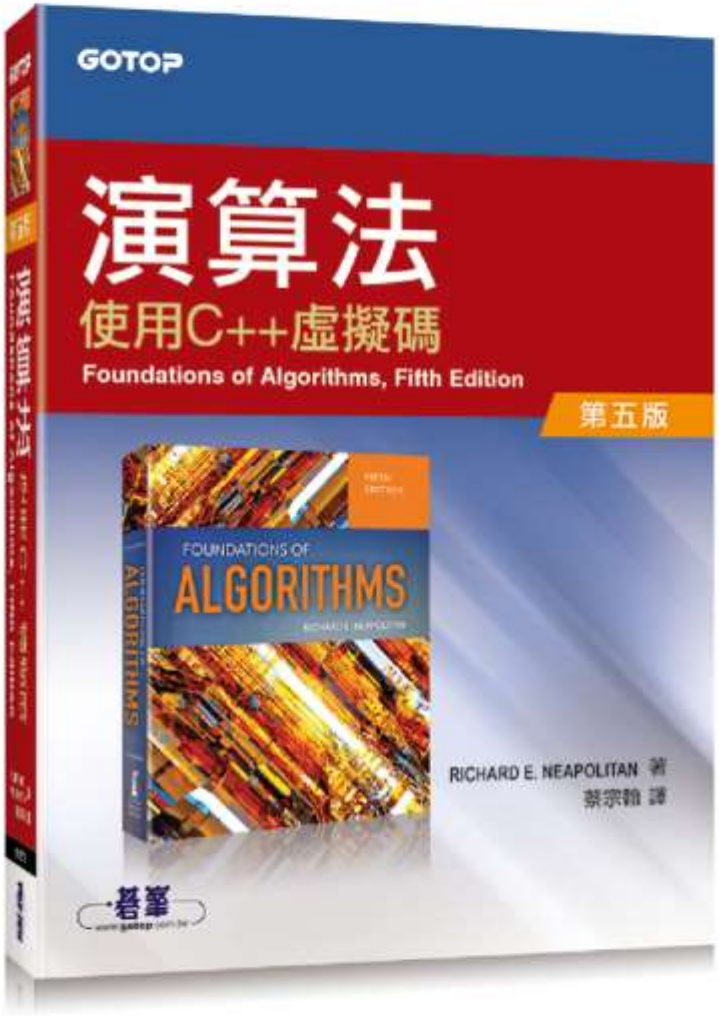


第四章
貪婪演算法

演算法
使用C++虛擬碼
Foundations of Algorithms, Fifth Edition
第五版

第四章貪婪(greedy)演算法

貪婪演算法(greedy algorithm)

- 不使用分割的方法，而是經過一步步地選擇過程來求解
- 每一步都是衡量當時最佳的情況而做出的選擇，也就是局部最佳解(local optimal)
- 這樣做的目的是希望經過所有的步驟所得的答案會是全域最佳解(global optimal)
- ➔結果不盡然如此！

從找錢的例子說起-1

- 問題：
  - 找給顧客的幣值要對，而且找的銅板數是最少的。
- 解法：

```
While （在"還有多餘硬幣"與"此問題尚未解出"之狀況下 ）{
  拿取剩餘硬幣中幣值最大的一枚；                          // 選擇程序
  if （該枚硬幣幣值加上目前"已經找給顧客的零錢總數"超過
                    "應找給顧客的最後總數"）              // 可行性檢查
      放回該枚硬幣；
  else
      將該枚硬幣幣值加上目前"已經找給顧客的零錢總數"；
  if （"已經找給顧客的零錢總數"
                    等於"應找給顧客的最後總數"）          // 解答檢查
      此問題解答完成；
}
```

