Ch12. 最小生成樹 Minimal Spanning Tree

接下來進入最小生成樹的部分。

首先，會先說明最小生成樹的定義：什麼樣的樹可以被稱作一個「最小生成樹」，以及有什麼原理或準則可以用來找到一個最小生成樹。

最後，會介紹兩種主要用來產生最小生成樹的演算法：Kruskal's Algorithm 和 Prim's Algorithm。

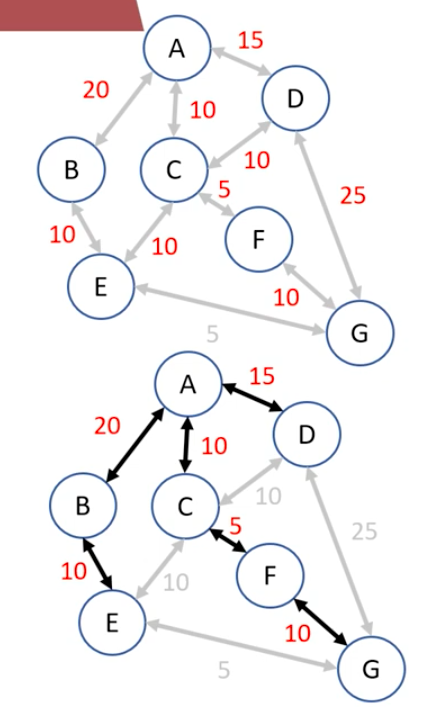
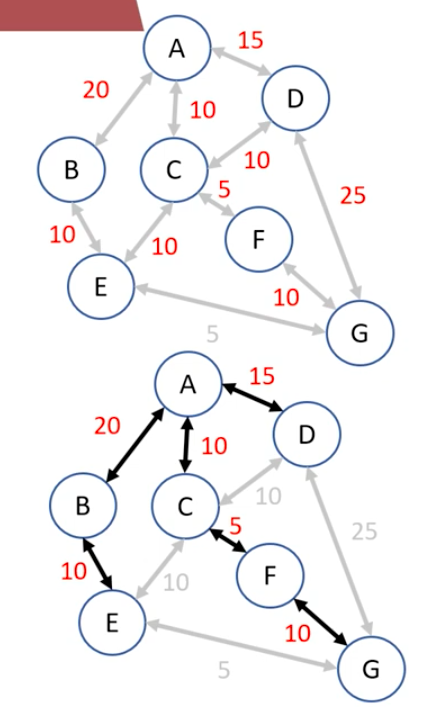
1. 最小生成樹定義與原理

在說明「最小生成樹」前，先探討一下什麼是「生成樹」。

1. 生成樹 Spanning Tree

生成樹的定義

1. 自圖中取出所有頂點與部分邊，使其形成一棵樹
2. 圖上的點能互相連通時，為「生成樹」
3. 圖上的點無法兩兩互相皆連通時，為「生成森林」
4. 生成樹有很多種可能
5. 通常以無向圖表示

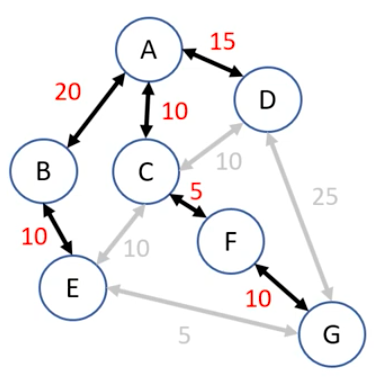


所謂生成樹就是取出圖中所有的頂點和部分的邊，讓它們可以形成一棵樹。像左上圖由頂點和邊所構成，取出「所有」頂點 A-G，再取出其中一些邊，如果這些邊使得所有頂點互相連通，那麼就叫做一個「生成樹」，如右上圖。

反之，如果選取的邊無法使所有頂點兩兩都互相連通，就叫做生成森林。

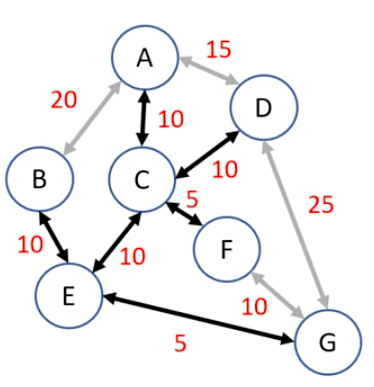
生成樹並不唯一，因為邊的組合可以任取。另外，通常是在無向圖上建立生成樹。因為生成樹也是一種「樹」，所以必須符合樹的性質，包括不能有環，以及若有 個頂點，則有 條邊。

