平行可變的間隙限制最長共同子序列與增長區間詢問

SHIANG-YUN YANG and PANGFENG LIU, National Taiwan University
JAN-JAN WU, Institute of Information Science, Academia Sinica

可變的間隙限制最長同子序列應用於基因、分子生物學中，在之前的研究已提出理論在 $O(nm)$ 的算法，而其中使用特殊的增長樹集合併均攤 $O(1)$ 的調整操作，在易於實作的版本中，一般並查集使用均攤 $O(\alpha(n))$ 完成調整操作。然而，原先的序列算法以直觀的波前平行方法無法達到有效的空間運作。

這篇論文修改了原本算法的資料結構，稀疏表可在均攤 $O(1)$ 時間內完成所有調整操作，並搭配平行建表的笛卡爾樹編碼，運行時間為 $O(nm/p + n \log n)$。我們提出的平行建表，搭配的動態卡塔蘭數索引編碼算法，可解決遞增任意區間最大值詢於均攤 $O(1)$ 時間。

關鍵字: 區間極值詢問、增長區間極值詢問、增長後綴極值詢問、最長共同子序列、笛卡爾樹、平行算法、多核心平台

ACM Reference format:
Shiang-Yun Yang, Pangfeng Liu, and Jan-Jan Wu. 2017. 平行可變的間隙限制最長共同子序列與增長區間詢問. 1, 1, Article 1 (July 2017), 27 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 介紹

最長共同子序列 (Longest Common Subsequence，簡稱 LCS) [10] 經常使用在字串處理中的經典問題。例如說，我們使用 `diff` 工具來呈現兩段文字不同的差異，找到它們最長的共同子序列；在版本控制系統中，SVN 和 Git 也經常使用 LCS 來呈現變動情況；在分子生物學中，生物序列對齊問題 [1, 12] 的其中一種 DNA、RNA 和蛋白質相似度也經常使用 LCS 表示之。

Iliopoulos 和 Rahman [15] 介紹在 LCS 問題下各種不同的約束變形。例如固定間距的 LCS (fixed gap LCS，簡稱 FGLCS)，要求挑選位置之間最多間隔 $k+1$ 個字元。對於輸入

DOI: 10.1145/nnnnnnn.nnnnnnn

長度分別為 n 和 m 的字串，我們已知 FGLCS 可以在 $O(nm)$ 內被解決 [15]。在另一個問題可變間距 LCS (variable gap LCS，簡稱 VGLCS) ，對於兩個相鄰的挑選位置，至多間隔後者提供的約束距離加一。因此，可以視 FGLCS 為 VGLCS 的特例，對於每一個位置提供的約束距離皆為 k。

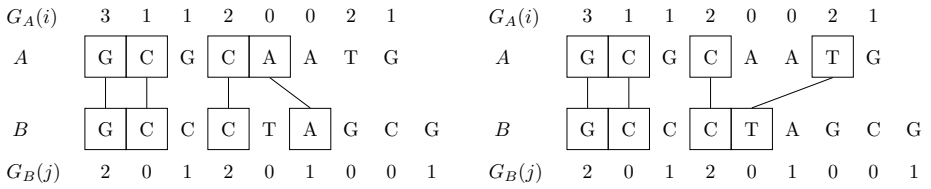


Fig. 1. 有限間距最長共同子序列的例子

這裡我們以一個例子說明 VGLCS 的間隔函數。假設字串 A 為 GCGCAATG，其 A 相應的間隔約束為 $(3,1,1,2,0,0,2,1)$，字串 B 為 GCCCTAGCG，其 B 相應的間隔約束為 $(2,0,3,2,0,1,2,0,1)$，參照圖 1 的說明，找到其中一組 GCCT 為此 VGLCS 的一組解，且滿足每一個的挑選位置與前繼者的位置之間最多為其位置約束值加一。

這篇論文著重於解決 VGLCS 的「有效率的平行算法」。在先前的研究中，Peng [14] 的研究提供易於實作的 $O(nm\alpha(n))$ 的版本以及漸近 $O(nm)$ 的算法，其中 α 為阿克曼函數的反函數 [2]。接續，論文將提出我們的 $O(nm)$ 算法，其具有易於實作的特性，同時在平行環境下能運行相當有效率。

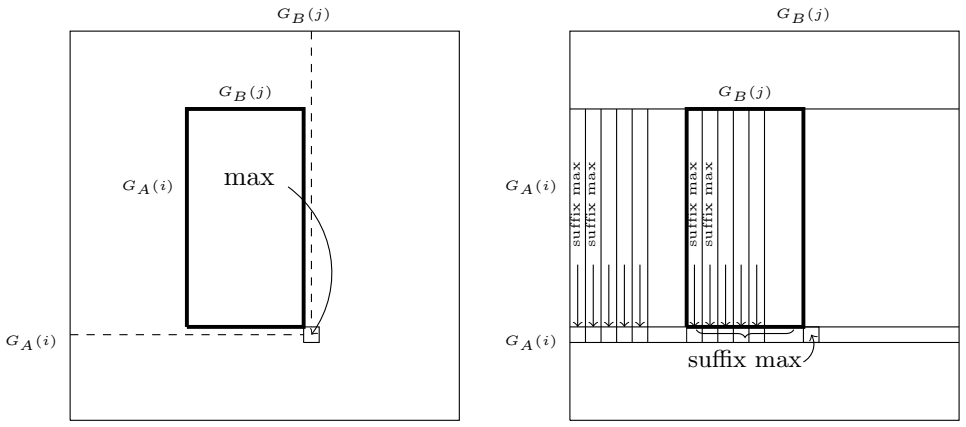
在多核心平台上，大部分 LCS 的平行方法都在「波前平行」(wavefront parallelism) 上，波前方法直接使用 LCS 的遞迴公式推導而成，故平行度與效能皆受限於遞迴公式，因此，有些研究致力於改善平行度與效能。如 Yang [18] 介紹新的轉換方法來拓展出更好快取效能。藉此類的概念，我們使用類似的方式，將 Peng 序列算法中的並查集 (disjoint set) 替換成更有力的「稀疏表」(sparse table) 以提高更好的快取效能。

這篇論文的架構如下述：在章節 2，我們提出先前的 VGLCS 平行算法。接著，在章節 3 和 4，提出我們的易實作的平行版本，其時間複雜度 $O(nm)$ 比先前的來得更好。章節 5 和 6 介紹相應的優化技巧與實驗結果。在最後章節 8 和 9 總結先前提出的技術與後續可能發展的方向。

2 平行 VGLCS 算法

2.1 基礎動態規劃法

首先，我們使用動態規劃法解決 VGLCS [14] 問題。令 A 和 B 為兩個輸入長度分別為 n 和 m 的字串，相應的 G_A 和 G_B 陣列為間隔約束，定義 $V[i][j]$ 為子字串 $A[1,i]$ 和 $B[1,j]$ 的最長 VGLCS 的長度，從上述的定義中，我們可以推出 $V[i][j]$ 來自於 $V[k][l]$，其中 k 介於 $i-G_A(i)-1$ 到 $i-1$、l 介於 $j-G_B(j)-1$ 到 $j-1$，意即陣列 V 往左上延伸的一個矩形內最大值，參照圖 2(a) 的說明。



(a) 如何計算 V (b) 使用增長後綴最大值詢問計算 V

Fig. 2. VGLCS 的動態規劃方法

計算動態規劃中的 V 的優化方法如下所述。當計算相同的行 i 上的 $V[i][j]$ 時，所有的矩形區域皆包含相同的約束 $G_A(i)$，故將計算步驟拆分成兩個階段，第一階段計算每一列的後綴最大值放入陣列 R 中，接著在 R 中找到後綴長度為 G_B 的最大值恰好為 $V[i][j]$ 的結果，請參照圖 2(b) 的說明。

上述的優化需要「增長」後綴最大值詢問，如在第一階段中，每一個列的最大值，當我們完成第 i 行上的結果轉移到下一行 $V[i+1][*]$ 時，將會詢問每一列上後綴長度 $G_A(i+1)$ 的元素最大值，故每一列將會各自維護資料結構。同樣地，在同一行的計算中，從 $V[i][j]$ 移動到 $V[i][j+1]$ 時，將從陣列 R 中提取後綴最大值，我們將這一系列的操作，稱呼為「增長後綴最大值詢問」(incremental suffix maximum query)，意即我們維護一個資料結構，允許插入一個新的元素至尾端，同時詢問任意位置到尾端的最大值。

2.2 Peng 算法

Peng [14] 提供 VGLCS 的循序算法 1，其使用先前提到的優化方式完成。在最外層迴圈中，依序完成每一行，而最裡層迴圈依序從左往右。我們假設資料結構 C 可以回答任意列的後綴最大值，我們定義 $C[j]$ 可以回答在 V 的第 j 列的後綴最大值。從先前的觀察中，在同一行 i 上有相同的約束長度 $G_A(i)$，故我們將從 C 中詢問從第 $i-1$ 行往前 $G_A(i)+1$ 個元素的後綴最大值，接著，我們將從 R 中提取後綴長度為 $G_B(j)+1$ 的最大值放入 $V[i][j]$，請參照圖 2(b)。

更新 V, C, R 的方法如下述。當字串 A 的第 i 個字元匹配字串 B 的第 j 個字元，我們將設 $V[i][j]$ 為左上矩形內部最大元素值加一，如圖 2(b) 所示。這一矩形內最大值可以從 R 中，透過後綴 $G_B(j)$ 個元素找到其值。而 R 處理從前一行的結果，並且包含 $j-1$ 的元素，

我們會在這 R 上詢問後綴長度為 $G_B(j)+1$ 的最大值結果，並成為 $V[i][j]$ 同時加入到 $C[j]$ 中。相反地，如果字串位置沒有匹配，直接將 $V[i][j]$ 設為 0，並更新相應的 R 和 $C[j]$。

ALGORITHM 1: Peng's algorithm for finding VGLCS [14]

Input: A, B: the input string; G_A, G_B: the array of variable gapped constraints;
Output: Find the LCS with variable gap constraints.
Create an array of m data structures $C[m]$ that support ISMQ;
Create an empty table $V[n][m]$;
for $i \leftarrow 1$ to n do
    Create a data structure R that supports ISMQ;
    for $j \leftarrow 1$ to m do
        if $A[i] = B[j]$ then
            $t \leftarrow$ Query R for the maximum among the last $G_B(j)+1$ elements ;
            $V[i][j] \leftarrow t+1$;
        else
            $V[i][j] \leftarrow 0$;
        end
        $t \leftarrow$ Query $C[j]$ for the maximum among the last $G_A(i)+1$ elements ;
        Append t to R ;
        Append $V[i][j]$ into $C[j]$;
    end
end
Retrieve the VGLCS by tracing $V[n][m]$;