從找錢的例子說起-2

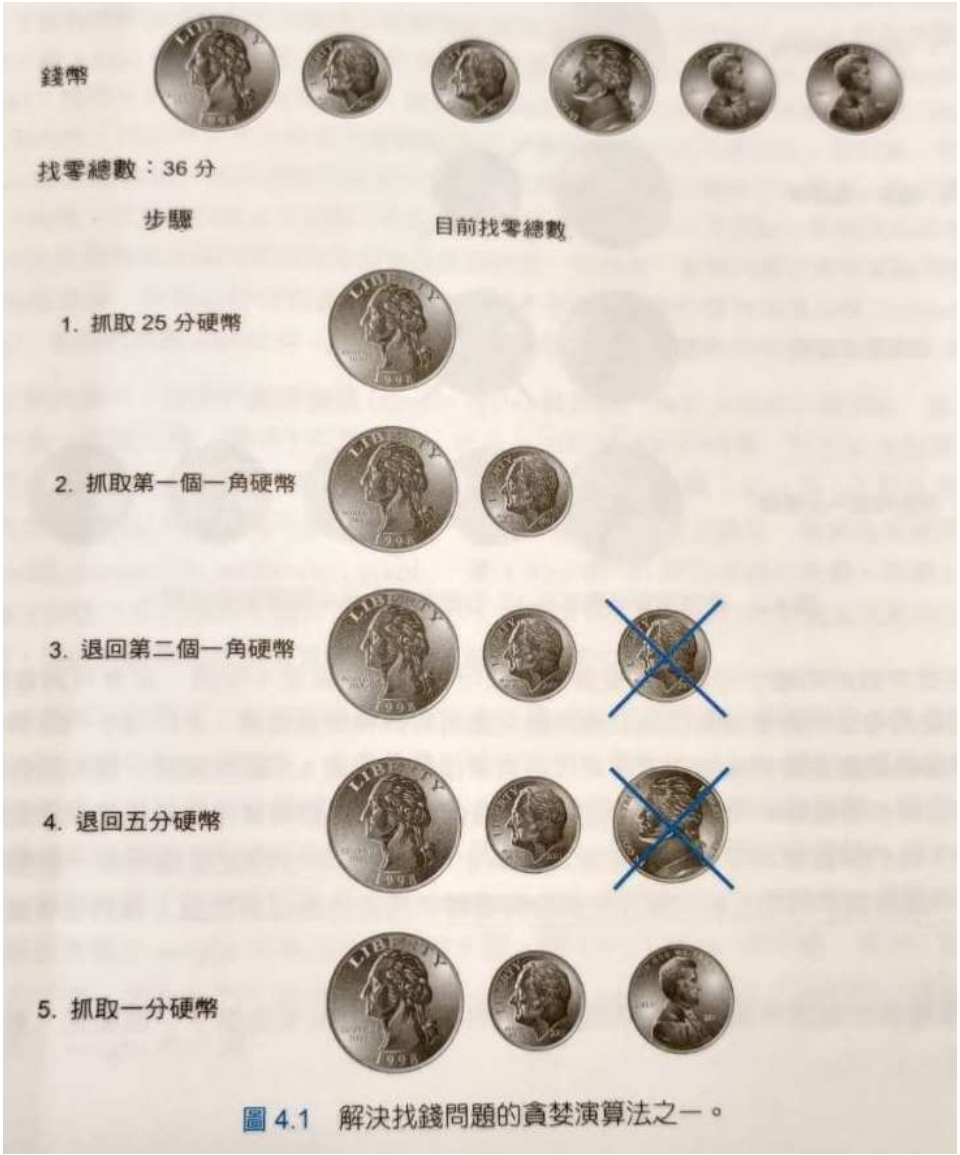


圖 4.1 解決找錢問題的貪婪演算法之一。

從找錢的例子說起-3

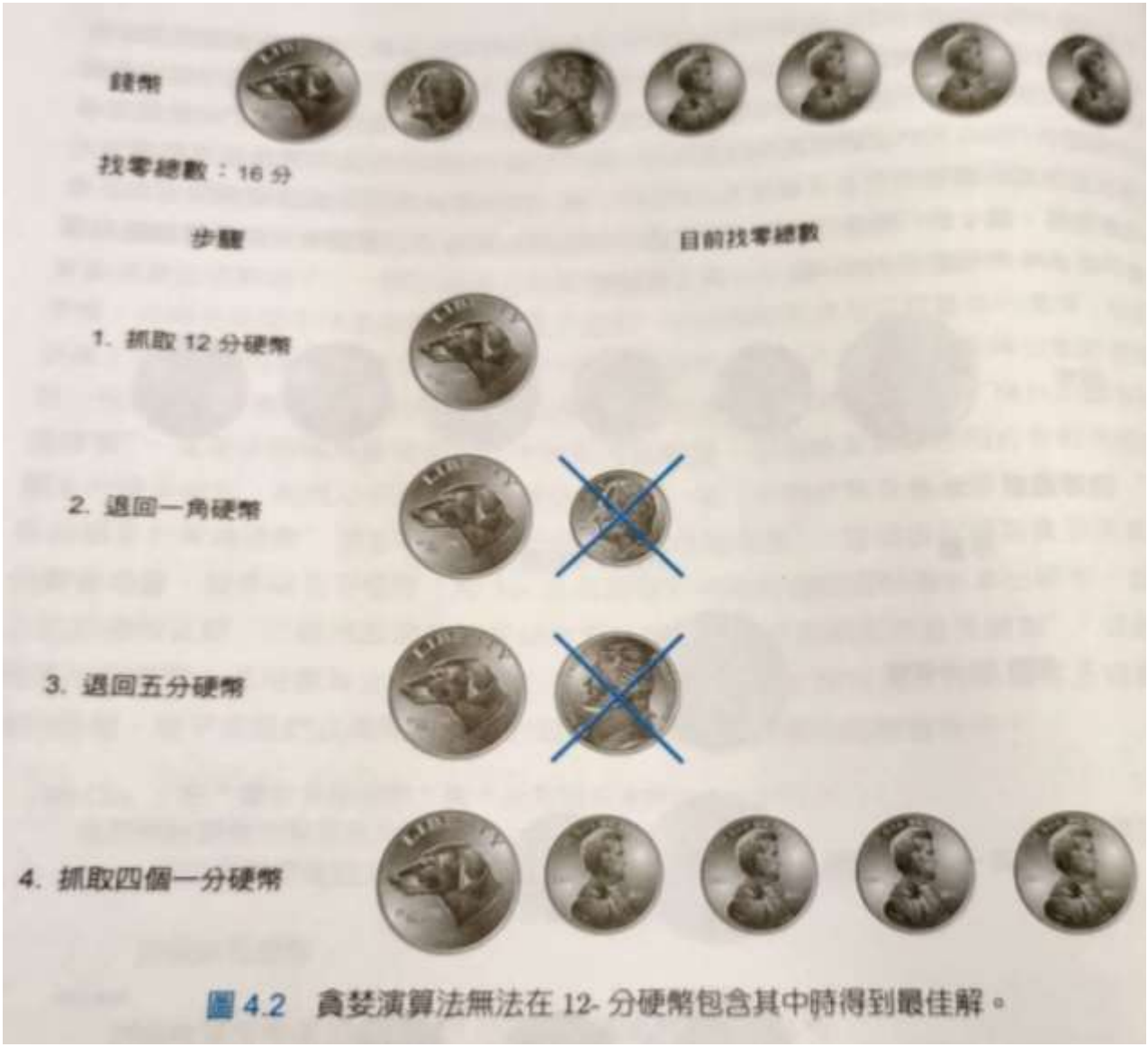
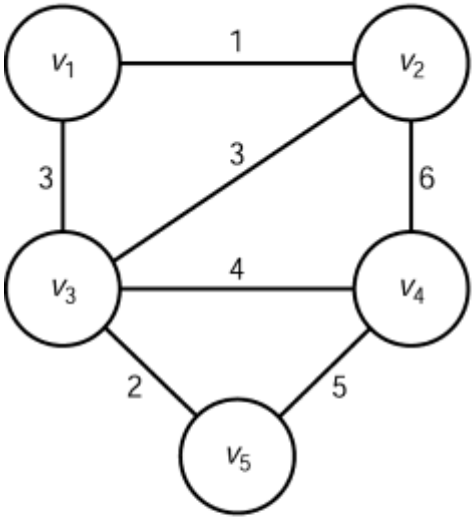


圖 4.2 貪婪演算法無法在 12- 分硬幣包含其中時得到最佳解。

4.1 最小生成樹(minimal spanning tree)

- 如何移除無向權重連通圖G的某些邊線，使得移除之後這些邊線之後得到的子圖仍保持連通性(connected)，且盡量降低該子圖所有邊線之weight總和
- 所謂G的生成樹 (spanning tree)就是：包含G中所有頂點並且符合樹的定義的連通子圖(connected subgraph)。

(a) 一個無向連通權重圖



(b) 即使(v_4, v_5)被移除，此子圖還是保持連通性

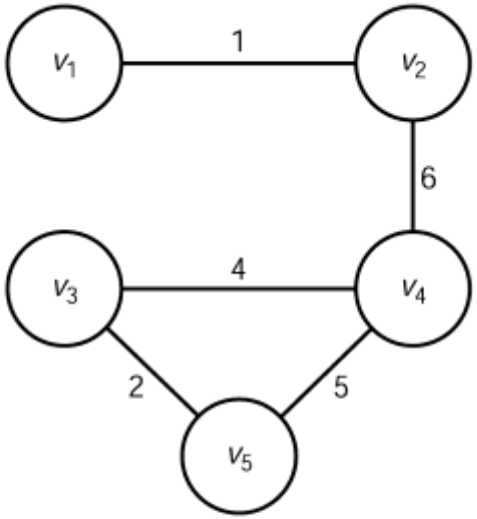
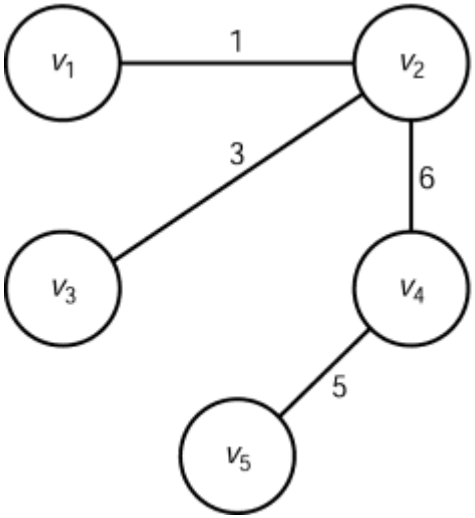
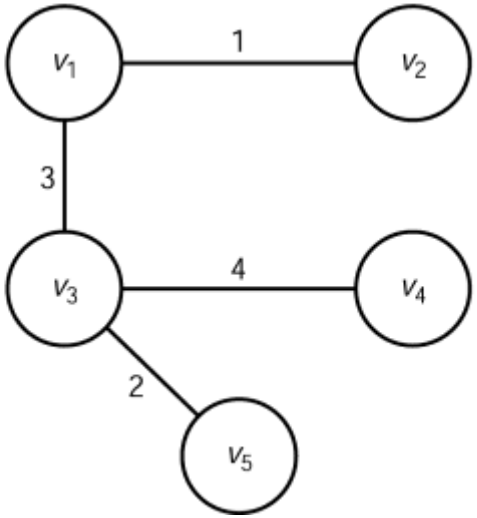


圖4.3 一個權重圖及其三個子圖

(c) G的生成樹



(d) G的最小生成樹



- G中的生成樹T與G其實有相同的頂點集合V，但是T的所有邊線構成的集合F是E的子集
- 我們以$T = (V, F)$來表示該生成樹。我們的目標即是找出E的子集合F使得$T = (V, F)$為G的最小生成樹。下面為解此問題的高階貪婪演算法：

```
F = ∅                                          // 將 edge 集合初始化為空集合
while (當此問題尚未得解){
      根據某些會得到區域最佳解的方法來選出一條 edge          // 選擇程序
  if (將選出的 edge 加入集合 F 中不會產生任何 cycle)
      將選出之 F 加入 edge 中;                    // 可行性檢查
  if (T=(V, F) 是生成樹)                         // 解答檢查
      此問題得解;
}
```

4.1.1 Prim演算法

- 從空的邊線子集合F與含有任意頂點的子集合Y開始
- 首先，將Y的初始值設定為$\{v_1\}$。所謂與Y最接近的頂點，就是所有連接$V-Y$中的頂點與Y中的頂點的邊線中，weight最小的那條邊線，在$V-Y$那端的頂點。找到最接近Y的頂點後，我們將該頂點加入Y集合中，同時將有最小weight的邊線加入集合F中
- 不斷將最接近的點加入Y，直到$Y=V$為止
  - 例如在圖4.3(a)中，當$Y=\{v_1\}$時v_2是與Y最接近的頂點，此時我們就將v_2加入Y中，(v_1,v_2)加入F中
- 下面為描述這個過程的高階演算法：

```
F = ∅;                                  // 將 edge 集合初始化為空集合
Y = {v1};                               // 將頂點集合初始化為僅包含第一個頂點
while (當此問題尚未得解){
   選擇 V - Y 中的某一個頂點且          // 選擇程序可行性檢查
      該點與 Y 有最近的距離之條件
   將選出的頂點加入 Y 中;
   將選出之 edge 加入 F 中;
   if (Y == V)                          // 解答檢查
         此問題得解;
}
```