（2）生成樹的權重

生成樹的權重即該生成樹中所有邊的權重總和。

上圖中，生成樹的權重是六條邊的權重和： 。由不同的邊組成的生成樹各有自己的權重，有大小之分。

（3）最小生成樹

「最小生成樹」指的就是一張圖所有可能的生成樹中，「權重最小」的那個，若有複數個權重最小者，都算作最小生成樹。

上圖的最小生成樹，權重為六個邊的權重和：。

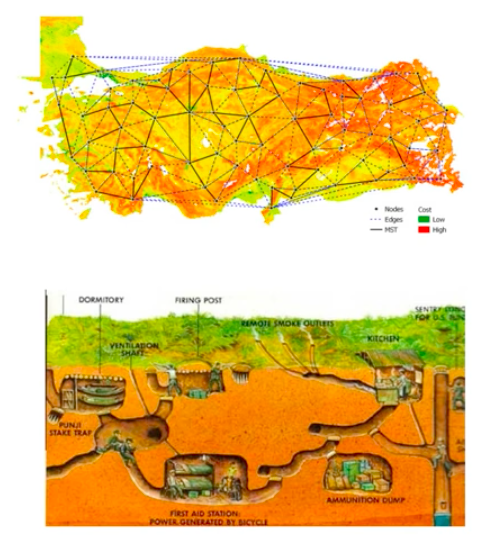
（4）最小生成樹的應用

最小生成樹有什麼功能呢？

把邊看作是成本，就可以拿來解決許多尋找「成本最小」方法的問題，看看如何連接每個頂點，可以使總成本最低。

比如「修路問題」：需要建造哪幾條路，才能在最低的成本下連接起所有城市。以及「礦井問題」：如何用最少的花費來打通所有礦井。

另外，網路通訊、水利工程等，都常應用到最小生成樹。



（5）最小生成樹的原理

想要得到一個最小生成樹，需要從圖中所有的邊裡選取 條邊（ 是頂點個數）。

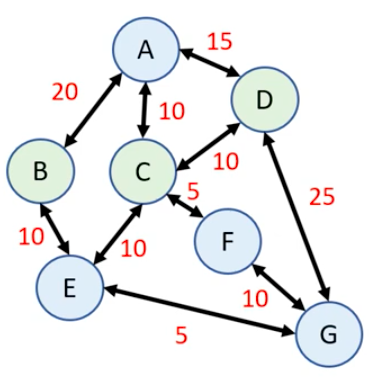
過程中，所有邊會被區分為兩個集合

* set A：屬於最小生成樹
* set B：不屬於最小生成樹

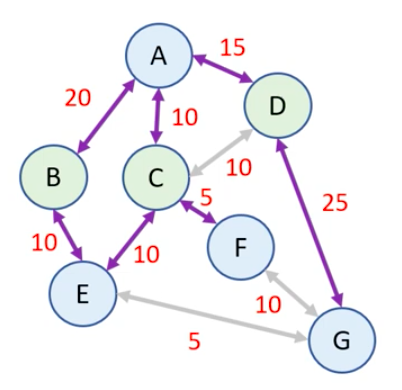
開始時，所有的邊都在 set B 中，進行過程中則從 set B 中逐步挑選邊放到 set A 裡。

如果某一次挑選了一條邊加入 set A 中後，set A 中的邊都仍屬於最小生成樹，那麼這次挑選過程被稱作是「安全」的。

（6）割 Cut

「割」是把圖中所有頂點分成兩個獨立集合，比如下圖中，所有頂點可以被分為 與 。

簡單來說，割就是把所有頂點任意分成「兩組」。

Crossing Edges 是一些可以「連接割產生的兩個集合」的邊。比如上面的例子裡，、、、、、、 這些邊都連接兩個集合中的各一個點，因此都屬於 Crossing Edges。

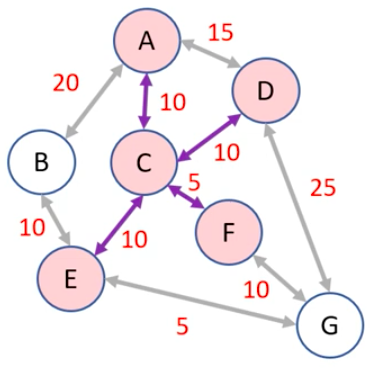
「割的權重」被定義為對應的所有 Crossing Edges 的權重和，上例中，權重為 。

如果一個特定的邊集合 A 中沒有任何一條邊屬於 Crossing Edges，比如 A 中所有的邊都連接割產生的兩個頂點集合 、 中 含有的頂點，那麼就說這個割「Respect」邊集合 A。比如上例的割 Respect 邊集合 （因為 和 都不是 Crossing Edges）。