2.3 增長後綴最大值詢問

從之前所討論的 Peng 算法，發現解決 VGLCS 問題需著重於「增長後綴最大值詢問」，為了解決增長後綴最大值詢問，資料結構需要要三種類型的操作：第一類操作 Make，建立空陣列 A；第二類操作 Append(V)，附加一個元素 V 至陣列 A 尾端：第三類操作 Query(x)，找到位置 x 至陣列 A 尾端的最大值。

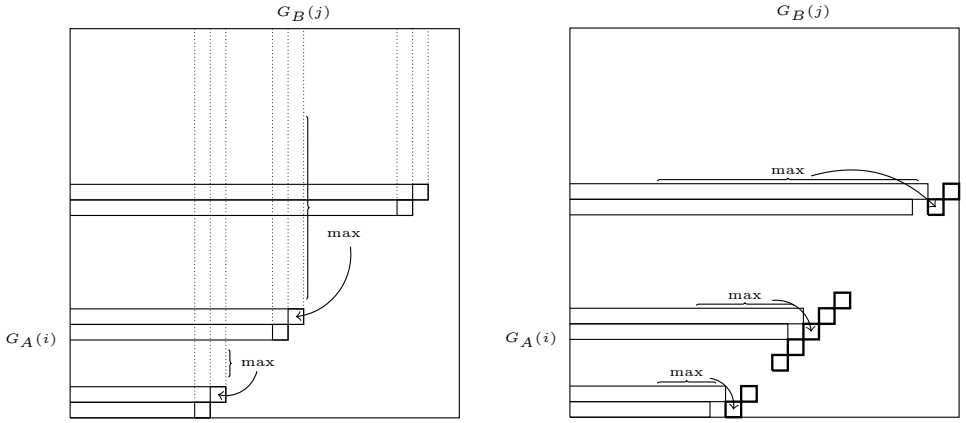
Peng 算法中使用並查集解決所有的後綴最大值詢問，而並查集最早由 Gabow [8] 和 Tarjan [16] 提出解決聯集與查找問題 (union-and-find problem)。在這裡我們視並查集中的集合由左往右排成一列，每一集合的根節點表示該集合的最大值，這些最大值排成一列呈現遞減順序。當我們加入一個元素 x 到陣列尾端，其 x 視為一個集合，將會往左合併比它小的集合。很輕易地明白，Query(x) 只需要查找 x 所在的集合，透過並查集的 find 找到最大值。任何合併和查找操作，我們得知攤銷複雜度為 $O(\alpha(n))$。

2.4 使用稀疏表平行 VGLCS 算法

Peng [14] 所提到的循序 VGLCS 算法 1 和其他 LCS 變形問題都難以使用一行接一行的平行方法，使用動態規劃的算法都依賴狀態的轉移，這使得建構方法有大量的資料資賴

性 (heavy data dependency)，這是難以使用一行接一行的平行方法的問題所在，因為每一個元素需要在同一行的元素結果。

Peng 算法需要使用波前平行方法，這使得需要「額外的」的空間維護每一行列的狀態，這些反而在循序算法中視可以省略的的部分，在平行處理再次被需要而增加實作的硬體需求。回到 VGLCS 需要找到 V 矩行內部的最大值，我們每一次平行處理同一個對角線的資訊，此時必須額外抄寫不同列的結果，因為每一列的約束限制都各有不同，參照圖 3 粗體說明表示平行處理的元素部分。每一個操作都會保留相應的資料結構，額外的記憶體保存必然會帶來一些處理花費。



(a) 第一階段 (b) 第二階段

Fig. 3. 波前平行方法額外抄寫的紀錄資料

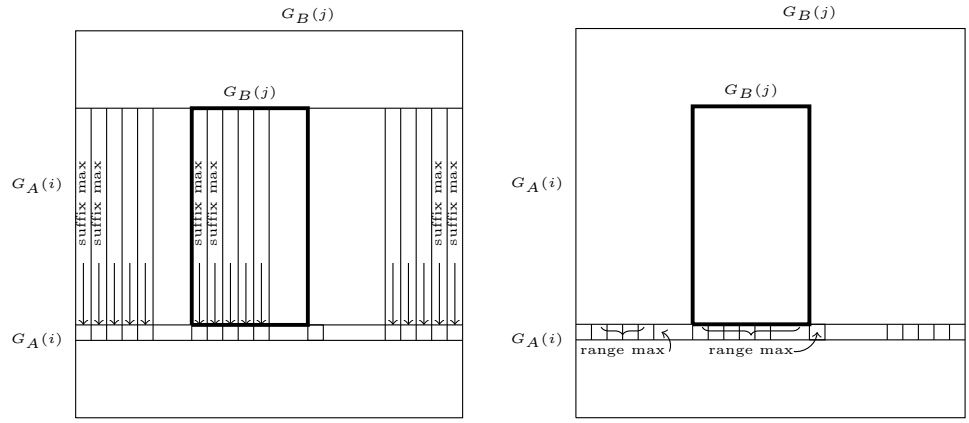
我們提出的列接列方法 (row-by-row approach) 將只需要一個資料結構維護行上的資訊，這個資料結構收集每一列的後綴最大值，並且移除掉先前提及的繁重的資料依賴性，不再依賴同一行上的處理資訊。

相較於波前平行方法，我們使用的列接列方法使用較少的空間、較好的工作負載平衡、較低的執行緒同步開銷。同時，在波前平行方法的關鍵路徑 (critical path) 長度為列接列方法的數倍。此外，我們移除掉同一行上的資料依賴性，將能「完全」和「均勻」地平行。其結果造成較平衡的工作負載分佈和較容易的執行緒同步。

我們提出的算法框架如下所述：每一次計算 V 的同一列，計算同一列時分兩個階段，第一階段從資料結構 C 中平行詢問每一列的後綴長 $G_A(i)+1$ 最大值，並且儲存到陣列 R 中，接著繼續使用算法 1，將 C 持續維護 V 上的增長後綴最大值詢問。請參考圖 4 的說明。

第二階段中，我們算法著重於「區間最大值詢問」(range maximum queries)，陣列 R 上紀錄 m 個列的後綴最大值，需要平行處理在第 i 行上的 m 的元素值。相較於循序算法，我們並不使用陣列 R 的後綴詢問，取而代之，我們需要詢問 R 上的區間詢問，每一個詢問

區間長度為每個列位置的間隔約束，請參照圖 4 的說明。當我們計算出 $V[i][j]$ 時，仍然需要將其加入 C 的第 j 列進行增長操作，以應付第 $i+1$ 行的計算結果。然而，這 C 只需要支持後綴詢問，並非區間詢問。相對地，我們對 R 維護區間最大值詢問操作，這些操作皆可以平行地去處理。



(a) 第一階段 (b) 第二階段

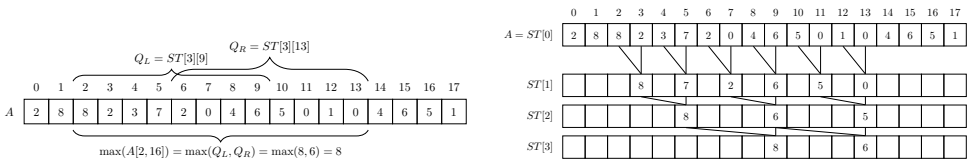
Fig. 4. 計算 V 的同一行，將分成兩個階段

為了解決資料依賴性，我們需要好的資料結構去解決「平行環境」的增長後綴/區間最大值詢問。首先，我們發現並查集這一類的資料結構不易於平行，原因在於以下三點：第一點，每一次詢問將會改變資料結構，因為通常都伴隨著「壓縮路徑」(compress path) 的操作，同時壓縮路徑造成難以在執行緒之間得到一致的結構；第二點，由於執行緒各自壓縮不同路徑，由於壓縮路徑長短不同，導致工作負載不平行；第三點，當有許多執行緒同時運行，多次的同步操作將無法有效率地運行。

2.4.1 稀疏表 由於先前提到並查集在平行環境下的效能問題，我們使用「稀疏表」(sparse table [3]) 來支援 VGLCS 算法中的增長後綴/區間詢問。稀疏表最原始的版本需要前處理，其花費 $O(n \log n)$ 時間，隨後對於任意一維的區間詢問只需要 $O(1)$ 時間。稀疏表的儲存架構採用二維陣列，對於第 j 行上的第 i 列元素，為原輸入陣列的第 i 的元素往前 $2^j - 1$ 個元素的最大值。

我們以圖 5 簡單說明稀疏表如何運作。對於原輸入陣列 A，接著根據先前提到的算法建造稀疏表 T。當給定一個陣列 A 上的區間詢問時，最多拆成 2 個稀疏表上的查找。例如，詢問區間從位置 2 到 13，我們將拆成詢問 2 到 9 ($T[3][9]$) 以及 6 到 13 ($T[3][13]$)，這兩者都是從第三層的稀疏表中提取連續 $2^3 = 8$ 個元素的最大值。

平行建造稀疏表是相當容易的，請參照算法 2 的說明，算法 2 建造稀疏表時間為 $O(n \log n/p + \log n)$，其中 n 為元素個數，而 p 為處理器個數，這個算法相當容易實作。



(a) 原輸入陣列 (b) 稀疏表

Fig. 5. 稀疏表例子

ALGORITHM 2: A Parallel Sparse Table Building Algorithm

Input: $A[0..N-1]$: the input array
Output: T: a sparse table for the input array A
Create a two-dimensional array $T[\log N][N]$;
Copy A to $T[0]$;
for $j \leftarrow 0$ to $\log N$ do
    for $i \leftarrow 2^j$ to N do in parallel
        $T[j][i] \leftarrow \max(T[j-1][i-2^j], T[j-1][i])$;
    end
end

2.4.2 基於稀疏表之平行 VGLCS 相較於並查集，稀疏表上的任何操作都較容易平行。因此，在 Peng 的循序算法中，內層迴圈需要交替地附加和詢問操作於 R 上，請參考算法 1 提及的細節，這些交替操作存在嚴重的資料依賴性。此外，平行度受到並查集的路徑長度影響，而這些路徑長通常很短，故約束平行度的發展。

若我們使用稀疏表於平行 VGLCS 算法中，請參考算法 3 的說明。算法中仍一次計算 V 的每一行，計算單一行時分成兩個階段，第一階段平行詢問 C 上的後綴長 $G_A + 1$，並將結果收集到陣列 R 中；接著根據陣列 R 平行建造稀疏表 T。在第二階段使用 T 平行詢問所有區間最大值。

實作兩階段的部分時，有著各自不同的挑戰。第一階中由於列彼此之間獨立詢問，故平行操作較為容易，然而，它仍需插入元素到 C 的尾端，同時要有效率地找到後綴最大值放入陣列 R。第二階段「不存在」強制的插入操作，故可以靜態地處理之，因我們平行計算 V 同一行上，需要的是區間詢問，而非後綴詢問於稀疏表 T 上。在接續的兩個章節中，我們將介紹如何克服這些挑戰。章節 3 優先解決在第二階段的問題，接著我們在章節 4 著重於第一階段的挑戰。