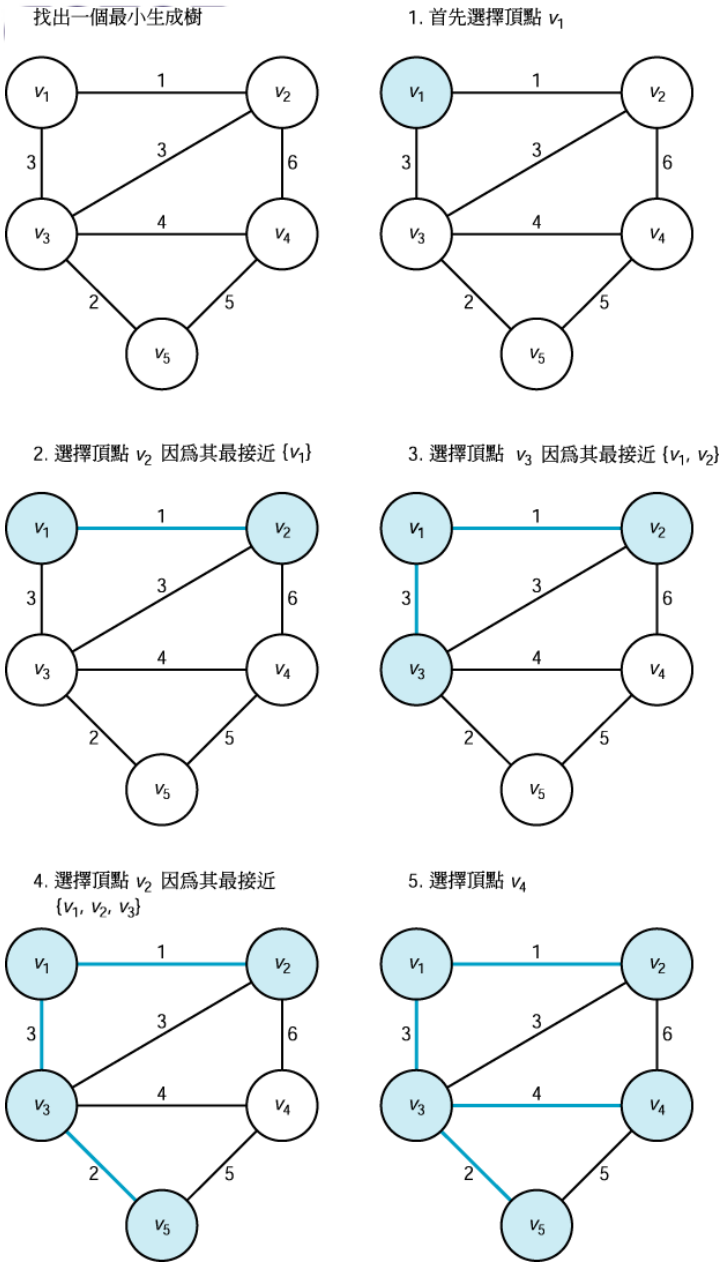


圖4.4 一個權重圖(於圖中左上角)以及在這個圖上面執行Prim演算法的過程。在每個步驟中，位於Y中的頂點及F中的邊線被塗成藍色。

演算法4.1 Prim 演算法

- 問題：如何找出最小生成樹
- 輸入：一個包含了*n*個頂點(*n*為大於等於2的整數)的無向權重連通圖。我們用一個二維陣列*W*來表示此圖，而*W*之行與列之索引值均由1到*n*。*W*[*i*][*j*]表示第*i*個頂點與第*j*個頂點連接而成的邊線之weight。
- 輸出：輸入圖的最小生成樹中的所有邊線構成的集合*F*。

```
void prim (int n,
           const number W[][],
           set_of_edges& F)
{
  index i, vnear;
  number min;
  edge e;
  index nearest [2..n];
  number distance [2..n];

  F = ø;
  for (i = 2; i <= n; i++){
     nearest [i] = 1;
     // 對於所有的頂點，設定 v1 為 Y 中離它們最近的頂點，並把該頂點與 vi 間的距離設為
        該頂點到 Y 的距離
     distance [i] = W[1][i];
}
```

```
    repeat (n - 1 times){
    // 將其他所有頂點共 n - 1 個加入 Y 中
      min = ∞;
      for (i = 2; i <= n; i++)    // 檢查每個不屬於 Y 中的頂找出與 Y 最接近的那一點
         if (0 ≤ distance [i] < min) {
            min = distance [i];
            vnear = i;
         }
         e = 連接 vnear 與 nearest[vnear] 兩頂點的 edge;
             將 e 加入 F;
         distance [vnear] = -1;          // 將 vnear 索引到的定點加入 Y
         for (i = 2; i <= n; i ++)
         if (W[i][vnear] < distance [i]) {
         // 對於 Y 集合以外的每個頂點自 Y 中更新其在陣列中之值
            distance [i] = W[i][vnear];
            nearest [i] = vnear;
         }
   }
}
```

分析演算法4.1

所有情況的時間複雜度(Prim演算法)

- 基本運算：在repeat迴圈中有兩個迴圈，每個迴圈各執行$n-1$次。執行迴圈中的所有指令一次可當作是執行基本運算一次。
- 輸入大小：n，頂點數目。

因為repeat迴圈會被執行$n-1$次，故其時間複雜度為：

$$T(n) = 2(n-1)(n-1) \in \Theta(n^2)$$

定理 4.1

Prim演算法必定可以產生一棵最小生成樹

- 證明：我們將利用數學歸納法證出在每次repeat迴圈執行過後，集合F均保持 promising。
- 歸納基點：空集合 ϕ 為promising。
- 歸納假設：假設在某次執行repeat迴圈過後，到那時為止所選擇的所有邊線構成的集合－亦即，F為promising。
- 歸納步驟：我們必須證明在邊線 e在下一個iteration中被選擇後，$F \cup \{e\}$ 為promising。因為e是連接Y 中頂點與$V - Y$中頂點具有最小weight的邊線，根據輔助定理4.1可知$F \cup \{e\}$為promising，故得證。

4.1.2 Kruskal 演算法

- 先自頂點集合V中產生等同於頂點數目且互不交集的頂點子集合，每個頂點子集合中僅有一個頂點
- 依照邊線所屬之weight值的大小作非遞減的排序，並根據此排序檢查各邊線。若該次檢查之邊線由位於兩個互不交集之頂點子集合中之兩個獨立頂點所連接而成，則此邊線即被選擇並加入邊線集合中，且兩個互不交集之頂點子集合則被合併為一個
- 此過程一直不斷重複到所有的頂點子集合都已經被合併在一起，成為一個包含所有頂點的集合為止。下面就是這個過程的高階演算法

```
F = ∅;                                          // 將邊線集合初始化為空集合
於 V 中產生等同於頂點數目且互不交集的頂點子集合，
每個頂點子集合中僅有一個頂點；
while （當此問題尚未得解） {
   選擇下一個邊線；                              // 選擇程序

   if （選出的邊線連接了兩戶不交集之子集合）{     // 可行性檢查
         合併該兩子集合；
         將選出之邊線加入集合；
   }
   if （所有的頂點子集合都已經被合併）           // 解答檢查
         此問題得解；
}
```

演算法 4.2 Kruskal 演算法

- 問題：如何找出最小生成樹
- 輸入：一個包含了 n 個頂點(n為大於等於2的整數)以及m個邊線(m為正整數)的無向權重連通圖。此圖以一個包含其所有邊線與各邊線的weight之集合E來表示之。
- 輸出：由最小生成樹構成之邊線集合F。

```
void kruskal (int n, int m,
              set_of_edges E,
              set_of_edges& F)
{
  index i, j;
  set pointer p, q;
  edge e;
  將集合 E 中的 m 個邊線以 weight 值大小的非遞減排序 ;
  F = ø;
  initial (n);                          // 初始化 n 個互不交集的子集合
  while ( 集合 F 中的邊線個數少於 n - 1){
    e = 尚未檢查到且有最小 weight 值之 edge;
    i, j = e 邊線上之兩頂點 ;
    p = find(i);
    q = find(j);
    if (! equal (p, q)) {
       merge (p, q);
       將 e 加入 F;
    }
  }
}
```

分析演算法4.2

最差情況的時間複雜度(Kruskal演算法)

- 基本運算：比較指令
- 輸入大小：n為頂點個數，m為邊線個數

在本演算法中有三種可能的考量點：

1. 將邊線排序所耗費的時間。對邊線作排序所耗費之時間複雜度為：

$$W(m) \in \Theta(mlgm)$$

2. while迴圈所耗費的時間。在最差情況下，在跳出while迴圈之前，每個邊線都要被考慮一次，共會執行這個迴圈m次。執行m次迴圈(其中包含了呼叫*find*、*equal*與 *merge*)的時間複雜度為：

$$W(m) \in \Theta(mlgm)$$

3. 初始化n個互不交集的集合所花的時間

$$T(n) \in \Theta(n)$$

- 由於$m \geq n-1$，故初始化時間主要耗費在排序與操作互不交集子集合上：

$$W(m,n) \in \Theta(mlgm)$$

- 可發現在最差狀況下時間複雜度與n無關。但在最差情況下，任一頂點可連接至其他所有頂點，這代表了：

$$m = \frac{n(n-1)}{2} \in \Theta(n^2)$$

- 因此我們可以歸納最差情況的時間複雜度如下：

$$w(m,n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 2lgn) = \Theta(n^2 lgn)$$

- 在比較Prim演算法與Kruskal演算法時，上面表示最差情況的兩個式子是很有用的。

$$T(n) \in \Theta(n)$$

- 由於$m \geq n-1$，故初始化時間主要耗費在排序與操作互不交集子集合上：

$$W(m,n) \in \Theta(mlgm)$$

- 由此可發現在最差狀況下時間複雜度與n無關。但是在最差情況下，任一頂點有連接至其他所有頂點的可能性，這代表了：

$$m = \frac{n(n-1)}{2} \in \Theta(n^2)$$

- 因此我們可以歸納最差情況的時間複雜度如下：

$$w(m,n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 2lgn) = \Theta(n^2 lgn)$$