Light Edge 是所有 Crossing Edges 中權重最小者，有複數條時，都屬於 Light Edge。

（7）最小生成樹的相關定理

已知圖 是有權重、連通的無向圖。



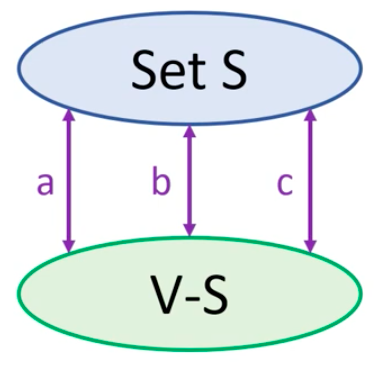
Set A 是當前（尚未完成）的最小生成樹中，「邊」的子集合（最小生成樹的邊的其中一些），比如上圖中，可以是 、、、。

Set S 是當前（尚未完成）的最小生成樹中，「頂點」的子集合（其中一些頂點），比如上圖中的 A、C、D、E、F。

如果 Set A 中的所有邊都只連接 Set S 裡的兩個頂點（比如上例中，已經取的四條邊只連接 A、C、D、E、F），即 repect Set A，那麼此敘述成立：把一個 light edge 邊 e 從 set B 放入 set A 中的過程是「安全」的。

也就是說，如果現在選到的邊集合 Set A 和頂點集合 Set S 都是某個最小生成樹的其中一部分，那麼觀察剩下的邊，可以向外連接到尚未連進來的頂點的所有邊裡（比如上圖中的 、、、、），權重最小的那個邊也會屬於這個最小生成樹。

簡單證明上面的定理，把 和 看成尚未完成的最小生成樹的兩個部分，且兩個部分的權重分別為 、， 和 兩部分各自都是連通的（把一棵樹的其中一條邊切斷，一定會形成兩個各自連通的部分）。



和 間有數條 crossing edges a、b、c，其中 a 是 light edge。

因為最小生成樹是一種「樹」，任兩點間必定有路徑連通，要選擇一條邊讓 Set 可以向外連到 裡的頂點，使得兩個各自連通的樹變成一顆完整的最小生成樹，就必須選擇 a、b、c 其中一條邊，最多也只能選擇其中一條（否則就會形成環）。

那麼既然選擇任一條 crossing edges 都能把 和 連接起來變成一顆完整的最小生成樹，那麼顯然必須選擇其中的 light edge 邊 a，得到的生成樹權重才會最小，為 。

根據上面的定理，只要一直選擇當前所有的 crossing edges 中權重最小者加入生成樹，就能得到一個「最小生成樹」，本章介紹的 Kruskal 和 Prim 演算法都運用了這個特性。

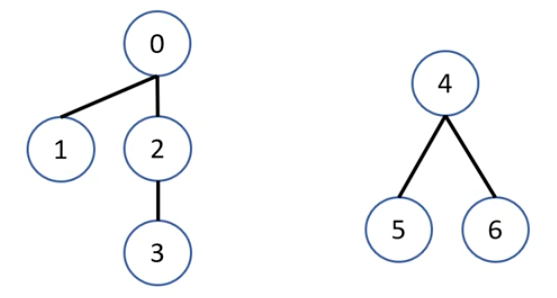
因為進行選擇時，都去找當下權重最小的 crossing edge，因此本質上這是一種「貪婪演算法」。

1. 集合的搜尋與合併

在進入 Kruskal 演算法的介紹前，需要先思考如何在資料結構裡有效率的記錄屬於不同集合的頂點，以及如何讓不同集合可以被合併在一起。

剛才講到要不斷選取 crossing edge 來連接不同割分成的兩個集合，那麼必須要有一個資料結構來存放當前每個頂點從屬的集合。

（1）Set



習慣上，把每個集合各自看成一棵樹，比如上圖中，頂點 0、1、2、3 屬於一個集合，頂點 4、5、6 屬於另一個集合，0、1、2、3 組成了一個以頂點 0 為根節點的樹，4、5、6 則組成一個以頂點 4 為根節點的樹。



這樣一來，就可以用一個陣列（取名叫 set）來存放圖中每個頂點所屬集合的資訊。當 時， 的值指的是該頂點 往根節點方向走的相鄰頂點，比如 的值是 2，代表頂點 3 接在頂點 2 下面。

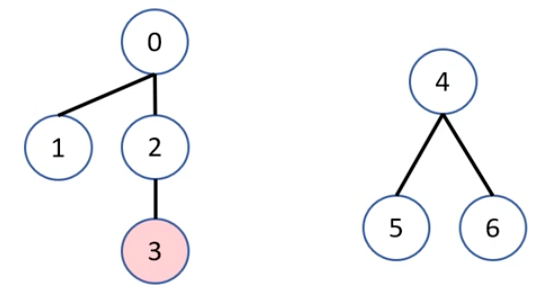
只有根節點 0 和 4 的 值才會小於 0，當 時， 代表的是該集合的總頂點數，上例中， 代表以 0 為根節點的集合共有 4 個頂點， 代表以 4 為根節點的集合共有 3 個頂點。