ALGORITHM 3: Parallel Algorithm for Finding VGLCS

Input: A, B: the input string; G_A, G_B: the array of variable gapped constraints;
Output: Find the LCS with variable gapped constraints
Create an array of m data structure $C[m]$ to support ISMQ ;
Create an empty table $V[n][m]$;
for $i \leftarrow 1$ to n do
    for $j \leftarrow 1$ to m do in parallel // The first stage
        $M[j] \leftarrow$ suffix maximum of length $G_A(i)+1$ from $C[j]$;
    end
    Build a sparse table T with data of M in parallel with Algorithm 2;
    for $j \leftarrow 1$ to m do in parallel // The second stage
        if $A[i] = B[j]$ then
            $t \leftarrow$ the range maximum among the $G_B(j)+1$ elements before $M[j]$ by querying T ;
            $V[i][j] \leftarrow t+1$;
            Append $V[i][j]$ to $C[j]$;
        end
    end
end
Retrieve the VGLCS by tracing $V[n][m]$;

3 區間最大值

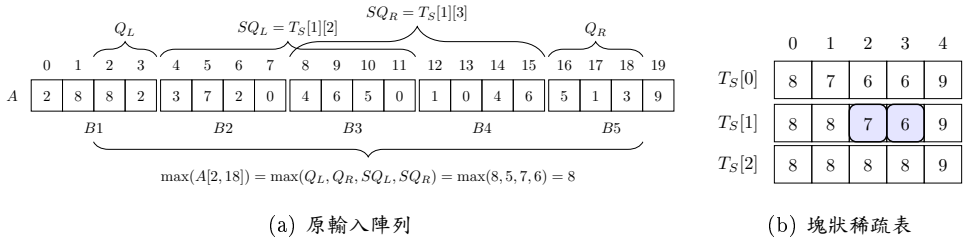
在這一章節中，我們將著重於算法 3 第二階段的挑戰，意即高效的「區間最大值詢問」，增長區間最大值詢問遠比增長後綴最大值詢問來得困難，原因在於後綴為區間的一種特例。

增長區間最大值詢問類似於後綴最大值詢問，分成三個操作：第一種 Make，建立空陣列 A；第二種 Append(V) 操作，附加元素 V 於陣列 A 的尾端；第三種 Query(L, R)，找到陣列 A 位置 L 到 R 的元素最大值。

3.1 塊狀稀疏表

為了改善平行 VGLCS 算法，我們採用「塊狀稀疏表」(blocked sparse table)，其最早被 Fischer [6] 提及。塊狀方法先將原輸入陣列拆成好幾個塊，每一塊有 s 個元素，將每一塊的最大值建立成稀疏表 T_s。相較於章節 2.4.1 提到的稀疏表，未分塊的稀疏表單一元素為一塊，並且需建造 $\log n$ 層的區間最大值。

塊狀方法解決區間最大值的方法如下所述：首先，我們分成兩種詢問「超塊詢問」(super block query) 以及「塊內詢問」(in-block query)，其中超塊詢問解決「連續完整」的塊狀最大值，而塊內詢問解決單一塊內的區間詢問。如先前章節 2.4.1 提及的稀疏表算法，超塊詢問最多拆成 2 個稀疏表上的查找。塊內詢問則需要「一次」查找，這個查找需在「最小共同祖先表」(least common ancestor table) 上運作，關於這個表將在後續的章節中



(a) 原輸入陣列 (b) 塊狀稀疏表

Fig. 6. 塊狀稀疏表例子

闡述細節。因此，任何區間詢問將會被拆成至多 2 次 T_s 上的查找以及 2 次的塊內詢問，整體而言最多有 4 次記憶體存取。由於超塊詢問與先前的稀疏表相當，接著將探討塊內詢問如何被解決。

Fischer 算法 [6] 掃描整個輸入陣列所產生的塊，接著將其轉換成「笛卡爾樹」(Cartersian tree)。笛卡爾樹的每個節點儲存兩個鍵值，分別為資料的索引值與值。我們可以將笛卡爾樹視為「堆」(heap)，按照資料值呈現最大/最小堆的特性；若只看資料的索引值，笛卡爾樹具有「二元搜尋樹」(binary search tree) 的特性，請參照圖 7 的說明。圖 7 的每一個樹節點的水平位置反應出其原始位置的相對關係，而垂直位置反應數值大小關係。換而言之，若需要得知塊內詢問的第 i 位置到 j 位置上的最大值，找到相應的笛卡爾樹的「最小共同祖先」(least common ancestor)。例如詢問區間位置 4 到位置 6 之間的最大值，從表格中得知「最大值位置」為 5，最後得到最大值為 7。

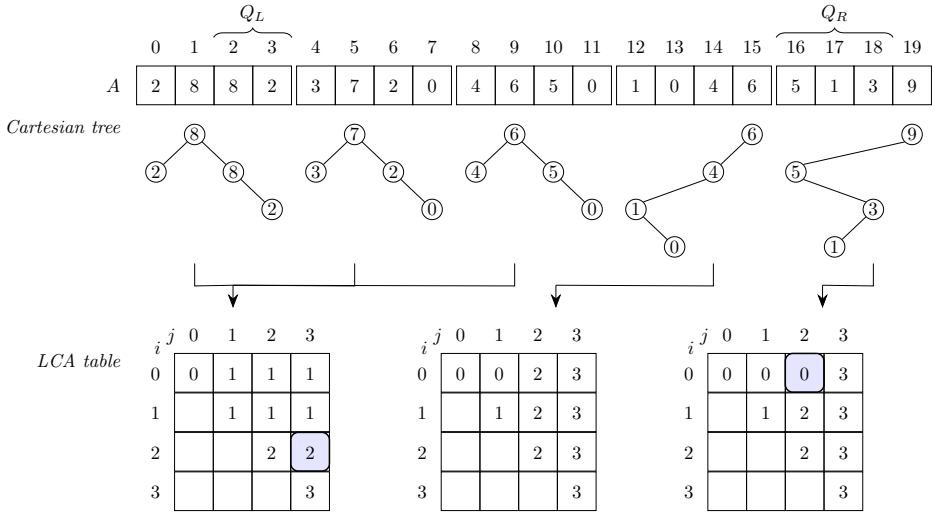


Fig. 7. 最小共同祖先表

為了解決塊內詢問，Fischer 算法需要計算每一塊相應的最小共同祖先表 (LCA table，簡稱「LCA 表」)，在掃描每一塊的資訊後，建造相應的 LCA 表，此一表儲存資訊為

$((i,j),k)$，對於任意位置 i 到 j 的最大值位置 k，請參照圖 7 的說明。我們可以從 LCA 表中提取位置 i 到位置 j 的最大值位置 k，隨後映射回原輸入陣列得到區間 (i,j) 的最大值。

特別注意到算法中，沒有儲存第 k 個元素的值，而是儲存其「索引值」，意即保留元素值的相對關係，因此兩個塊之間可以共用同一個 LCA 表。如圖 7 所示，前三個區塊共用同一個 LCA 表，因此，無論在哪塊中詢問區間 $(1,3)$ 最大值，任何塊內詢問皆回答 2。

Fischer 算法的核心在於如何計算每一塊相應的 LCA 表，間接地應答所有塊內詢問。我們也可以發現將會有 C_s 種不同形的笛卡爾樹，其中 C_s 為節點個數為 s 的二元搜尋樹的不同形狀個數。每一塊將對應一個笛卡爾樹，接著我們將使用「卡塔蘭索引值」(Catalan index) 表示塊與笛卡爾樹的相應關係，這相應關係如圖 7 所描述。

Fischer 算法 [6] 建立 LCA 表前，基於效能考量而選用塊大小 $s = \frac{\log n}{4}$，因卡塔蘭數 $C_s = \frac{1}{s+1}\binom{2s}{s} = O(\frac{4^s}{s^{3/2}})$，故掃描和建立笛卡爾樹的時間皆為 $O(n)$。意即 Fischer 算法在前處理階段需要 $O(n)$ 時間，對於任意區間詢問需 $O(1)$ 時間，容易地明白前處理和詢問操作皆已經理論最佳的結果。

然而，Fischer 算法在大量資料下易造成「快取未中」(cache miss)，而算法中採用「請求」(on-demand) 式的建立所需要的 LCA 表，唯有在掃瞄過程中找到尚未建立的 LCA 表時，才進行新的 LCA 表建造。一旦使用 LCA 表，這意味著使用上隨著帶入 LCA 表至快取時，易將另一個 LCA 表逐出快取，反覆使用的次數一多，就容易造成嚴重的快取未中。

為減少快取未中的次數，Demaine 算法 [4] 提出在笛卡爾數上的「快取適性」(cache-aware) 操作，Demaine 算法並不需要使用記憶體檢查是否需要建立 LCA 表，而是建立所有的可能笛卡爾樹的表。因此，Demaine 算法使用以長度為 $2s$ 的二元字串表示笛卡爾樹，這個二元字串上可回答塊內區間最大值。然而，檢測過程中將會需要計算兩個相鄰 1 的位元之間有多少個 0，這一操作難以在當代計算機架構上高效地運行。

3.2 右側棧出編碼

我們提出另一種塊的編碼方法－名為「右側棧出」(rightmost-pops)，有別於 Demaine [4] 提出的二元字串，其為了改善查詢區間極值查找的效能。然而，右側棧出主要仍來自 Demaine 和笛卡爾樹的想法。

右側棧出編碼主要維護笛卡爾樹的「右側路徑」(rightmost path) 於棧 (stack)，同時，當加入新的元素到笛卡爾樹時，保留在棧上的推出 (pop) 次數，請參照圖 8 的說明。為了維護加入第 i 的元素 a_i 滿足笛卡爾樹的堆性質，將會把樹上右側較小的元素翻轉到 a_i 的左子樹，也就是將棧上小於等於 a_i 的元素推出，其推出的次數令為 t_i，在過程中滿足 $0 \le t_i < s$，其中 s 為塊大小。接著，這些 t_i 將會是編碼笛卡爾樹的主要素材。

請參照圖 8，當我們插入元素 $a_1 = 0$ 時，將不會有任何元素被推出棧，接著插入元素 $a_2 = 4$ 時，將會把 a_0 和 a_1 推出棧，此時得到 t_2 為 2，從先前的定義中，我們可以明確地將棧表示為笛卡爾樹的最右側結構，接著將這些 t_i 組合來表示笛卡爾樹的狀態。

「右側棧出編碼」主要藉由數個 t_i 「隱式辨識」(implicitly identify) 出個別的笛卡爾樹。對於任意的塊內詢問，只需要在數個 t_i 上測試即可。假設回答塊狀詢問區間 l 到 r 於