4.1.3 Prim演算法與Kruskal演算法之比較

- 由上面推導可知兩者之時間複雜度如下：
- Prim演算法：$T(n) \in \Theta(n^2)$
- Kruskal演算法：$W(m,n) \in \Theta(mlgm)$ 及 $W(m,n) \in \Theta(n^2 lgn)$
- 我們也已證明出在連通圖中：

$$n-1 \leq m \leq \frac{n(n-1)}{2}$$

- 對於邊線數目m接近下限的graph來說，Kruskal演算法的時間複雜度為$\Theta(nlgn)$，此時使用Kruskal會有較佳的速度

- 但對於邊線數目接近上限的graph來說(也就是此 graph 為高度 connected)，此時Kruskal演算法的時間複雜度為$\Theta(n^2 lgn)$，即此時使用Prim演算法會有較佳的速度。

4.2 解單一起點最短路徑問題之Dijkstra演算法

- 單一最短路徑問題
  - 某特定頂點對其他頂點的最短路徑，不需要用到3.2節中那麼複雜的方法。
- 接下來我們將利用貪婪演算法的精神來發展一套複雜度為$\Theta(n^2)$的方法。此演算法源自於1959年，由 Dijkstra 發展出來

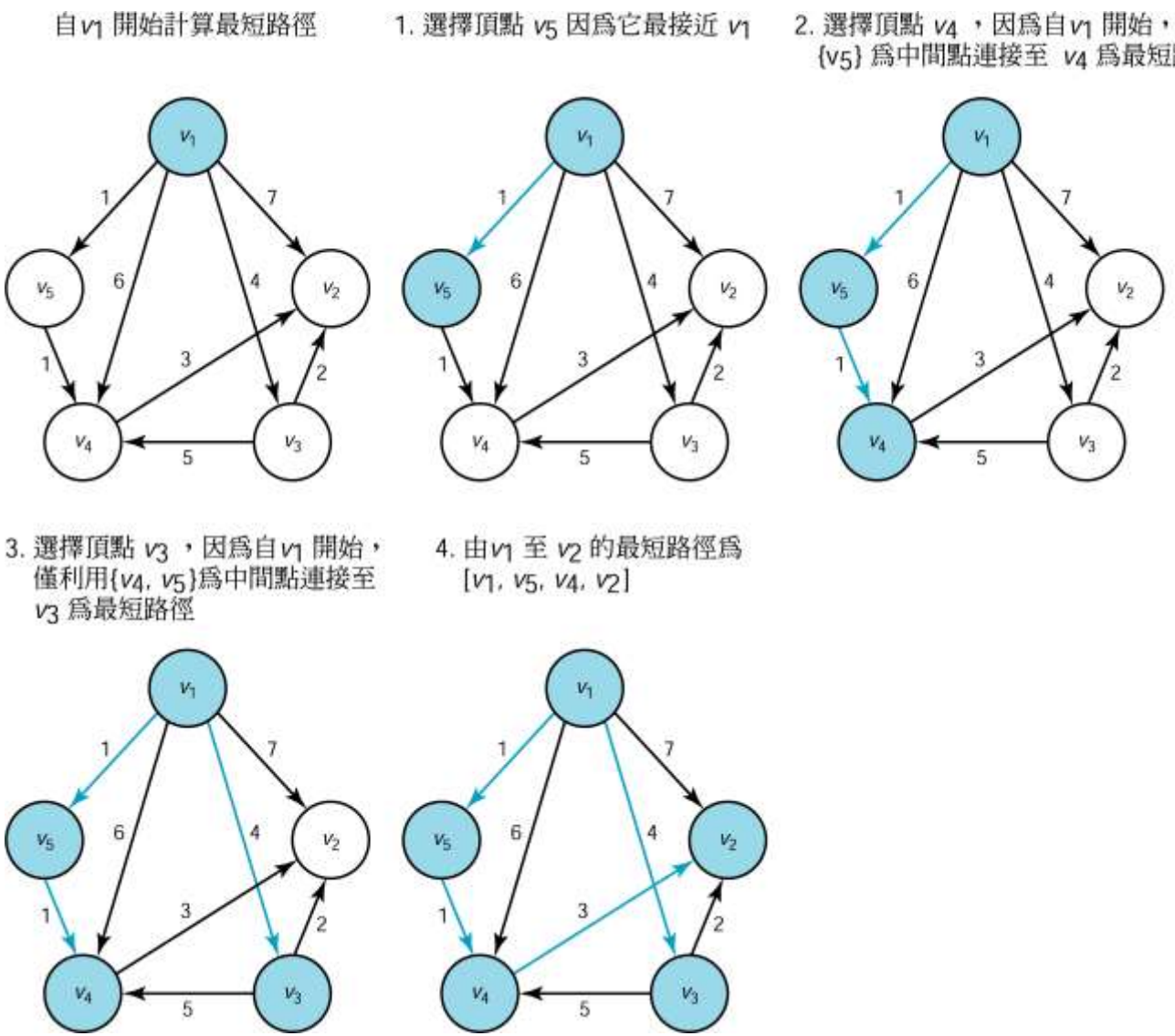
- 首先我們初始化Y集合使其中僅包含最短路徑已經被算出的頂點(此處設該頂點為 v_1)
- 另外我們將邊線集合F初始化為空集合。接下來選擇與v_1最接近之頂點v，並將其加入至Y集合中，同時將邊線$<v_1,v>$加入至F集合中($<v_1,v>$表示由v_1至v之有向邊線)。很明顯可看出此邊線為v_1至v間的最短路徑。
- 接下來我們檢查由v_1通到$V-Y$中之頂點的路徑，並只容許Y中的頂點成為路徑中的中間點。這些路徑中最短的即為最短路徑。將在此最短路徑上之終點(v_1為起點)加入Y集合中，並將此最短路徑加入至F集合中。
- 此過程一直持續至Y與V(包含所有頂點的集合)相等才結束。此時，F包含了答案的最短路徑中的所有邊線。下面就是這個過程的高階演算法。

```
Y = {v1};
F = ∅;

while （當此問題尚未得解）{
   從 V -Y 中選個頂點，                                    // 選擇程序
   使得在僅用 Y 中的頂點做為中間點的情況下，                // 可行性檢查
   該頂點到 Y 的路徑是最短的；
   將選出之頂點 v 加入 F 中；
   將選出之邊線（在最短路徑上）加入 F 中；
if （Y==V)
      此問題得解；                                         // 解答檢查
}
```

- 圖 4.8 中描述了Dijkstra演算法。上述高階演算法僅適用於人類以直觀的方式解出較小圖的問題時之狀況。因此在下面我們將講解更詳細的過程。
- 在此演算法中，我們可以仿照 3.2節中的做法，以二維陣列來表示權重圖，此法也非常類似 Prim演算法，不同之處在於需要將*nearest*與*distance*兩個陣列替換為*touch* 與*length*兩個索引值由2到*n*的陣列。
- *touch* [*i*] = 在*Y*中的頂點*v*的索引值，使得邊線＜*v* , v_i ＞為目前從v_1到v_i，僅使用*Y*中頂點做為中間點之最短路徑上的最後一條邊線。
- *length* [*i*] = 目前從v_1到v_i，僅使用*Y*中頂點做為中間點之最短路徑。

圖4.8 一個權重圖(圖中左上角) Dijkstra演算法在此圖上執行的步驟。在每個步驟中，位於Y的頂點及F中的邊線被塗成藍色。



演算法4. 3 Dijkatra 演算法

- 問題：求出以v_1為起點連接至其他各頂點的最短路徑。
- 輸入：一個包含了n個頂點的有向權重連通圖。可用一個行與列之索引值均為1至n的二維陣列W來表示此graph，$W[i][j]$為第i個頂點與第j個頂點連接而成的邊線之weight。
- 輸出：由這些最短路徑中所有邊線所構成之集合F。

```
void dijkstra (int n, const number W[][], set_of_edges& F)
{
  index i, vnear;
  edge e;
  index touch[2..n];
  number length[2..n];

  F = ø;
  for (i = 2; i <= n; i ++){
     touch[i] = 1;
     length[i] = W[1][i];
  }
```

// 對各頂點，將 v_1 設為以 v_1 為起點之目前最短路徑上的最後一個頂點，並將該路徑的長度啟始為從 v_1 連到該頂點的長度

```
    repeat (n - 1 times) {          // 將其他所有頂點共 n - 1 個加入 Y 中
        min = ∞;
        for (i = 2; i <= n; i++)                // 檢查每個頂點，找出符合
            if (0 ≤ length[i] < min) {          // 最短路徑之頂點
                min = length[i];
                vnear = i;
            }
        e = vnear索引到的頂點與 touch[vnear] 索引到的頂點所連成的邊線 ;
        將 e 加入 F;
        for (i = 2; i <=n; i++)
            if (length[vnear] + W[vnear][i] < length[i]) {
                length[i] = length[vnear] + W[vnear][i];
                touch[i] = vnear;           // 更新每個不在 Y 中的頂點的最短路徑
            }
            length[vnear] = -1;             // 將指標為 vnea r之頂點加入 Y 中
    }       // to Y.
}
```

