents”, In Proceedings of the SIGMOD Conference, 2011.

[XP05] Yu Xu, Yannis Papakonstantinou, “Efficient Keyword Search for Smallest LCAs in XML Databases”, In Proceedings of the SIGMOD Conference, 2005.

[YCHH12] Junjie Yao, Bin Cui, Liansheng Hua, Yuxin Huang, “Keyword Query Reformulation on Structured Data”, In Proceeding of the ICDE, 2012.

[ZBWL+12] Junfeng Zhou, Zhifeng Bao, Wei Wang, Tok Wang Ling, Ziyang Chen, Xudong Lin, Jingfeng Guo, “Fast SLCA and ELCA Computation for XML Keyword Queries based on Set Intersection”, In Proceedings of the ICDE conference, 2012.