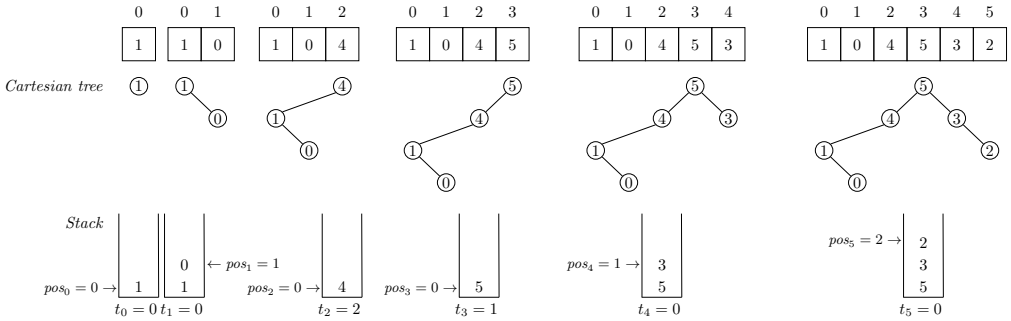


Fig. 8. 表示右側路徑於棧

這些 t_i 上的最大值，我們維護棧上的推出次數 x，初始值令 x 為 0，接著依序檢測 t_l 到 t_r，令迴圈索引 j 依序從 l 到 r，每經過一次迴圈，將 x 扣除 $t_j - 1$，我們只需要紀錄最後一次發生 x 小於等於 0 的索引值 j 為何，則「最後的」索引值 j 即是需要的「最大值位置」，請參照算法 4 的說明。算法 4 中提供的每一步操作都可以明確地在當代電腦中找到相應的指令，別於當初 Demaine 提出的算法，這大幅度地容易以簡單指令表示之。

ALGORITHM 4: Range Minimum Query in 64-bits Cartesian Tree

Input: tmask: 64-bits Cartesian tree; $[l, r]$: query range
Output: minIdx: the index of the minimum value in interval
minIdx $\leftarrow l$, $x \leftarrow 0$;
for $l \leftarrow l + 1$ to r do
    $x \leftarrow x + 1 - ((\text{tmask} \gg (l \ll 2)) \,\&\, 15)$;
    if $x \leq 0$ then
        minIdx $\leftarrow l$, $x \leftarrow 0$;
    end
end
return minIdx ;

算法 4 的正確性請參照理論 3.1 的證明。直觀的方法來自於找到樹根，其表示區間內最後插入到根的元素。

Theorem 3.1. *算法 4 正確回答塊內區間最大值詢問*

Proof. 當推出次數 x 小於等於 0 時，其意涵著該位置為區間內的樹根，則最後一次成為樹根的位置則為區間內的最大值。因為笛卡爾樹具有堆性質，根為樹內最大值。 □

選用塊大小 $s = 16$，則所有 t_i 小於 16 且每一個 t_i 可以表示成 4-bit 整數，串接這十六個 4-bit 整數可以壓縮到 64-bit 整數來表示一個完整的笛卡爾樹，請參照算法 5，其算法只需要 $O(s)$ 時間。算法 5 中的左移、右移和加減法都可以很明確地對應到機器指令。

ALGORITHM 5: Encode a data block of sixteen data with rightmost-pops encoding into a 64-bits integer.

Input: $A[1..16]$: input data block; s: the fixed block size as $\frac{\log n}{4} = 16$
Output: t: a 64 bit rightmost-pops encoding of A
Create an array D of size $s+1$;
$p \leftarrow 0$, $D[0] \leftarrow \infty$;
$t \leftarrow 0$;
for $i \leftarrow 1$ to s do
    $v \leftarrow A[i]$, $c \leftarrow 0$;
    while $D[p] < v$ do
        $p \leftarrow p-1$, $c \leftarrow c+1$;
    end
    $p \leftarrow p+1$;
    $D[p] \leftarrow v$;
    $t \leftarrow t \mid (c \ll ((i-1) \ll 2))$;
end
return t ;

考量到效能問題，選用塊大小 $s = \frac{\log n}{4} = 16$ 的原因來自於大部分的機器皆為 64-bit 暫存器，且有許多非常高效率的暫存器操作。此外，這一類的方法並沒有使用到 LCA 表，這意涵著我們只需要維護 64-bit 的 t_i 表示笛卡爾樹的狀態，充分地改善記憶體使用和快取效能，從理論上已經可以支持到 $n = 2^{64}$ 大小的問題。

我們提出的「右側棧出編碼」改善了快取問題，卻增加了詢問操作的複雜度。相較於 Fischer 算法，算法 4 提供較高的資料局部性和較有規律性的管理，前處理同樣需要 $O(n)$ 時間，單一詢問操作需要 $O(s)$ 的時間，其中 s 為塊大小 $\frac{\log n}{4}$。在實作中，我們選用 s 為 16 來支持 $n = 2^{64}$ 的問題，所以 s 可以視為一個很小的常數，空間複雜度仍為 $O(n)$。帶入平行 VGLCS 算法中使用，整體時間複雜度為 $O(n^2 \log n/p + n \log n)$，其中的 $\log n$ 來自於塊大小的選用 $s = O(\log n)$ 影響。

4 增長區間最大值詢問

在這一章節中，我們將著重於算法 3 第一階段的挑戰，這裡我們將計算 V 每一列的後綴最大值，同時提供泛用性更高的「增長區間最大值詢問」，增長區間最大值詢問將逐漸地加入新的元素到陣列尾端，並且隨問任意區間的最大值，

4.1 建立最小共同祖先表

在章節 3 中，我們曾討論如何使用最小共同祖先來回答區間最大值詢問，這裡我們著重兩個議題如下：第一個問題—「如何給予一個二元搜尋樹相應的『卡塔蘭索引值』」，以及第二個問題—「如何找到任意兩個樹上節點的共同祖先」

4.1.1 **標記笛卡爾樹** 我們標記笛卡爾樹時，採用「字典順序」(lexicographical order)，從 0 編號到第 n 個卡塔蘭數減一。二元索引樹的字典順序下述遞迴定義：當一個樹 x 出現在樹 y 前需滿足下述的其一條件，請參照圖 9 說明節點大小為 1、2、3 的標記情況。

- x 的左子樹節點個數比 y 的左子數節點個數多
- x 和 y 的左子樹節點個數相同，且 x 的左子樹字典順序小於 y 的左子樹字典順序。
- x 和 y 的左子樹節點個數相同且 x 和 y 的左子樹字典順序相同，且 x 的右子樹字典順序小於 y 的右子樹字典順序。

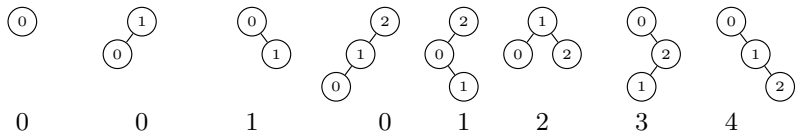


Fig. 9. 二元搜尋樹的標記情況，由左而右分別是節點個數為 1、2 和 3

4.1.2 **最小共同祖先** 隨著加入的資料，我們需要有效率地決定「最小共同祖先」以回答區間最大值。令 t 為二元搜尋樹的卡塔蘭索引值，則 t 介於 0 到 $C_s - 1$ 之間，其中 s 為二元搜尋樹的節點個數。定義 $\mathcal{A}(s,t,p,q)$ 為在節點個數為 s 時，卡塔蘭索引值為 t 的二元索引樹上的節點 p 和 q 的最小共同祖先。例如，從圖 9 中，得知 $\mathcal{A}(3,2,0,2) = 1$。令 s_l 為左子樹節點個數、s_r 為右子樹節點個數，而 t_l, t_r 分別為左、右子樹的的卡塔蘭索引值。藉由上述的定義，得到最小共同祖先 $\mathcal{A}$ 遞迴如公式 1，其相應的平行算法 6。

$$\mathcal{A}(n,t,p,q) = \begin{cases} \mathcal{A}(s_r,t_r,p-s_l-1,q-s_l-1)+s_l+1 & s_l \le p \le q < n \\ \mathcal{A}(s_l,t_l,p,q) & p \le q < s_l \\ s_l & 0 \le p \le s_l, s_l \le q \le i \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

分析算法 6 的空間複雜度，從表 $\mathcal{A}$ 維護所有節點大小 1 到 s，當子樹節點個數為 m 時，不同的二元搜尋樹個數為第 m 個卡塔蘭數 C_m，其 C_m 可表示成 $\frac{1}{m+1}\binom{2m}{m} = O(\frac{4^m}{m^{3/2}})$，對於每一個節點個數 m 的二元樹，我們儲存所有節點之間的最小共同祖先，故需要 $O(m^2)$。整體而言，空間複雜度為 $O(s\,\frac{1}{s+1}\binom{2s}{s}\,s^2)$，其中 s 為塊大小，倘若挑選 s 為 $\frac{\log n}{4}$，其空間複雜度為 $O(\sqrt{n}\,\log^{3/2} n)$。然而，對於任意區間詢問只會在節點個數為 s 上運行，我們仍需要個數小於 s 的表來建立表 $\mathcal{A}$。

分析算法 6 的時間複雜度，在算法 6 的循序版本，其時間複雜度為 $O(\frac{s^3}{s+1}\binom{2s}{s})$，因公式 1 相應的結果皆為 $O(1)$。當選用 s 為 $\frac{\log n}{4}$ 時，時間複雜度為 $O(\sqrt{n}\,\log^{3/2} n)$。

對於算法 6 的平行版本，我們觀察計算不同個 C_m 的樹是「彼此獨立」的，故可以平行化處理之。然而，當節點個數為 m 時，找到左右子樹的卡塔蘭索引值的複雜度為 $O(m)$ (算法中的行 4 所示)，同時行 4 和行 5 為同一層，故沒有平行的必要性。最後平行化最外層的

ALGORITHM 6: A parallel algorithm that computes the least common ancestor table $\mathcal{A}$

```
Input: s: the maximum tree size
for n ← 1 to s do
    for t ← 0 to C_n − 1 do in parallel
        for p ← 0 to n − 1 do in parallel
            Compute s_l, t_l, s_r, and t_r ;
            for q ← p to n − 1 do
                Compute A[n][t][p][q] according to Equation 1 ;
            end
        end
    end
end
```

兩個迴圈，得到平行算法 6 的時間複雜度為 $O(\frac{s^3}{s+1}\binom{2s}{s}/p + s^2) = O(\sqrt{n}\,(\log^{3/2} n)/p + \log^2 n)$，其 p 為處理器個數。

4.2 計算卡塔蘭索引值

算法 6 中需要卡塔蘭索引值 t，所以我們需要從塊中的資料中高效地決定 t 為何，這裡有兩種方法來計算—「建立樹」或者「維護右側路徑」

4.2.1 建立樹 根據塊中數個元素決定卡塔蘭索引值，我們可以建立笛卡爾樹後，再決定其索引值為何，意即我們確切地把樹建造後，利用左右子樹的索引值和節點個數組合出卡塔蘭索引值，這將會要求遞迴遍歷整棵樹於 $O(n)$ 時間，其 n 為節點總數。令 $\mathcal{T}$ 為卡塔蘭索引值，根據公式 2 的定義，從左右子樹的節點個數 s_l, s_r、左右子樹的卡塔蘭索引值 t_l, t_r，以及第 i 個卡塔蘭數 C_i 得到。

$$\mathcal{T}(s_l, t_l, s_r, t_r) = t_l\, C_{s_r} + t_r + \sum_{i=0}^{s_l-1} C_i\, C_{s_l+s_r-i} \tag{2}$$