4.3 排程

- 系統耗費時間
  - 等待時間加上服務時間

4.3.1 最小化總系統耗費時間

- 簡單方法
  - 即找出所有可能的工作排程方式與每一種工作排程所花費的時間並互相比較

範例4.2

給定三項工作以及其花費時間如下：

$t_1 = 5, t_2 = 10, t_3 = 4$

若我們按照 工作1，工作2，工作3 的順序來排定工作，則這三項工作的系統中耗費時間可列於下表：

工作	系統中耗費時間
1	5（工作 1 之服務時間）
2	5（等待工作 1 之時間）+10（工作 2 之服務時間）
3	5（等待工作 1 之時間）+10（等待工作 2 之時間） +4（工作 3 之服務時間）

則此種工作排程的方式所得到之總系統耗費時間為：

$$\underbrace{5}_{\text{工作 1 所耗的時間}} + \underbrace{(5+10)}_{\text{工作 2 所耗的時間}} + \underbrace{(5+10+4)}_{\text{工作 3 所耗的時間}} = 39$$

用同樣的計算方法對下列每一種可能的工作排程求出總系統耗費時間：

工作排程	總系統耗費時間
[1, 2, 3]	5 +(5 + 10) +(5 + 10 + 4) = 39
[1, 3, 2]	5 +(5 + 4) +(5 + 4 + 10) = 33
[2, 1, 3]	10 +(10 + 5) +(10 + 5 + 4) = 44
[2, 3, 1]	10 +(10 + 4) +(10 + 4 + 5) = 43
[3, 1, 2]	4 +(4 + 5) +(4 + 5 + 10) = 32
[3, 2, 1]	4 +(4 + 10) +(4 + 10 + 5) = 37

可得工作排程 [3, 1, 2] 有最短的總系統耗費時間 = 32。

- 上述考慮所有工作排程的可能性之演算法的結果與每項工作之耗費時間是有關的
- 此種排程法之所以會是最佳化的原因在於先把服務時間最短的工作做完。以下是這個過程的的高階貪婪演算法：

```
while （當此問題尚未得解） {
   執行下一項工作；                          // 選擇程序以及可行性檢查
   if （所有工作均已被執行完成）             // 解答檢查
       此問題得解；
}
```

4.3.2 依照截止期限來進行工作排程

- 在這種問題中，每項工作均需要花一單位的服務時間完成，且每項工作均有特定的截止期限以及其利潤。只要該項工作在截止期限以前或同時開始進行，就可以獲利
- 此問題的目標是找出一個可以獲得最大總利潤的工作排程。並非所有的工作都需要被排進工作排程中。我們不需要考慮任何包含了在截止期限後才開始進行的工作排程，因為不管有沒有被排入工作排程中，在截止期限後才開始進行的工作排程是不會得到任何的利潤。我們稱這種排程為**impossible**。
- 下面的範例描述了此種問題：

範例4.3

假設我們有如下的工作，截止期限以及其利潤：

工作	截止期限	利潤
1	2	30
2	1	35
3	2	25
4	1	40

當我們說工作1的截止期限為2的意思代表工作1可以在時間1或時間2開始進行。且在本例中沒有時間0。因為工作2的截止期限為1，所以代表了它只能在時間1開始進行。因此possible的工作排程以及其總利潤如下：

工作排程	總利潤
[1, 3]	30 + 25 = 55
[2, 1]	35 + 30 = 65
[2, 3]	35 + 25 = 60
[3, 1]	25 + 30 = 55
[4, 1]	40 + 30 = 70
[4, 3]	40 + 25 = 65

impossible 的工作排程並沒有被列出。舉個例子，在排程 [1, 2] 的中工作1必須在時間1開始進行，並且需要花一單位的時間去完成，造成工作2必須延後至時間2才能開始(其截止時間為時間1)，所以排程 [1, 2] 並不是 possible 的，因此沒有列出。但是排程 [1, 3] 中工作1在其截止時間前開始，而工作3在截止時間的同時開始，因此為 possible 的排程。我們可以發現排程 [4, 1] 為總獲利等於 70 的最佳化排程

定義

- 可行序列(feasible sequence)
  - 某序列的所有工作全都在截止期限前開始進行
    - 正例：範例4.3中的序列[4, 1]
    - 反例：[1,4]
- 可行集合 (feasible set)
  - 某工作集合中至少存在一個可行序列
    - 正例：範例4.3中，集合{1,4}
    - 反例：{2,4}，因為工作排程中不允許有同樣截止期限的兩者同時存在
- 最佳序列
  - 有最多總利潤的可行序列
- 最佳化工作集合

貪婪排程演算法

```
S = ∅;
while（當此問題尚未得解）{
   選擇下一項工作；                                    // 選擇程序
   if（加入選擇後的 S 仍為可行集合）                    // 可行性檢查
         將該工作加入 S 中；
   if（所有工作都已被選擇或檢查完畢）                   // 解答檢查
         此問題得解；
}
```

範例4.4

假設我們有如下的工作，截止期限以及其利潤：

工作	截止期限	利潤
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

我們已經將其按照利潤的順序排好，根據前面的貪婪演算法我們計算如下：

1.*S*設定為空集合。
2.*S*設定為{1}，因為序列[1]為可行序列。
3.*S*設定為{1, 2}，因為序列[2, 1]為可行序列。
4.因為在{1, 2, 3}中沒有可行序列因此不採用此集合。
5.*S*設定為{1, 2, 4 }，因為序列[2, 1, 4]為可行序列。
6.因為在{1, 2, 4, 5}中沒有可行序列因此不採用此集合。
7.因為在{1, 2, 4, 6}中沒有可行序列因此不採用此集合。
8.因為在{1, 2, 4, 7}中沒有可行序列因此不採用此集合。

最後的*S*為{1, 2, 4}，且得到的可行序列為[2, 1, 4]。因為工作1與4的截止期限均為3，因此我們可以用可行序列[2, 4, 1]替換之。

輔助定理 4.3

- 令S為所有工作組成的集合，則S為可行集合若且唯若將S中的工作依照截止期限大小的非遞減順序排列所得到之序列為可行序列。

證明：設S為可行集合，則在S中必存在至少一個可行序列。在此序列中，假設工作x被排定在工作y之前執行，而工作y的截止期限比工作x的截止期限小。如果我們將兩者在序列中的排定位置交換，則因為工作y必須更早開始進行，所以它仍然符合截止期限前起始的條件。此外因為工作x的截止期限大於工作y，故重新排列後工作x被分配到的執行時間區間仍不會與工作y相衝突，也就是工作x仍然符合截止期限前起始的條件。因此我們可以得知新的序列仍將是可行的。當我們在原來的可行序列上進行交換排序(演算法1.3)時，可重複地使用這個事實來證明這個已排序的序列是可行的。反之，若這個已排序的序列為可行的，S當然是可行的。

演算法4.4

依照截止期限來進行工作排程

- 問題：每項工作均有一個利潤值，找出一個有最大總利潤的工作排程的唯一方法是將這些工作根據其截止期限排序。
- 輸入：n 為工作的數目。還有一個用來存放為整數值的截止期限之陣列，其索引值由1至n，$deadline[i]$表示第i項工作的截止期限。此陣列必須根據每個工作利潤的非遞增順序排列。
- 輸出：此群工作的最佳序列 J。

```
void schedule (int n,
               const int deadline [],
               sequence_of_integer& J)
{
  index i;
  sequence_of_integer K;

  J = [1];
  for (i = 2; i <= n; i++){
     K = 根據 deadline[i] 值的非遞減順序將 i 加入所得之 J;
     if (K為可行的)
        J = K;
  }
}
```

範例4.6

假設我們有範例 4.4 中的工作，回想它們的截止期限如下：

工作	截止期限
1	3
2	1
3	1
4	3
5	1
6	3
7	2

根據前演算法 4.4 計算如下：

*J*設定為[1]。

*K*設定為[2, 1]且被判斷為可行的。

*J*設定為[2, 1]，因為*K*為可行的。

*K*設定為[2, 3, 1]，因為其不具可行性，故不採用。

*K*設定為[2, 1, 4]且被判斷其為可行的。

*J*設定為[2, 1, 4]，因為*K*為可行的。

*K*設定為[2, 5, 1, 4]，因為其不具可行性，故不採用。

*K*設定為[2, 1, 6, 4]，因為其不具可行性，故不採用。

*K*設定為[2, 7, 1, 4]，因為其不具可行性，故不採用。

最後的*J*為 [2, 1, 4]。

分析演算法4.4

最差情況的時間複雜度(依照截止期限來進行工作排程)