根據陣列 set 的資訊，就可以畫出對應的頂點集合。

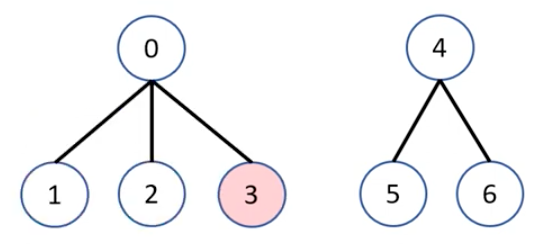
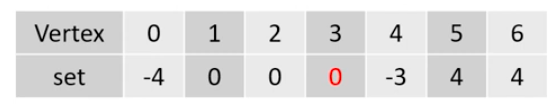
（2）Find\_Set

要如何找到某個頂點 屬於哪個集合呢？只要從該頂點出發，一路往根節點方向移動，直到 ，確定找到根節點後回傳即可。

|  |  |
| --- | --- |
| Find\_Set | |
| 1  2  3  4  5  6 | Find\_Set(set, v){  while (set[v] >= 0)  // 讓 v 向上移動  v = set[v]  return v  } |

（3）Collapsing

為了增進尋找根節點（集合編號）的效能，希望樹高被限制為 1，也就是說，所有屬於該集合的頂點都直接被連接在根節點上，使得之後都只需經過 1 次搜尋就可知道該點屬於哪個集合。

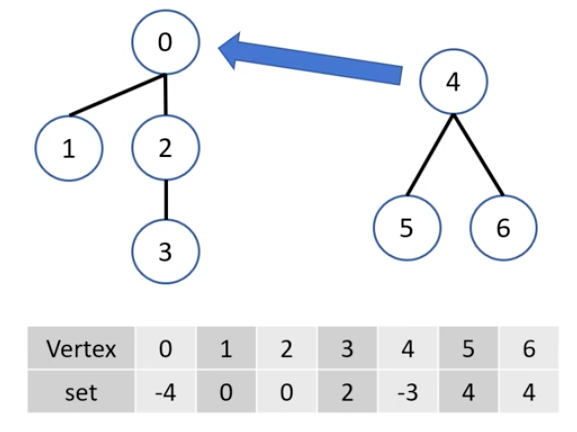


完成 Collapsing 後，頂點 3 直接連到根節點 0 上，以 0 為根節點的樹高變為 1。

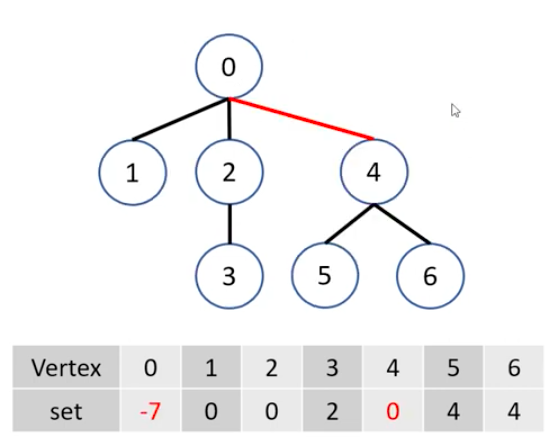
具體的作法是從頂點 出發往根節點 移動的過程中，把中間經過的所有頂點（包括一開始的頂點 ）的 值都改為根節點的值。

|  |  |
| --- | --- |
| Find\_Set\_Collapsing | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | Find\_Set\_Collapsing(set, v){  // 先從 v 開始往上找到根節點  // 跳出迴圈時，root 就是根節點  root = v  while (set[root] >= 0)  root = set[root]  // 重新從 v 開始往上  // 把經過的每個頂點 set 值都寫成 root  while (v!=root)  // 先記錄下 predecessor 的值  predecessor = set[v]  // 把 set[v] 改為根節點 root  set[v] = root  // v 往 predecessor 方向移動  v = predecessor  // 回傳根節點  return root  } |

（4）Merge\_Set

再來，要如何合併兩個集合呢？其實相當簡單，只要把其中一個集合的 指向另一個集合的 即可。

但是怎麼決定誰應該從屬於誰？因為「通常」頂點數越多的 樹高越大（但不一定），所以為了搜尋時的效能考量，會讓頂點個數少的 從屬於頂點個數多的 ，上例中，就是讓 s 從屬於 。



另外，還要記得把合併後 的值更新以對應兩個樹合併後的總頂點數。

|  |  |
| --- | --- |
| Merge\_Set | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Merge\_Set(set, u, v){  // 找到 u,v 兩個集合的根節點  u\_root = find\_set\_collapsing(set, u)  v\_root = find\_set\_collapsing(set, v)  // 如果 u 的頂點數比較多（絕對值較大）  if (set[u\_root] <= set[v\_root])  // 加總頂點數  set[u\_root] += set[v\_root]  // v 的根節點接到 u 上  set[v\_root] = u\_root  // 如果 v 的頂點數比較多  else  // 加總頂點數  set[v\_root] += set[r\_root]  // u 的根節點接到 v 上  set[u\_root] = v\_root  } |