更近一步優化等式 2 的計算，我們預處理卡塔蘭數乘積的「前綴和」，並儲存這些結果於記憶體中，將可以直接地使用它們，而不用再次地在等式 2 計算之。

4.2.2 維護右側路徑 先前計算卡塔蘭索引值的方法依賴完整的子樹資訊，這是較沒效率的做法。我們提出使用「棧」維護笛卡爾的「右側路徑」，不需要維護整棵樹的資訊，這一種方法類似章節 3.2 提及的「右側棧出編碼」。在我們知道卡塔蘭索引值 t 後，我們可解決算法 6 和公式 2 中使用的 LCA 和索引值的需求。

以棧維護右側路徑，將可以高效地計算卡塔蘭索引值 t，我們不再需要建立完整的樹才能得到結果，相應的代價是維護左子樹的節點個數與其卡塔蘭數索引值於棧 D 中，棧 D 的每一個元素皆包含其左子樹的卡塔蘭索引值和左子樹的節點個數，請參照算法 7 的說

明。在算法 7 中，棧 D 每一個元素帶有元素值 v、左子樹的節點個數 s、左子樹的卡塔蘭索引值 t，並且我們使用 p 表示指向棧頂的指針。

ALGORITHM 7: Catalan index computation for a data block

Input: $A[1..n]$: input data block; n: the number of elements;
Output: t: The Catalan index of the input data block
Create a stack D of $n+1$ elements. Every element has s, t, and v ;
$p \leftarrow 0$, $D[0] \leftarrow \langle 0, 0, \infty \rangle$;
for $i \leftarrow 1$ to n do
    $v \leftarrow A[i]$;
    $s \leftarrow 0$, $t \leftarrow 0$;
    while $D[p].v < v$ do
        $t \leftarrow \mathcal{T}(D[p].s, D[p].t, s, t)$;
        $s \leftarrow s + D[p].s + 1$;
        $p \leftarrow p - 1$;
    end
    $p \leftarrow p + 1$;
    $D[p] \leftarrow \langle s, t, v \rangle$;
end
$s \leftarrow 0$, $t \leftarrow 0$;
while $p > 0$ do
    $t \leftarrow \mathcal{T}(D[p].s, D[p].t, s, t)$;
    $s \leftarrow s + D[p].s + 1$;
    $p \leftarrow p - 1$;
end
return t ;

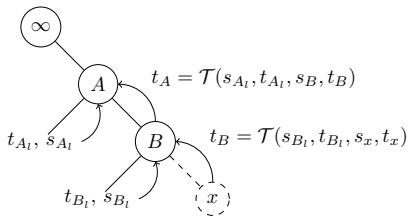


Fig. 10. 計算樹的卡塔蘭索引值，令 A_l、B_l 分別為 A、B 的左子樹

算法 7 根據塊內元素和棧 D 計算出卡塔蘭索引值。在第一個雙層迴圈中，外層迴圈依序加入元素到右側路徑的尾端，其中棧頂為 $D[p]$，每次將從 D 中推出「較小」的元素，當我們推出時相當於翻轉笛卡爾樹的子樹，需要計算新的索引值 t 以及節點個數 s，並且抽換掉右側路徑的狀態。最終，整個樹的卡塔蘭索引值 t 將根據公式 2，統合棧 D 上的左

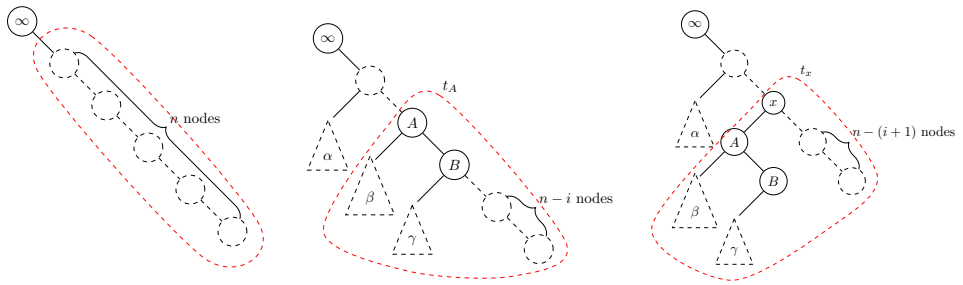
右子樹的索引值和節點個數計算得到，請參照算法 7 和圖 10 的說明。圖中 A_l、B_l 分別為 A、B 的左子樹，在運行完所有插入元素的推出操作後，我們需要從棧 D 中推出所有元素得到整棵樹的卡塔蘭索引值，也就是算法 7 最後一個迴圈所運行的內容。

算法 7 可以計算任何笛卡爾樹的卡塔蘭索引值，由於每一個元素「最多」進出棧一次，故只需 $O(s)$ 時間完成。

4.3 動態計算卡塔蘭索引值

先前的研究中，已經有數種編碼笛卡爾樹的方法。如 Fischer [6] 介紹了第一種方法，以及 Masud [9] 在其論文中也描述如何運用編碼方法降低運行指令次數。不幸地，這些算法的建造方案都是離線操作，意即它們需要給予完整塊才能計算出索引值，同時，這造成它們無法適應持續地附加新的資料。此外，它們要求前處理時間為 $O(n)$，其 n 為元素個數。這些前處理需要額外的空間，或者需要從硬碟中提取資料。

概括說明我們提出的笛卡爾樹標記方法，我們加入新的資料時，使用「正規化」的方式，將樹的節點個數統一為 n，意即對於任何的笛卡爾樹將會填充節點個數直到 n。當節點個數為 i 時，我們將附加 $n-i$ 個「右傾節點」(right-child-only) 於樹上。因此，當樹為空，我們添加 n 個右傾節點，請參照圖 11(a) 的說明。當插入第 i 節點時，添加 $n-i$ 個右傾節點於右側路徑上，如圖 11(b) 所示。為了說明這一加入的路徑，我們將其命名為「虛擬路徑」(virtual path)。



(a) 初始化 $t_{root_0} = C_n - 1$ (b) t_{root_i} (c) $t_{root_{i+1}} = t_{root_i} + t_x - t_A$

Fig. 11. 正規化笛卡爾樹的過程，使用增加虛擬路徑來增加節點個數

樹的正規化有兩項優點。其一，將可以減化卡塔蘭索引值的計算，意即插入新的元素時，更新卡塔蘭索引值較為容易。其二，正規化操作補齊最大元素到塊的尾端，對於現存的元素不受任何影響，可以運行相應的區間詢問，同時使得編碼算法可以「動態」維護相應的 LCA 表，便可支持增長區間詢問。

動態卡塔蘭索引值的計算將使用樹的正規化操作，其使用方法如下：令當前笛卡爾樹的卡塔蘭索引值為 t^*，笛卡爾樹的右側路徑在棧 D 上，隨著插入新的元素 x 到樹上時，結合算法 6 和公式 2，並且加入正規化操作得到在線版本的算法 8，在任意時刻 i，插入的第 i 個散速，將可以得到當前的卡塔蘭索引值 t^*，接著拿 t^* 解決區間最大值詢問。

為方便表示，使用 t 和 s 表示卡塔蘭索引值和笛卡爾樹節點個數，這裡的笛卡爾樹並「不包含」虛擬路徑。這些變數命名類似算法 7 使用的，額外使用 t' 和 s' 表示「包含」虛擬路徑的卡塔蘭索引值和笛卡爾樹節點個數。請參照圖 11，紅色虛線為包含虛擬路徑的部分。

在算法 8 初使化 s 和 t 的方法，接著如算法 7 運行模式相當。然而，在算法 8 中，額外初使化 s' 為 $n-i$、t' 為 $C_{s'}-1$，以應對「包含」虛擬路徑長 $n-i$ 的運行需求。

算法 8 在插入第 i 個元素 x 時，分成兩個階段。在算法中的行 4 和 5 皆推出小於 x 的元素出栈，並更新 t, s 為最後一個子樹「不包含」虛擬路徑的值，同理，在算法中的行 6 和 7 更新 t', s' 為「包含」虛擬路徑長 $n-s'$ 的結果。請參照算法 8 和圖 11 的說明。

ALGORITHM 8: Online Type of Cartesian Tree

Input: x: the added data ;
t^*: The Catalan index of the tree before adding x ;
i: the index of x ;
D: A stack where an element has s and t ;
Output: t^*: The Catalan index of the tree after adding x ;
D: A updated stack where an element has s and t

1 $s \leftarrow 0$, $t \leftarrow 0$;
2 $s' \leftarrow n-i+1$, $t' \leftarrow C_{s'}-1$;
3 while $D[p].v < x$ do
4 $\quad t \leftarrow \mathcal{T}(D[p].s, D[p].t, s, t)$;
5 $\quad s \leftarrow s + D[p].s + 1$;
6 $\quad t' \leftarrow \mathcal{T}(D[p].s, D[p].t, s', t')$;
7 $\quad s' \leftarrow s' + D[p].s + 1$;
8 $\quad p \leftarrow p-1$;
9 end
10 $p \leftarrow p+1$;
11 $D[p] \leftarrow \langle s, t, x\rangle$;
12 $t^* \leftarrow t^* + t' - \mathcal{T}(s, t, s-i, C_{s-i}-1)$;
13 return t^* ;

接著，更新棧 D 且調整「整體的」卡塔蘭索引值 t^*，此時棧 D 頂部必為元素 x 的資訊，其包含左子樹的節點個數 s 和左子樹的卡塔蘭索引值 t，請參照圖 11(c) 的說明。整體的卡塔蘭索引值 t^* 將會在更新棧後，請參照圖 11(b) 和圖 11(c) 的說明。

觀察圖 11(b) 和圖 11(c) 的紅色虛線框出的差異，我們只需要找到紅色虛線部分的卡塔蘭索引值的「差值」，使用「補丁」的方式，加上差值即可修正 t^*。所幸地，我們提供的字典順序採用左子樹先於右子樹，故圖 11(b) 和圖 11(c) 的紅色虛線區域的差值為 $t' - \mathcal{T}(s, t, s-i, C_{s-i}-1)$。最後，新的 t^* 將會增加 $t' - \mathcal{T}(s, t, s-i, C_{s-i}-1)$。

我們提出的算法 8 並不會增加原先的時間複雜度，儘管提供動態加入新的元素到樹上，複雜度仍同於先前提到的靜態索引值計算的算法 7，其時間複雜度為 $O(n)$，因每個元素恰好被推出棧一次。相似地，運行 n 次算法 8，又因在迴圈內的任何計算都屬於 $O(1)$ 操作，攤銷複雜度仍為 $O(n)$。最後，我們得知在塊內詢問與卡塔蘭索引值計算皆為攤銷 $O(1)$ 的操作。