- 基本運算：比較運算
- 輸入大小：工作數目 n

在進行演算法的程序前需要花時間$\Theta(nlgn)$將工作依照順序排列完成。而在每輪執行for-i迴圈時，我們需要做最多$i-1$次的比較運算來將第i項工作加入K中，另外還要做最多$i-1$次的比較運算來檢查K是否可行。因此最差的情況是：

$$\sum_{i=2}^{n}[(i-1)+i]=n^2-1\in\Theta(n^2)$$

- 上述等式可由附錄A中的範例A.1中得到。因為總時間被排列順序的時間所支配，因此我們可以得到：

$$W(n)\in\Theta(n^2)$$

4.4 霍夫曼編碼

- 假設我們有一個字元集{*a*,*b*,*c*}，則我們可以將其中的每個字元以兩個位元來編碼。因為兩個位元可提供四個不同字碼的可能性，而實際上僅僅需要使用其中三個字碼。可編出下列的碼：

  a: 00　　b:01　　c:11

- 根據此編碼方式，若我們有一個檔案為

  ababcbbbc　　(4.1)

  可編碼為

  000100011101010111

- 可變動長度的二進位碼
  - 更高的編碼效率。這種編碼方式可以用不同的位元數來表示不同的字元。
  - 以上面的例子來說，我們可以把其中一個字元以0來表示。因為字元‘b’出現的頻率最高，因此我們把0這個字碼分配給‘b’則可以獲得最高的編碼效率。但是我們不能再將‘a’以‘00’來表示，因為我們將會無法分別‘00’到底是一個‘a’還是兩個‘b’。此外，我們也不能把‘a’編碼為‘01’，因為當遇到一個0的時候，若沒有看下面的位元，我們會無法判斷到底現在位元的0是代表‘b’還是‘a’的第一個位元。所以我們可依下列方式編碼：

    a: 10　　b:0　　c:11　　(4.2)

  - 根據此編碼方式，若檔案4.1可被編碼為

    1001001100011

4.4.1 前置碼

- 可變動長度二進位碼
  - 沒有任何一個字元所屬的字碼會與另外一個字元所屬的字碼之起始位元相同
    - 舉例來說，若 01是‘a’這個字元的字碼，則011就不能當作是‘b’這個字元所屬的字碼
    - 另外固定長度的二進位碼也是前置碼的一種。
  - 每一種前置碼均可以用二元樹來表示之，樹上的葉子就是要被編碼的字元。在圖 4.9 中可以看到與4.2的編碼相對應的二元樹
- 前置碼的優點是不需要檢查接下來的位元即可完成解碼。
- 在進行解碼時，我們由檔案最左邊的位元與二元樹的根部開始解碼。循序的檢查檔案中每一個位元，並同時在二元樹中根據該位元為0或1來決定在樹中的行進方向是該往右下還是左下走。當我們進行至一片二元樹中的葉子時，就表示我們已經解出該葉子代表的字元為何
- 接下來必須回到二元樹的根部，並開始檢查下面的位元，重複整個解碼流程。

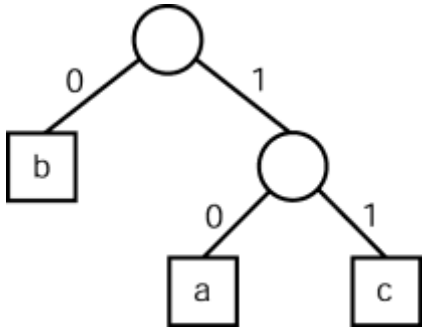


圖4.9 編碼4.2所對應的二元樹。

範例4.7

- 假設我們有一個字元集是{*a*, *b*, *c*, *d*, *e*, *f* }，每個字元在檔案中的出現次數列於表4.1中，同時表中也列出了三種不同編碼表供我們來將此檔案編碼。我們先計算每一種編碼將使用的位元數如下：
- *Bits*(C1) =16(3) + 5(3) + 12(3) + 17(3) + 10(3) + 25(3) = 255
- *Bits*(C2) = 16(2) + 5(5) + 12(4) + 17(3) + 10(5) + 25(1) = 231
- *Bits*(C3) = 16(2) + 5(4) + 12(3) + 17(2) + 10(4) + 25(2) = 212

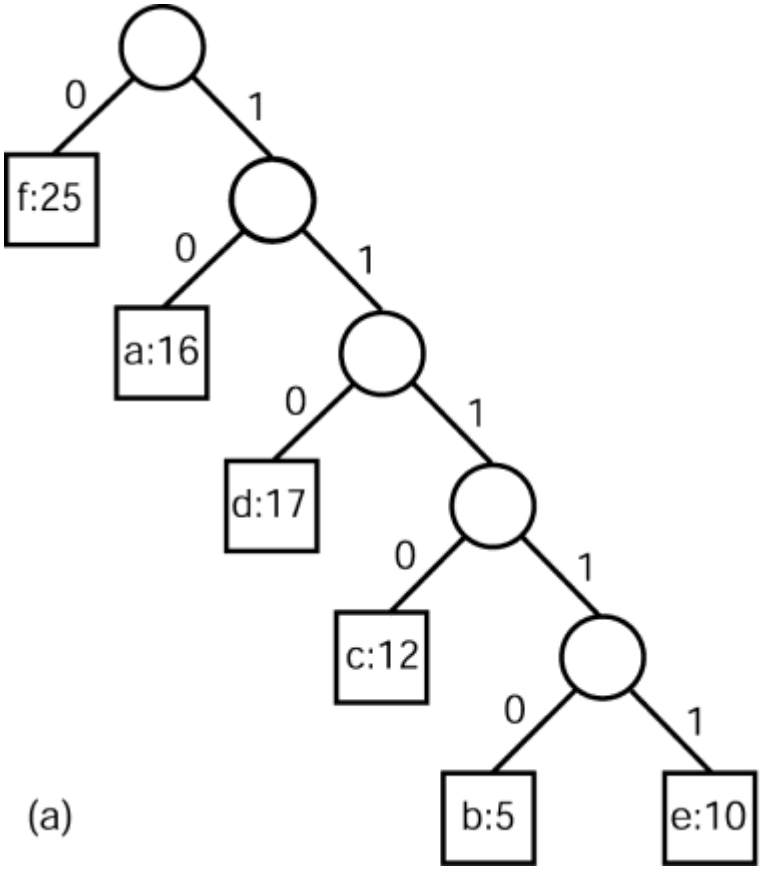
- 編碼表C2比固定長度的編碼表C1有更高之效率，但是C3 (霍夫曼編碼)卻比C2編碼還要好。而與C2與C3分別相對應的二元樹位於圖 4.10(a)與(b)。每一個字元被使用的頻率可以在對應的二元樹中看到。

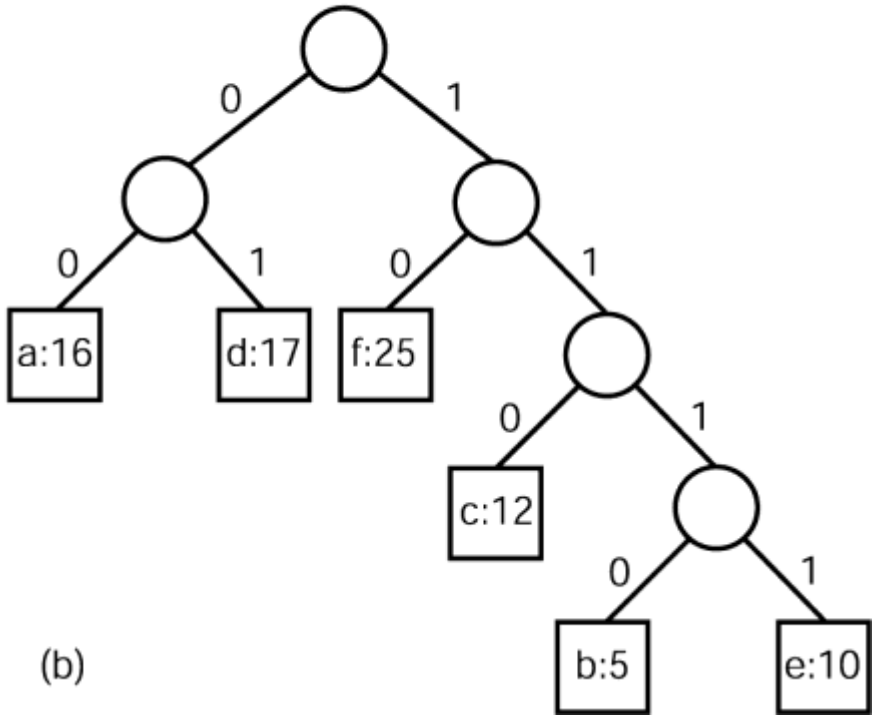
範例4.7

- 表4.1

字元	頻率	C1(固定長度)	C2	C3(Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

圖4.10 (a) 圖中所示為範例4.7中C2之二進位字元碼，而(b)圖中所示則為 C3(霍夫曼) 之二進位字元碼。





- 如同範例中所進行的計算程序，對某個檔案編碼所需用的位元數可由選定編碼的二元樹得到：

$$\text{位元數}(T) = \sum_{i=1}^{n} \text{頻率}(v_i)\text{深度}(v_i) \quad (4.3)$$

- 此處的$\{v_1, v_2, \ldots v_n\}$是此檔案的字元集，頻率(v_i)則表示字元v_i在檔案中出現的次數，而深度(v_i)則表示字元v_i在二元樹T中的深度。