（5）實作 Find\_Set 與 Merge\_Set

在之前寫過的 Graph 中，先寫好 Find\_Set 與 Merge\_Set 兩個函式備用：

|  |  |
| --- | --- |
| Find\_Set & Merge\_Set | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | #include<iostream>  #include<stdlib.h>  #include<vector>  #include<list>  using namespace std;  typedef struct{  // !注意要加上 from  int from;  int to;  int weight;  }edge;  class Graph{  private:  // 記錄頂點從屬的集合  vector<int> MST\_Set;  int vertex;  vector<list<edge\*>> edges;  // 找到一個頂點從屬的集合  int Find\_Set(int);  // 合併兩個集合  void Merge\_Set(int, int);  public:  Graph(int);  void print\_edges();  void add\_edge(int, int, int=1);  };  // 調整建構式  Graph::Graph(int v){  vertex = v;  edges.resize(vertex);  MST\_Set.resize(vertex); // 初始化設定 MST\_Set  }  // 使用無向圖，加上 from  void Graph::add\_edge(int from, int to, int weight){  edges[from-1].push\_back(new edge[from-1, to-1, weight]);  edges[to-1].push\_back(new edge[to-1, from-1, weight]);  }  int Graph::Find\_Set(int u){  int root = u;  while (MST\_Set[root] >= 0){  root = MST\_Set[root];  }  while (u!=root){  int predecessor = MST\_Set[u];  MST\_Set[u] = root;  u = predecessor;  }  return root;  }  void Graph::Merge\_Set(int u, int v){  int u\_root = Find\_Set(u);  int v\_root = Find\_Set(v);  if (MST\_Set[u] <= MST\_Set[v]){  // u 的頂點數比較多時  MST\_Set[u\_root] += MST\_Set[v\_root];  MST\_Set[v\_root] = u\_root;  } else {  // v 的頂點數比較多時  MST\_Set[v\_root] += MST\_Set[u\_root];  MST\_Set[u\_root] = v\_root;  }  }  // 其他函式內容 ... |

1. Kruskal 演算法

本節正式介紹 Kruskal 演算法。

1. Kruskal 演算法的概念

開始時，Kruskal 演算法把圖上的每個頂點都視為一個獨立的集合，也就是說，頂點 各自都是只含一個頂點的樹。

接下來，會逐步挑選適合的邊，以把這些獨立的集合合併，直到所有頂點都屬於同一個集合。

每次挑選的邊必定是一個 crossing edge，因為集合內部的頂點已經是連通的了，不需要再多加一個邊，要加上一個邊的目的一定是要把其它集合合併進來。

再來，為了符合「最小」的性質，挑選的邊會是目前所有 crossing edge 中權重最小者。

總合來說，只要先將邊的權重由小到大排序，然後從權重最小的邊開始一路檢查每個邊目前是否還是 crossing edge，若是，就加入 MST 所需的邊中，不是則跳過，直到選取的邊數為「頂點數- 1」，即所有頂點都被連接時就得到「最小生成樹」了。

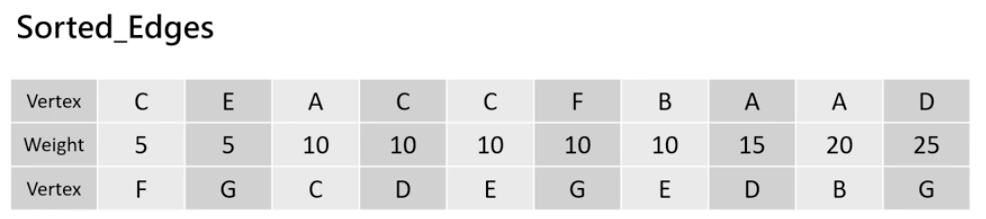
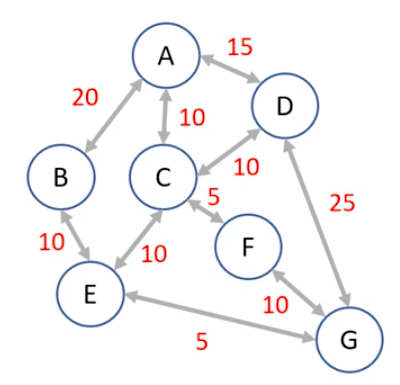
具體來說，需要三個變數：