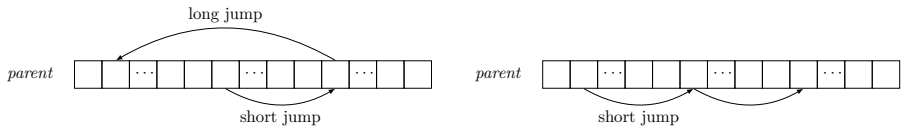
5 實作與優化

這一章節將描述如何優化 VGLCS 算法，不論是循序或者平行環境，我們將討論實作上的優化策略。特別注意到，有些技術將根據硬體特性進行優化，如快取行為等，這些並不會影響到漸近分析。

5.1 並查集實作策略

在 VGLCS 算法中，循序版本使用的並查集，將在這一章節探討如何在實作上優化。

5.1.1 快取效能 並查集的運行效能易受到快取行為影響。在 Patwary，Blair 和 Manne [13] 等人的研究中，探討不同的實作策略導致不同程度的快取未中問題。在實作中，快取未中將會發生在如何從子節點找到父節點的部分，這些伴隨著路徑壓縮 (path compression) 中使用指針找到父節點。然而，一個複雜度較低的算法，通常會有很多的「遠跳」(long jumps)，而這些遠跳將會存取距離較遠的記憶體位址而導致快取未中，請參照圖 12 說明。



(a) 低複雜度的算法 (b) 高複雜度的算法

Fig. 12. 並查集所需的父節點跳躍

「雷姆算法」(Rem's algorithm [5]) 使用「索引值合併」(merge-by-index) 策略以得到較好的快取效能。傳統的並查集使用「秩合併」(merge-by-rank) 或者「權重合併」(merge-by-size [16])，這些方法分別使用根的秩和節點個數決定合併方向。儘管它們有漸近最好的理論時間複雜度，但它們在實作上並不是這麼出色，原因在於先前的快取未中問題。相比之下，雷姆算法在合併操作中，將索引值「低」的集合指向索引值「高」的集合，將提供更容易預測的快取算法。

在我們的實驗中，我們仍使用秩合併為主要操作，然而當秩相同時，我們偏向索引值合併，打破等價情況並改善快取效能。從實驗中，我們得到「索引值合併破壞等價」(merge-by-index tie-breaking) 提供 3 % 的效能改善相較於隨機合併，請參照圖 13 說明。

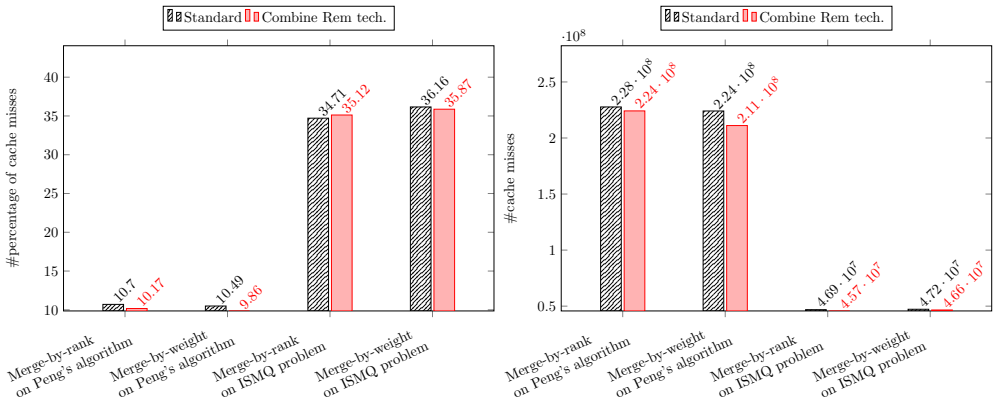


Fig. 13. 「索引值合併破壞等價」應用在不同合併策略與不同應用問題上的效能，透過 Linux perf 分析工具實驗於 E5-2620 伺服器

5.1.2 應用於 VGLCS 算法更進一步優化 Peng 算法 1，將採用「懶插入」(lazy insertion) 技巧。在算法 1 中，在匹配第一個字串的第 i 個字元與第二個字串的第 j 個字元時，大部分的情況由於沒有匹配而填入元素 0 至 $V[i][j]$，存在大量的 0 插入到並查集中，每一次操作頻繁地與前一個 0 一同合併。因此，懶插入將會改善快取效能。

在懶插入操作中，我們優化許多次的插入、聯結許多個零元素。我們直到第一個非 0 元素 v，才將前大段的 0 元素一同合併，直接將每一個元素的父節點指向到 v 所在的集合。從實驗中，我們觀察到這減少指針鏈以及更新路徑效能。

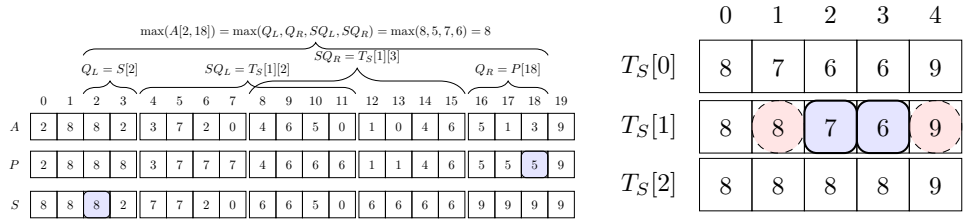
特別注意到我們不使用「懶插入」於平行環境下，那是因為多核心平台下的執行緒同步也是重要的效能指標。在平行 VGLCS 算法中，懶插入降低同步效率，故將不應用於平行實驗中。

5.2 平行區間詢問於 VGLCS 算法中

關於章節 3.1 所提及的塊狀稀疏表方法，我們額外維護兩個表以增進其效能，因此共包含三張表：其一，塊狀稀疏表 T_S，其二，前綴最大值表 (prefix maximum table) P，其三，後綴最大值表 (suffix maximum table) S。如同先前章節 3.1 的描述，表 T_S 解決 A 中連續完整塊的區間詢問，而前綴最大值表 P 維護各自塊內的前綴最大值，同理後綴最大值表 S，請參照圖 14 的說明。

對於任一區間詢問，將會被拆成 2 次的 T_S 上的詢問以及各 1 次的 P、S 上的詢問。例如圖 14 詢問區間 2 到 18 的最大值，拆分 2 次詢問於 T_S 中，分別為塊 1 到塊 2 的最大值、塊 2 到塊 3 的最大值，在塊 0 內的後綴最大值和塊 4 的前綴最大值。請參照圖請參照圖 14 的藍色圈選的部分說明。

我們認為存取表格的「順序」相當重要。在實作中，首先存取稀疏表 T_S，接著存取前綴最大值表 P，最後才存取後綴最大值表 S，其原因在於以下幾點：由於我們存取 T_S 時會



(a) 塊內前綴、後綴最大值表 (b) 稀疏表 T_S

Fig. 14. 塊狀稀疏表 T_S、前綴最大值表 P、後綴最大值表 S

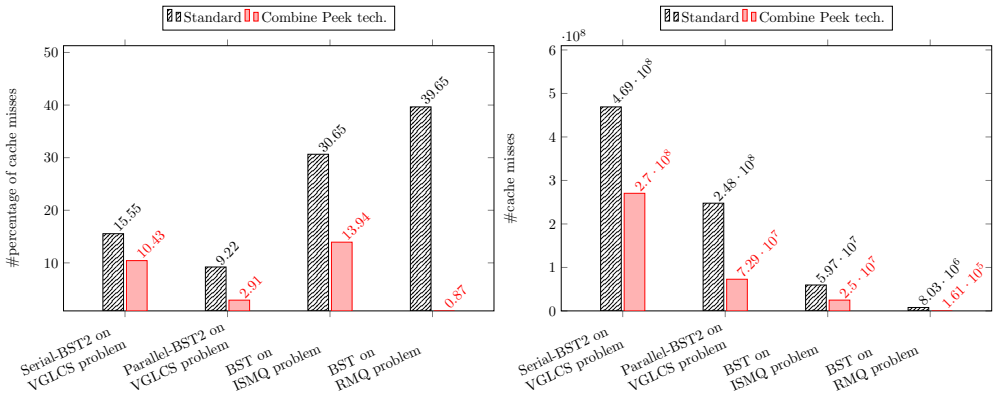


Fig. 15. 「偷看」於不同應用與環境上的效能，透過 Linux perf 分析工具實驗於 E5-2620 伺服器

在二維陣列的同一層，對於快取存取時容易一同帶入。如圖 14(b) 所描述，我們依序存取 $T_S[1][2]$ 時，硬體快取容易一同帶入 $T_S[1][3]$。

接著，我們使用「偷看」(peeking) 的方式，將存取 T_S 時的左右鄰居紀錄下來，在前半部的存取偷看前一個元素，在後半部存取偷看後一個元素。

如圖 14(b) 的說明，將偷看 $T_S[1][1]$、$T_S[1][4]$ 兩個元素，由於它們分別表示該塊內的最大值，必然會大於等於任意的塊內的前綴或後綴最大值。意即如果 $T_S[1][4]$ 已經小於目前的最大值，這使得我們無須到表 P 中存取，同理 $T_S[1][1]$ 也類似防止到表 S 中存取。「偷看」將會改善不少的效能，這些偷看的原因大部分落在同一段快取中，減少存取 P 和 S 的機會，便大幅度減少快取未中的可能。

算法 9 呈現我們在實作中的存取順序，以減少快取未中的方法。特別注意，存取前綴最大值表 P 先於後綴最大值表 S，原因在於動態規劃法的運作模式使得索引值大的元素值有大於索引值小的趨勢。

實驗結果仍需要額外的 $O(n)$ 空間，而偷看技巧可以應用上 $n = 10^4$ 時，提供 8% 的效能改善。若單看詢問操作，當 $n = 10^7$ 時可以改善 35% 的效能。請參照圖 15 的說明。

ALGORITHM 9: The process of answering a range query. Note that it accesses T_S, then P, then S.