4.4.2 霍夫曼演算法

- n = 檔案中的字元數;
- 將n個索引值指向記錄於優先順序列PQ的nodetype 如下：
- 對於PQ中的每個指標p

  p->symbol = 檔案中出現的一個字元

  p->frequency = 該字元於檔案中的出現頻率

  p->left = p->right = NULL;
- 其中優先順序端視出現頻率的數值大小，頻率越低則優先順序越高

```
for (i =1; i < = n-1; i ++) {                    //此處不檢查解
    remove (PQ, p);                              //解將於 i = n - 1. 時得到
    remove (PQ, q);                              //選擇程序
    r = new nodetype;                            //此處不做可能性檢查
    r->left = p;
    r->right = q;
    r->frequency = p->frequency + q->frequency;
    insert(PQ, r);
}
remove (PQ, r);
return r;
```

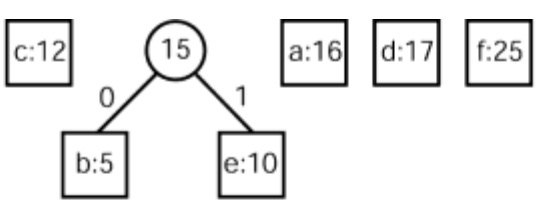
若以堆積實作優先順序列，則可在$\Theta(n)$次初始完成。此外，每個堆積運算需要$\Theta(lgn)$次。因為在for-i的迴圈中要執行$n-1$次，所以此演算法會執行 $\Theta(nlgn)$次。下面的範例說明了整個演算法的流程。

範例4.8

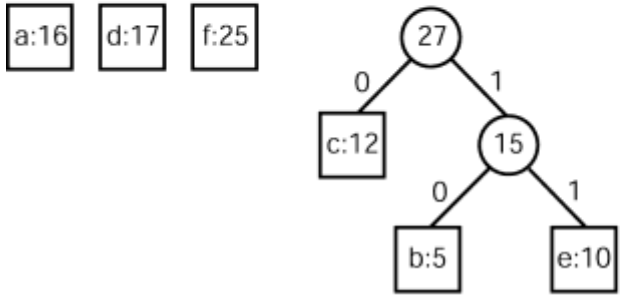
- 假設我們有一個字元集是{*a*, *b*, *c*, *d*, *e*, *f*}，每個字元在檔案中的出現次數列於表4.1中。圖4.11中可看到演算法中經過每次for-*i*迴圈運算後所建立出之中間解。圖中的第一組表示了在進入迴圈運算前之狀態。而在每個節點內的數值表示出現頻率。最後二元樹的結果請見圖4.10(b)。


b:5
e:10
c:12
a:16
d:17
f:25

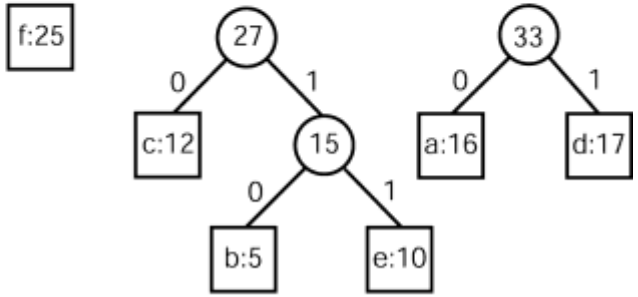
(0)


c:12
15
0
1
a:16
d:17
f:25
b:5
e:10

(1)

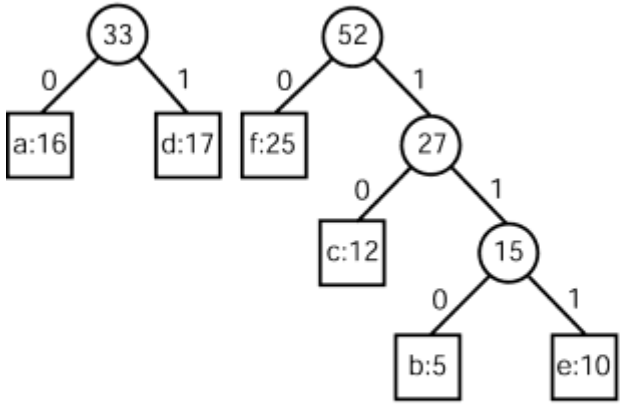

a:16
d:17
f:25
27
0
1
c:12
15
0
1
b:5
e:10

(2)

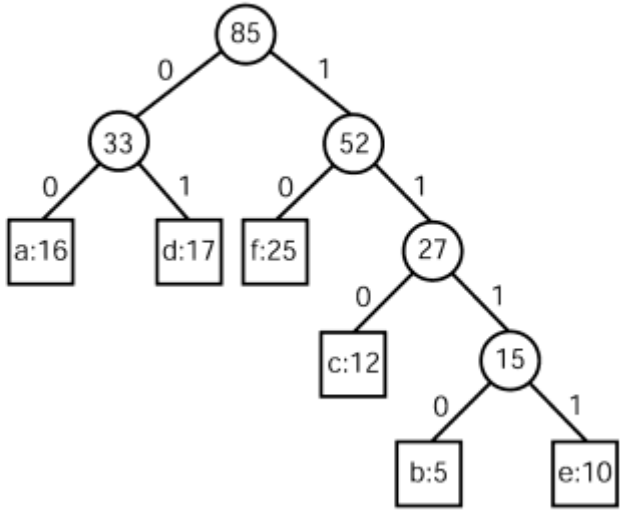

f:25
27
0
1
c:12
15
0
1
b:5
e:10
33
0
1
a:16
d:17

(3)

圖 4.11


33
0
1
a:16
d:17
52
0
1
f:25
27
0
1
c:12
15
0
1
b:5
e:10

(4)


85
0
1
33
52
0
1
0
1
a:16
d:17
f:25
27
0
1
c:12
15
0
1
b:5
e:10

(5)

4.5 貪婪演算法與Dynamic Programming之比較：背包(Knapsack)問題

- 貪婪演算法與dynamic programming是兩種解最佳化問題的方法
  - 單一起點最短路徑問題，可利用演算法3.3中dynamic programming得解，也可利用演算法4.3中的貪婪演算法得解
  - 但是在此問題中dynamic programming演算法的使用實屬過度，因為它可以找出所有起點的最短路徑。我們無法修改它以便有效率地找出單一起點的最短路徑，因為整個陣列D都需要被考慮進去。因此，對於此問題來說，dynamic programming為$\Theta(n^3)$的演算法，而貪婪演算法為$\Theta(n^2)$的演算法。通常用貪婪演算法來解問題是一個比較簡單且有效率的方案

4.5 貪婪演算法與Dynamic Programming之比較：背包(Knapsack)問題

- 另一方面，貪婪演算法較難得知是否總是可以得到最佳解。如同找錢問題中所顯示，並非所有的貪婪演算法均可以得到最佳解。我們需要針對特定某個貪婪演算法來證明其總是可得到最佳解，而在反例中我們仍得證明其無法總是得到最佳解。在dynamic programming的情況中我們僅需判斷其是否應用了最佳化原則
- 為了描述兩種演算法之間更詳細的不同，我們將提出兩個非常類似的問題：0-1背包問題與Fractional背包問題。我們將發展一個可以成功解出 Fractional背包問題但失敗於0-1背包問題的貪婪演算法。接著我們將用 dynamic programming 來成功解出0-1背包問題