1. MST\_Edges：記錄所有已挑選進最小生成樹的邊，即先前提過的 set A
2. MST\_Set：記錄目前每個頂點所屬的集合
3. Sorted\_Edges：把原圖中所有的邊依照權重由小到大排列

（2）變數的初始化



一開始， 中沒有邊，因此是空的。



中每個點對應到的值都是 -1，代表只含自己一個頂點的集合（前面提過若值 < 0，代表本身是根節點，該負值的絕對值是集合中的頂點數目）。

中是各邊連接的兩個頂點與權重，且邊之間按照權重由小到大排列。

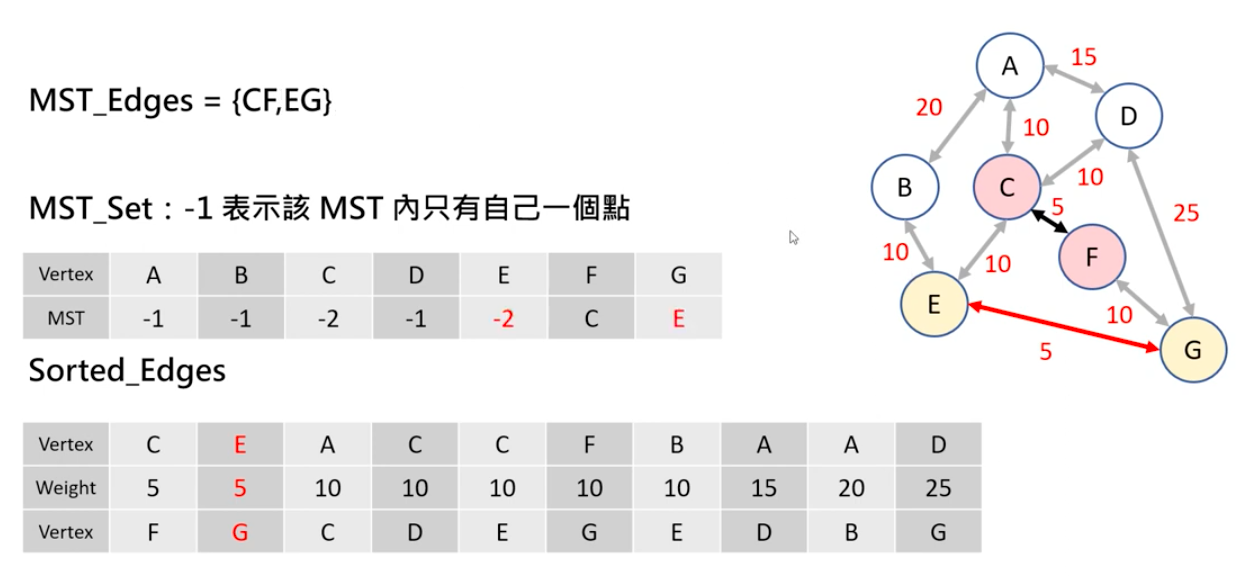
（3）Kruskal 演算法的進行過程

一開始，從 中取出權重最小的邊來檢查，發現 跨越了兩個不同的集合，是一個 crossing edge，可以把這個邊加入 中。

隨後，要用剛才寫過的 Merge\_Set 函式把 C 和 F 兩個集合合併在一起，合併後， 裡 C 的值變為 -2，代表這個集合有兩個頂點（C 和 F），F 對應的值則變為 C，代表 F 現在屬於以 C 為根節點的集合。

再來，檢查其餘邊中權重最小的 是不是 crossing edge，只要透過剛才的 Find\_Set 函式，看兩個邊所屬的集合是否相同就可以得知。

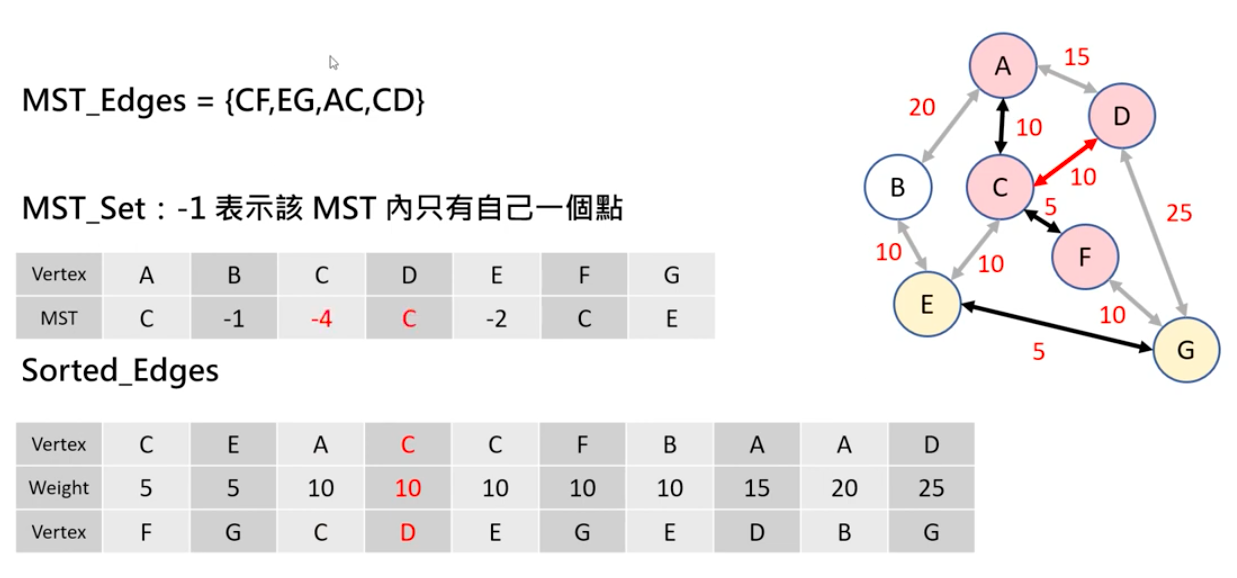
因為 也是一個 crossing edge，把它加入 中，並且合併 E 和 G 所屬的集合。