Input: A: the input array;
P, S: the prefix/suffix maximum arrays for each block ;
$\mathcal{T}$: the Catalan index array for each block ;
T_s: the blocked sparse table ;
$[l, r]$: the bound of the ranged query ;
Output: v: the maximum value in $A[l..r]$;
if l and r are in the same block b then
    $i \leftarrow$ Query the ranged maximum query $[l, r]$ on $\mathcal{T}_k$, where k is the Catalan index of the b ;
    return $A[i]$;
end
$v \leftarrow -\infty$;
$l' \leftarrow$ The starting index of the next block from the position l ;
$r' \leftarrow$ The ending index of the previous block from the position r ;
$t \leftarrow \lfloor \log_2(r' - l' + 1) \rfloor$;
if $l' \le r'$ then
    $v \leftarrow \max(T_s[t][l' + 2^t - 1], T_s[t][r'])$;
end
if $T_s[t][l' + 2^t - 2] > v$ then // Access P only when necessary
    $v \leftarrow \max(v, P[r])$;
end
if $T_s[t][r' + 1] > v$ then // Access S only when necessary
    $v \leftarrow \max(v, S[l])$;
end
return v ;

6 實驗結果

實驗主要分成以下三種：其一，測試分塊與未分塊之間的稀疏表，測試平行區間詢問的效能；其二，增長後綴最大值詢問於不同的資料結構上的效能差異；最後，計算不同資料結構應用在 VGLCS 計算上的效能差異。

我們實驗於 Intel Xeon E5-2620 2.4 Ghz 處理器，其包含 384K bytes 的 L1 快取、1536K bytes 的 L2 快取、15M bytes 的 L3 快取。而 Intel CPU 支持「超執行緒」(hyper-threading)，每一處理器包含 6 個核心。使用的作業系統為 Ubuntu 14.04，程序撰寫使用 C++ 和 OpenMP，編譯器使用 gcc 以及相應的編譯參數 `-O2` 和 `-fopenmp` 進行實驗。

6.1 分塊與未分塊稀疏表

首先，我們比較未分塊稀疏表 (章節 2.4.1 所述) 和分塊稀疏表 (章節 3.1 所述) 於平行區間詢問的效能。在分塊稀疏表中，我們使用「右側棧出」編碼，而非查找 LCA 表，因我

們發現「右側棧出」編碼比 LCA 表來得更有效率。接續的實驗中，我們實驗不同區間長度與不同元素個數 N 的關係。

參照表 1 比較分塊與未分塊的稀疏表在平行區間詢問的效率。從實驗中，發現到使用右側棧出的分塊稀疏表比未分塊的版本來得快，效能隨著 N 增加而更加明顯。如在 N 達到 10^5 時，分塊稀疏表相交於未分塊快上 1.4 倍。

我們相信限制最大區間詢問的長度 (在表 1 中的 L) 影響存取稀疏表的快取效能。此外，當固定元素個數為 N 時，隨著 L 增加，加速比例越明顯，其原因可能為以下幾點所致：當詢問的區間長度較大時，分塊稀疏表只需要在 $\log N/s$ 層之間跳躍存取，2 次超塊詢問以及 2 次 $O(s)$ 的塊內詢問。相反地，未分塊稀疏表需要在 $\log N$ 層之間存取，因此在不同層數之間存取稀疏表的影響下，易造成應答區間詢問時，大部分存取與先前不同的層，導致資料局部性下降而產生快取未中。然而，上述的問題在 N 小不會發生，唯有 N 大到影響快取效能才使得分塊效益更為顯著。

Table 1. Total running time (ms) for finding RMQ of different sizes N and maximum interval sizes L.

	$N = 30000$			$N = 50000$			$N = 100000$			
L / Method	2^{10}	2^{12}	2^{14}	2^{10}	2^{12}	2^{14}	2^{10}	2^{12}	2^{14}	2^{16}
$ST_{standard}$	10.0	11.7	12.9	13.7	15.7	17.5	23.0	26.1	30.1	35.1
$BST_{rightmost-pops}$	9.6	9.8	11.3	13.8	13.5	13.6	23.8	22.8	22.9	24.9
speedup	1.01	1.19	1.14	0.99	1.16	1.28	0.96	1.14	1.31	1.40

6.2 「右側棧出」與「LCA 表」

這一小節，我們比較四種資料結構解決「增長後綴詢問」(incremental suffix query) 的能力，這四個資料結構分別為並查集、未分塊稀疏表、使用右側棧出編碼的塊狀稀疏表以及使用 LCA 表的塊狀稀疏表。使用並查集解決增長後綴詢問的細節請參照章節 3。對於並查集的實作細節，使用「路徑壓縮」(path compression) 和「秩合併」(merge-by-rank) 的策略，每個操作的攤銷時間複雜度為 $O(\alpha(n))$。在分塊稀疏表中，我們選用塊大小 s 為 8，所有稀疏表建造時皆以「行為主」(row-major manner) 的管理來減少快取未中，請參照算法 3 和圖 5 的說明。

接下來的簡化案例中，我們交替使用「附加」(appending) 和「詢問區間」(range querying) 操作，實驗中的長度和詢問位置皆採用均勻分布，如圖 16 說明四個不同資料結構在增長後綴詢問的效能。其中，右側棧出編碼的分塊稀疏表快於其他的資料結構。當 n 達 10^6 時，分塊稀疏表皆快於並查集。而當 n 達 10^7 時，右側棧出編碼的稀疏表比並查集快上 1.8 倍。

在另一個複雜的案例中，我們討論在動態規劃上的一些變因如何影響效能，這些變因包含插入值分布、最大區間詢問的分布以及插入與詢問之間的比例。為方便討論，我們使

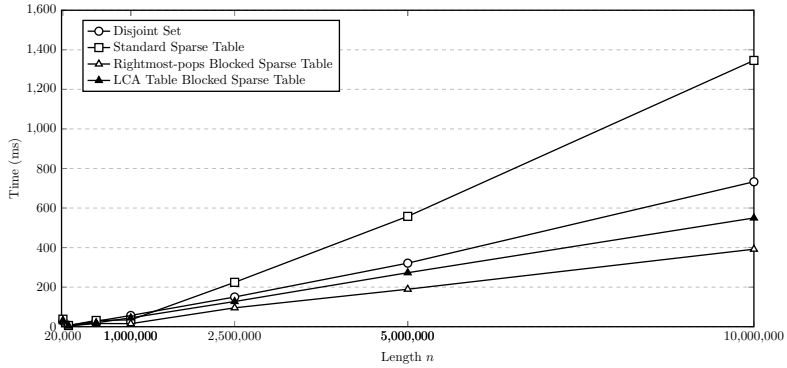


Fig. 16. 不同資料結構在增長後綴最大值詢問的效能差異，實驗環境於 E5-2620 機器上

用 p 表示插入比前一個更大的元素機率、q 表示插入一個零元素的機率和 L 表示詢問區間的最大長度。在我們的實驗中，我們固定元素個數 N 為 10^7，詢問區間最大長度 L 為 4 到 16 之間，p 和 q 之間的機率於 0% 到 100% 且詢問次數為插入次數的 10 倍。

如圖 16，我們觀察到分塊稀疏表比並查集或者未分塊都來得好，所以我們著重要塊狀稀疏表的實作，意即我們先前討論的右側棧出和 LCA 表的實作方式，而右側棧出編碼對於增長詢問很容易地想到，只需要對最後一個塊直接添加資訊即可。

表 2 比較「右側棧出編碼」與「LCA 表」解決增長後綴區間詢問的運行時間，特別注意到 LCA 表的方法提供理論上攤銷時間 $O(1)$ 的時間，而先前的實驗中，右側棧出編碼明顯地快於 LCA 表 1.5 之多，我們相信有以下兩個原因導致這個現象：其一，LCA 表需要更多的指令來計算出卡塔蘭索引值，相反地，右側棧出編碼只需要維護推出次數；其二，根據理論 [6] 選擇塊大小 s 為 $\frac{\log n}{4}$，這影響到無法使用太大 s，因為 LCA 表所造成的空間過大，越大則造成更多的快取未中問題發生。

Table 2. The timing (in seconds) of answering incremental suffix maximum query using rightmost-pops sparse table and the theoretically better LCA table sparse table (in bold font).

	$L=4$					$L=8$					$L=16$					
q \ p	0%	25%	50%	75%	100%	0%	25%	50%	75%	100%	0%	25%	50%	75%	100%	speedup
0%	1.15	0.89	0.86	0.88	0.91	0.88	0.87	0.87	0.85	0.87	1.02	1.00	0.99	1.00	1.02	1.56
	1.30	**1.05**	**1.05**	**1.05**	**1.05**	**1.32**	**1.32**	**1.32**	**1.32**	**1.32**	**1.35**	**1.34**	**1.34**	**1.34**	**1.34**	
20%	0.98	0.95	0.98	0.99	0.96	1.16	1.16	1.19	1.19	1.18	1.24	1.28	1.31	1.25	1.21	1.26
	1.09	**1.09**	**1.09**	**1.09**	**1.09**	**1.40**	**1.40**	**1.40**	**1.40**	**1.40**	**1.53**	**1.53**	**1.53**	**1.53**	**1.53**	
40%	1.01	1.01	1.02	1.02	0.99	1.23	1.24	1.25	1.24	1.21	1.39	1.43	1.45	1.31	1.26	1.28
	1.13	**1.13**	**1.13**	**1.13**	**1.12**	**1.46**	**1.46**	**1.47**	**1.47**	**1.45**	**1.62**	**1.62**	**1.62**	**1.62**	**1.61**	
60%	1.01	1.02	1.04	1.02	0.99	1.23	1.25	1.27	1.26	1.20	1.44	1.48	1.51	1.34	1.26	1.28
	1.13	**1.14**	**1.15**	**1.14**	**1.12**	**1.47**	**1.48**	**1.50**	**1.49**	**1.45**	**1.63**	**1.64**	**1.66**	**1.65**	**1.61**	
80%	0.99	1.01	1.03	1.01	0.97	1.20	1.22	1.25	1.23	1.15	1.38	1.43	1.49	1.31	1.19	1.31
	1.11	**1.12**	**1.15**	**1.12**	**1.10**	**1.44**	**1.46**	**1.49**	**1.47**	**1.41**	**1.59**	**1.61**	**1.65**	**1.63**	**1.56**	
100%	0.94	0.96	1.00	0.97	0.91	0.96	1.01	1.01	0.99	1.03	1.09	1.12	1.16	1.34	1.20	1.39
	1.04	**1.06**	**1.10**	**1.07**	**1.03**	**1.34**	**1.36**	**1.39**	**1.36**	**1.33**	**1.39**	**1.41**	**1.44**	**1.41**	**1.39**	

我們觀察到機率 p 將影響「偷看」技術中的改善效能比例。當 p 越接近 1，則偷看技術越能帶來更多的效能改善。偷看操作也依賴塊的大小，而我們也知道右側棧出編碼能使用的塊大於 LCA 表的版本，故改善情況的程度幅度不盡相同。

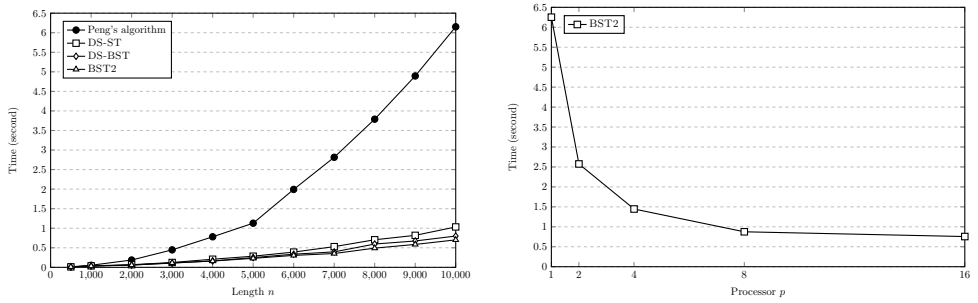
我們觀察到機率 q 越接近 1，則右側棧出編碼與 LCA 表的效能差意越不明顯，這可以明白 LCA 表在計算卡塔蘭索引值上的問題。當 q 越接近 1，卡塔蘭索引值將會重複計算更多次，因此效能會越接近右側棧出編碼的版本。