4.5.1 用貪婪演算法來處理 0-1 背包問題

- 0-1背包問題
  – 使裝的物品總價值最大化同時不會超過背包的最大總載重W
- 假設有 n 個物品，令：

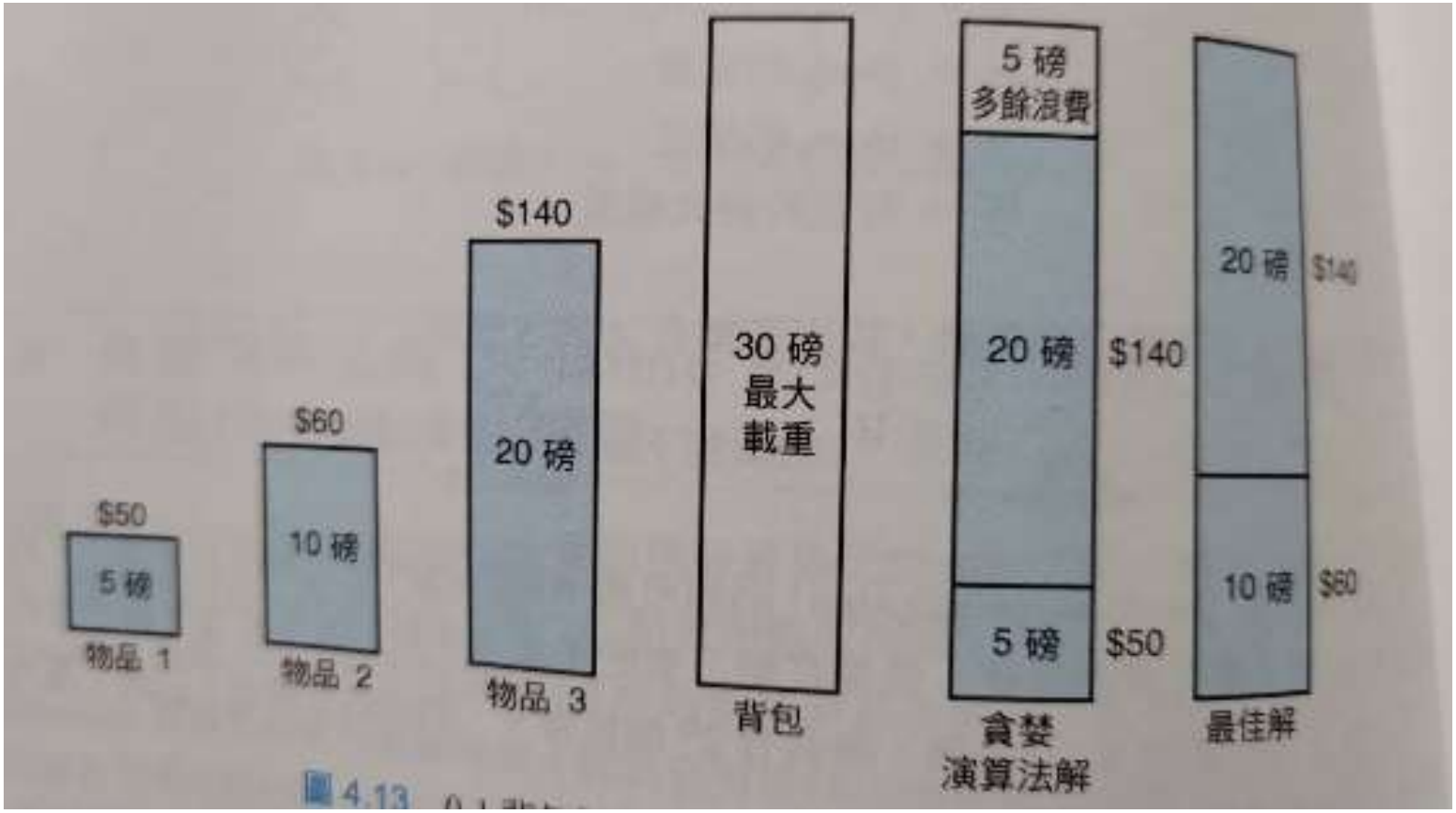
$$S = \{item_1, item_2, ..., item_n\}$$
$$w_i = item_i \text{的重量}$$
$$p_i = item_i \text{的價值}$$
$$W = \text{背包的最大載重}$$

- 其中，w_i、p_i、W均為正整數，找出子集合A與S使得：

在 $\sum_{item_i \in A} w_i \le W$ 的限制下， $\sum_{item_i \in A} p_i$ 的最大值

貪婪策略

- 從最高價值的物品開始
  - 但是在當最高價值的物品有很大的重量時，這個策略就運作的不是很好
- 由最輕的物品開始
  - 這個策略在最輕的物品的價值很低時也會運作不佳
- 自每一單位重量有最大價值的物品開始



item1 =$50/5 =$10, item2 =$60/10 =$6, item3 =$140/20 =$7

結論：Greedy Algorithm無法解決0-1背包問題！

4.5.2 用貪婪演算法來處理Fractional背包問題

- 在 Fractional背包問題中，竊賊不需要偷取整個物品，而是可以偷取部份物品的任何部份。
- 我們可以將0-1背包問題中的物品想像為金塊，銀磚之類，而Fractional問題中的物品則是像一袋金粉、銀粉之類的東西。
- **結論：Greedy Algorithm可以解決Fractional背包問題！**

4.5.3 用Dynamic Programming演算法來解0-1背包問題

- 若我們可以證明其符合最佳化原則，則我們可以使用 dynamic programming 來解決0-1背包問題
- 為了達到這個目標，令A為n個物品的最佳子集合，則會有以下兩種情況：不是A包含$item_n$，就是A不包含$item_n$。
  - 若A不包含 $item_n$，則A等於首n-1項物品的最佳化子集合。
  - 若A包含$item_n$，則A中物品的總利潤等於p_n加上由首n-1項物品中進行挑選所得到之最佳利潤，且挑選時遵守總重量不能達到W-w_n的限制。因此符合最佳化原則。

- 上述結果可以推斷如下：若$i > 0$及$w > 0$，我們設$P[i][w]$為遵守總重量不能達到w的限制下由首i項物品中進行挑選所得到之最佳利潤：

$$P[i][w] = \begin{cases} \text{最大值}(P[i-1][w] \text{ ，} p_i + P[i-1][w-w_i]) & \text{若 } w_i \le w \\ P[i-1][w] & \text{若 } w_i > w \end{cases}$$

- 最大利潤等於$P[n][W]$。我們可以用列由0到n，且行由0到W的二維陣列P來找出最大利潤值。我們用前面關$P[i][w]$的式子來按照順序來計算列的值。
- $P[0][w]$與$P[i][0]$均設為0。很直覺的我們可以知道該陣列所需計算的項目數量為：

$$nW \in \Theta(nW)$$

4.5.4 用修正的Dynamic Programming演算法來解 0-1 背包問題

- 4.5.3的演算法，對於一個給定的n來說，我們會因為任意增大的W而使得事件中的計算時間跟著無限增加。舉例來說，若W等於，則計算數目為$\Theta(n \times n!)$
- 此演算法經過改進後可以減少在最壞情況中所需的計算數目為，並且效率永遠不會比暴力法糟糕，通常還比暴力法好的多。改進的原因是因為在第i列中且w介於1與W之間的項目不需要被計算。更精確的說，在第n列中我們僅需要計算$P[n][W]$。為了計算$P[n][W]$，我們需要第$n-1$列中的某些項目，因為：

$$P[n][w] = \begin{cases} \text{最大值}(P[n-1][W]\ ,\ p_n + P[n-1][W-w_n]) & \text{若 } w_n \le W \\ P[n-1][W] & \text{若 } w_n > W \end{cases}$$

- 而第$n-1$列中的需要的項目為：

$$P[n-1][W] \text{ 與} P[n-1][W-w_n]$$

- 繼續從n往回找哪些項目是需要的。也就是說，當找出第i列中哪些項目是需要的，就可藉此找出第列中的需要的項目，原因如下：

  $P[i][W]$ 由 $P[i-1][W]$ 與 $P[i-1][W-w_i]$計算得出

- 在$n=1$或$w \leq 0$的時候計算停止。當找出需要的項目之後，我們從第一列開始進行計算。

範例4.9

P	0	10	20	30
0				
1				
2				
3				

若有圖4.13中的物品，且 $W = 30$。
首先找出每一列中所需要之項目。

找出第三列需要的項目：

我們需要

$$P[3][W] = P[3][30]$$

找出第二列需要的 entries：

為了計算$P[3][30]$，我們需要

$P[3-1][30] = P[2][30]$以及$P[3-1][30-w_3] = P[2][10]$

找出第一列需要的 entries：

為了計算$P[2][30]$，我們需要

$P[2-1][30] = P[1][30]$以及$P[2-1][30-w_2] = P[1][20]$

為了計算$P[2][10]$，我們需要

$P[2-1][10] = P[1][10]$以及$P[2-1][10-w_2] = P[1][0]$

接下來做以下的運算：

計算第一列：

$$P[1][w] = \begin{cases} \text{最大值} \quad (P[0][w], \$50 + P[0][w-5]) & \text{若} w_1 = 5 \le w \\ P[0][w] & \text{若} w_1 = 5 > w \end{cases}$$

$$= \begin{cases} \$50 \text{ 若 } w_1 = 5 \le w \\ \$50 \text{ 若} w_1 = 5 > w \end{cases}$$

因此，

$P[1][0] = \$0$
$P[1][10] = \$50$
$P[1][20] = \$50$
$P[1][30] = \$50$

計算第二列：

$$P[2][10]=\begin{cases}\text{最大值}(P[1][10]\ ,\ \$60+P[1][0]) & \text{若 } w_2=10\le 10\\ P[1][10] & \text{若 } w_2=10>10\end{cases}$$
$$=\$60$$

$$P[2][30]=\begin{cases}\text{最大值}(P[1][30]\ ,\ \$60+P[1][20]) & \text{若 } w_2=10\le 30\\ P[1][30] & \text{若 } w_2=10>30\end{cases}$$
$$=\$60+\$50=\$110$$

計算第三列：

$$P[3][30]=\begin{cases}\text{最大值}(P[2][30]\ ,\ \$140+P[2][10]) & \text{若 } w_1=20\le 30\\ P[2][30] & \text{若 } w_1=20>30\end{cases}$$
$$=\$140+\$60=\$200$$