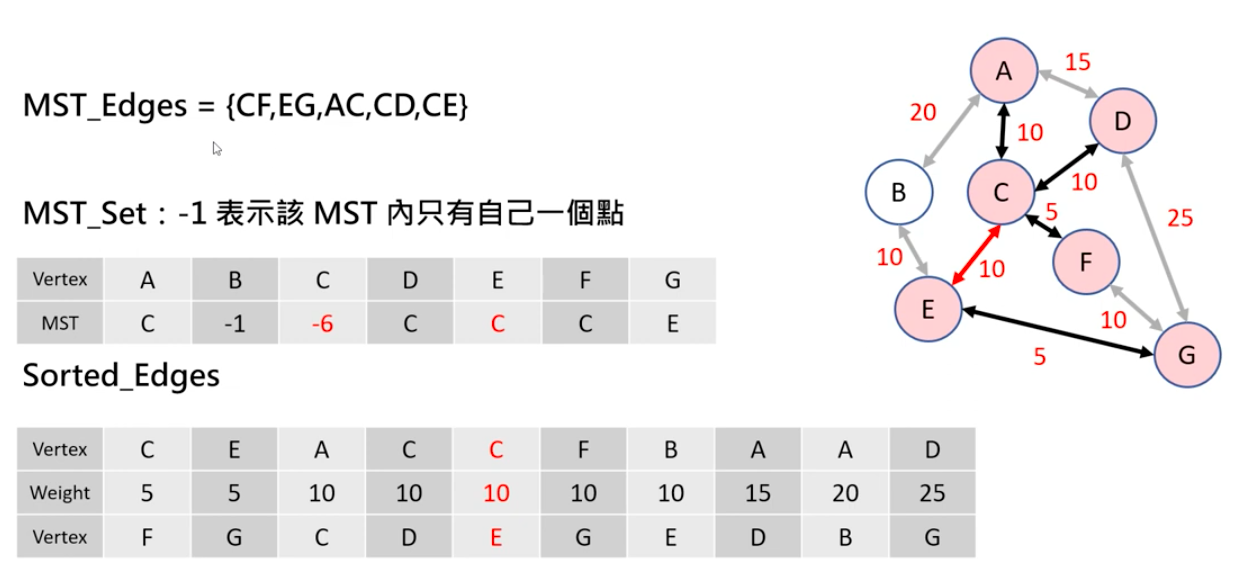
接下來，對 做處理，因為 是 crossing edge，所以同樣把 加入 中。接下來，合併兩個集合，因為 C 所屬集合含有的頂點個數比 A 所屬的集合多，因此是把 A 合併進 C 所屬的集合內。