在區間詢問的長度 (表 2 中的 L) 影響到右側棧出的稀疏表，卻不影響 LCA 表的效能，其原因在於算法 4 的內層迴圈，將運行最多 s 次，而實作採用 s 為 16，將造成詢問長度增加而效能下降。在另一方面查找 LCA 表皆為 $O(1)$ 操作，效能不受 L 增長而改變。

6.3 可變的間隙限制最長共同子序列

我們接下來比較四種資料結構的組合於 VGLCS 的問題上，並統計它們的運行效能。第一種採用 Peng 算法的循序版本，其採用並查集於兩個階段，由於並查集只支持增長後綴最大值詢問，因此兩階段接使用後綴最大值詢問。除了第一種外，另外三種組合皆以平行方式運行。第二種 DS-ST 採用並查集和未分塊稀疏表，分別於算法的第一階段和第二階段。而第三種 DS-BST 採用並查集和塊狀稀疏表，分別於算法的第一階段和第二階段。同理，第四種 BST2 在兩階段皆使用塊狀稀疏表。特別注意到，上述的塊狀稀疏表皆使用「右側棧出」編碼。

圖 17(a) 展示四種不同組合的資料結構對於不同問題大小的效能，輸入字串使用字母集 $\{A,T,C,G\}$ 隨機產生，因為這是生物學常用的幾個字母。我們從實驗中發現 BST2 遠比其他平行版本來得好。圖 17(b) 展示我們提出最好的 BST2 資料結構，在 6 核心且支持超執行緒的環境下，提供至少 8 倍快的效能改善。



(a) 四種不同組合的運行時間　　(b) 右側棧出編碼的規模伸縮性

Fig. 17. 實驗於 E5-2620 主機，其包含 2 個 6 核心處理器且支持超執行緒技術

7 相關研究

大部分的 LCS 平行方法都著重在「波前平行」，然而此方法維護波形掃描整個動態規劃表，大部分的波前計算對快取並不友善，也就是說當實驗環境存在「快取」，將無法高效率地運行。為解決此問題，Maleki [11] 等人提出一些技巧來拓展動態規劃問題的平行度。

取代波前平行的方法如常見的「列接列」，如在 Peng 算法 [14] 提供 $O(nm\alpha(n))$ 解決 VGLCS 問題，或者漸近最好的 $O(nm)$ 算法，其中 α 為阿克曼函數的反函數。

使用傳統列接列的平行方法存在許多挑戰，在 VGLCS 應用上需解決有效率的後綴或區間詢問於平行環境。Peng 算法索使用的並查集，其由 Gabow [8] 和 Tarjan [16] 提出和分析，可用來解決後綴最大值詢問。而我們使用稀疏表 [3] 來支持增長後綴/區間最大值詢問來解決 VGLCS 問題，稀疏表其易於實作的特性代給平行環境下較好的效能。

理論上的進展，由 Fischer [6] 提出塊狀稀疏表以改善未分塊稀疏表 [3] 的效能，我們使用塊狀稀疏表於本篇論文的實驗中。而 Fischer [6] 提出的最小共同祖先表來解決塊內詢問，我們則是使用本篇提出的「右側棧出」來編碼笛卡爾樹。

Demaine [4] 提出「快取適性」的操作於笛卡爾樹 [17]，可解決 Fischer 使用 LCA 表造成的快取未中問題。而 Masud [9] 則使用另一種編碼方式來減少計算編碼的使用次數。在這篇論文中，我們減化「右側棧出」編碼，其減化了 Demaine 提出的編碼方法，提高了編碼速度與改善快取未中的問題。

最後，本篇想指出目前最好的方法，可能都不太適用於「動態」笛卡爾樹編碼，或者不支援平行的 LCA 建表，也就是說它們是用於「離線」或者「循序」環境。相反地，我們在章節 4.3 提出的動態笛卡爾樹編碼，不僅支持有效率的區間詢問，且支持增長資料的操作，同時更能更有效地平行建造 LCA 表。

8 結論

這篇論文中，我們提出以列接列的平行方法，將兩階段 VGLCS 算法充分平行，而提出的兩階段的算法相較於一般的波前平行方法來得更有效率，且運行的規律性和其易實作造就更好的效能。

我們使用「稀疏表」來解決可變的間隙限制最常共同子序列，而稀疏表可在平行環境下提供更好的同步效能和均勻的工作負載，這些都是並查集無法達到的。此外，我們提出「右側棧出」編碼，其易於實作且運行地相當高效於平行環境。最後，我們的 VGLCS 算法使用「右側棧出」的稀疏表，運行時間為 $O(n^2 s/p + n \max(\log n, s))$，其中 n 為元素個數、p 為處理器個數、s 為塊大小。

我們提出的標記方法採用字典順序，將任意二元搜尋樹找到相應的卡塔蘭索引值，接著使用這一標記方法提供笛卡爾樹的編碼方式於攤銷 $O(1)$ 時間，最後，VGLCS 問題使用這些技巧於時間 $O(n^2/p + n \log n)$ 內被解決。

我們提出的動態卡塔蘭索引值的算法，搭配稀疏表可解決增長區間最大質詢問，每一詢問可在攤銷時間 $O(1)$ 解決，使得 VGLCS 算法的時間複雜度為 $O(n^2/p + n\log n)$。特別注意到動態卡塔蘭索引值計算，可以普遍使用在數據不斷插入的二元樹的情況下。

從實驗中觀察到兩個有趣的結果：其一，在平行環境下，塊狀稀疏表效能上比未分塊的稀疏表好上許多，適當大小的塊更能給予更好的效能。其二，漸近最好的算法在實作中，未必能提供最好效能，如我們提出的「右側棧出」編碼在詢問操作為 $O(s)$ 時間，實際表現遠比理論攤銷時間 $O(1)$ 來得更好。最後，我們相信易於實作且直觀的算法將會是高效能的關鍵。

9 未來展望

許多研究著重於如何最小化編碼的計算量，如重新定義遞迴公式。我們便可以使用數個暫存器狀態間的轉換取代記憶體存取，然而這些研究仍在離線編碼中應用居多。這篇論文中在處理後綴最大值時，提供的方法尚未僅使用暫存器進行轉換，若能改變數學上的定義，效能將會有更明顯地改善。

Acknowledgments

能順利地完成這篇論文，首先，感謝劉邦鋒和吳真貞老師的指導，在論文架構和描述手法的教導，才使得篇幅雜亂的初稿便得更加地易於理解，討論過程中更加地精練專業知識，學生在此衷心感謝老師。

特別感謝中央大學的郭人維學長，拉拔在演算法及資料結構領域上的研究，其給予的助力使得論文開花結果。更感謝在網路上許許多多來自各方的朋友分享研究心得，讓彼此切磋茁壯。

在進入臺灣大學研究所的這兩年中，感謝實驗室學長們的鼓勵與支持，本對於學術研究文化和自身能力的迷茫，接受學長們的啟示後，最終得以撐過第一學年。在第二年中，感謝共同奮戰論文的古耕竹、古君葳、鄭以琳、吳軒衡等同學，彼此加油打氣，使得在撰寫論文的路上並不孤單，督促進展、協助撰寫與驗證想法更加地順利，願你們也能順利畢業、研究出滿意的成果。

在研究實驗上，感謝實驗室學弟張逸寧、林明璟、蔡慶源、朱清福的參與，被迫實作出放置於批改娘系統上題目，這些題目原本為論文的一小部分，可透過不同資料結構與算法解決，在本篇追求效能極致的路上貢獻了一份心力，為本實驗結果給予更有信心的立論基礎。願你們在接下來的一年裡，經過老師指導與同學們相互引領下順利畢業。

此外，特別感謝高中時期帶入門的溫健順老師，在選擇領域分組時，相信我在資訊領域上的發展，拉近資訊組培養，經歷三年的教導後，才能順利走向這一條道路。

最後，感謝家人們一路相伴，進入資訊工程領域後，經歷大學轉學、延畢到研究所的路上，對我的選擇給予支持。

感謝上述的各位與師長們一路上的支持與資助。

References

[1] Hsing-Yen Ann, Chang-Biau Yang, Yung-Hsing Peng, and Bern-Cherng Liaw. 2010. Efficient algorithms for the block edit problems. Inf. Comput. 208 (2010), 221–229.

[2] Lech Banachowski. 1980. A Complement to Tarjan's Result about the Lower Bound on the Complexity of the Set Union Problem. Inf. Process. Lett. 11 (1980), 59–65.

[3] Omer Berkman and Uzi Vishkin. 1993. Recursive Star-Tree Parallel Data Structure. SIAM J. Comput. 22 (1993), 221–242.

[4] Erik D. Demaine, Gad M. Landau, and Oren Weimann. 2009. On Cartesian Trees and Range Minimum Queries. In Algorithmica.

[5] Edsger Dijkstra. 1976. A discipline of programming. Prentice-Hall, Englewood Cliffs, N.J.

[6] Johannes Fischer and Volker Heun. 2006. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In CPM.

[7] Johannes Fischer and Volker Heun. 2007. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In ESCAPE.

[8] Harold N. Gabow and Robert E. Tarjan. 1983. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. J. Comput. Syst. Sci. 30 (1983), 209–221.

[9] Masud Hasan, Tanaeem M. Moosa, and Mohammad Sohel Rahman. 2010. Cache Oblivious Algorithms for the RMQ and the RMSQ Problems. Mathematics in Computer Science 3 (2010), 433–442.

[10] Daniel S. Hirschberg. 1975. A Linear Space Algorithm for Computing Maximal Common Subsequences. Commun. ACM 18 (1975), 341–343.

[11] Saeed Maleki, Madan Musuvathi, and Todd Mytkowicz. 2016. Efficient parallelization using rank convergence in dynamic programming algorithms. Commun. ACM 59 (2016), 85–92.

[12] David Mount. 2001. Bioinformatics : sequence and genome analysis. Cold Spring Harbor Laboratory Press, Cold Spring Harbor, N.Y.

[13] Md. Mostofa Ali Patwary, Jean R. S. Blair, and Fredrik Manne. 2010. Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure. In SEA.

[14] Yung-Hsing Peng and Chang-Biau Yang. 2011. The Longest Common Subsequence Problem with Variable Gapped Constraints.

[15] Mohammad Sohel Rahman and Costas S. Iliopoulos. 2006. Algorithms for Computing Variants of the Longest Common Subsequence Problem. In ISAAC.

[16] Robert E. Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM 22 (1975), 215–225.

[17] Jean Vuillemin. 1980. A Unifying Look at Data Structures. Commun. ACM 23 (1980), 229–239.

[18] Jiaoyun Yang, Yun Xu, and Yi Shang. 2010. An Efficient Parallel Algorithm for Longest Common Subsequence Problem on Gpus.