再來，對 做一樣的處理。

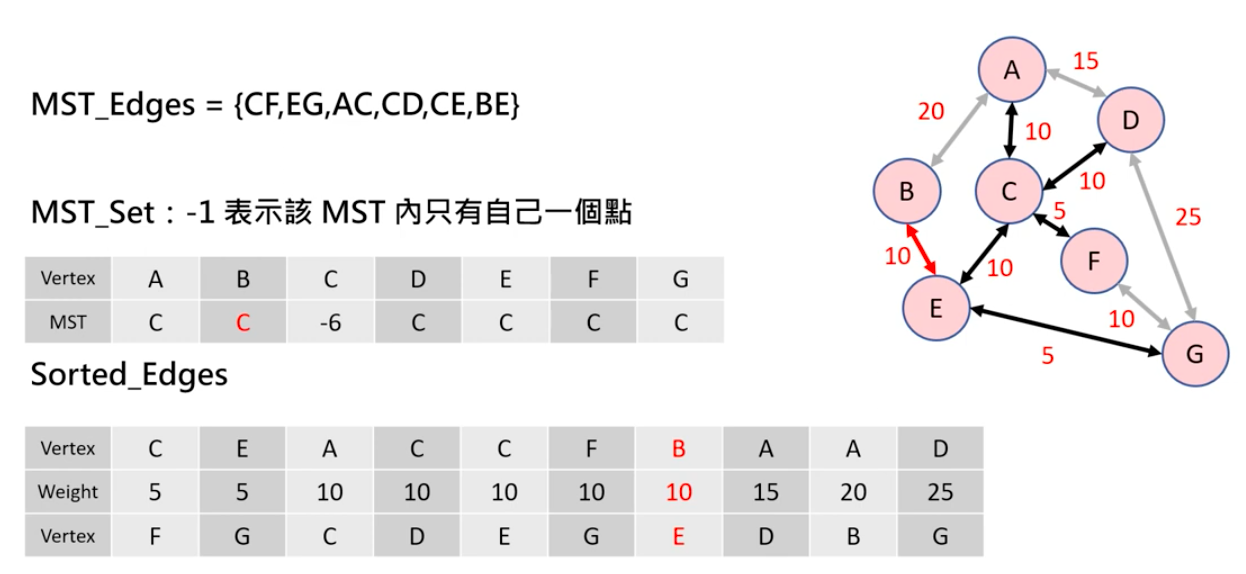


再對 做一樣的處理。



接下來，檢查到 ，因為頂點 F 和頂點 G 透過 Find\_Set 函式找到的集合編號都是 C（、），所以 此時不是 crossing edge，必須跳過，同時，透過 collapsing 的過程把 中 G 對應的值改成 C。

檢查到 ，同樣加入 中，並進行融合。



加入 後， 中已經含有六個邊，也就是一個七個頂點的圖的「最小生成樹」所需的邊數了，因此可以結束 Kruskal 演算法的處理，輸出最小生成樹中的所有邊。

實際上，剩下的幾個邊 、、 此時都已經不是 crossing edge，若再加入 中，必定會形成環。

而且因為同樣權重的邊之間，在排序時不一定誰在前誰在後，因此得到的最小生成樹的解並不唯一。

（4）Kruskal 演算法的複雜度

|  |  |
| --- | --- |
| Kruskal 演算法的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | Kruskal(G,V,E){  // 已加入最小生成樹的邊數  edges\_completed = 0  MST\_Edges = array of edge  // priority\_queue 中存放的邊有順序  Sorted\_Edges = priority\_queue  // 初始化 MST\_Set  for each v in V:  MST\_Set[v] = -1  // 初始化 Sorted\_Edges  for each e in E:  Sorted\_Edges.push(e)  for each e(u,v) in Sorted\_Edges:  // 如果是 crossing edge  if (find\_set\_collapsing(u) != find\_set\_collapsing(v)):  // 把邊 e 加入 MST\_Edges 中  MST\_Edges[edges\_completed++] = e  // 合併兩個集合  merge\_set(set,u,v)  return MST\_edges  } |

初始化：

對每個邊做處理：

處理一個邊所需：

總和：

又 ;

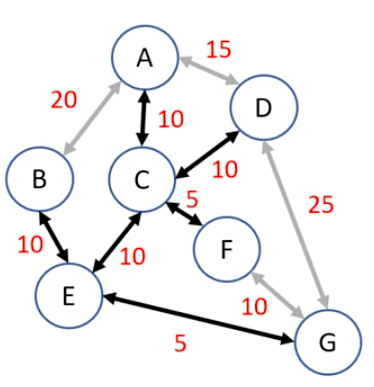
把 中每個頂點值初始化為 -1 的過程複雜度為 ，而根據 priority\_queue 的結構特性，把每個邊都加入其中的複雜度為 。

完成初始化後，還要對每個邊做處理，最多可能需要處理全部 個邊，而對每個邊做的處理所需複雜度平均不超過 ，因此這個迴圈的總複雜度也是 。

總和來說，整個演算法的複雜度為 ，或者因為 和 的關係，也可以表示為 。

（5）實作 Kruskal 演算法

|  |  |
| --- | --- |
| Kruskal’s Algorithm | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124 | #include<iostream>  #include<stdlib.h>  #include<vector>  #include<list>  #include<queue> // for priority\_queue  using namespace std;  typedef struct{  // !注意要加上 from  int from;  int to;  int weight;  }edge;  class Graph{  private:  // 記錄最小生成樹中的邊  vector<edge> MST\_Edges;  // 記錄頂點從屬的集合  vector<int> MST\_Set;  int vertex;  vector<list<edge\*>> edges;  int Find\_Set(int);  void Merge\_Set(int, int);  public:  Graph(int);  // Kruskal 函式  void MST\_Kruskal();  void print\_edges();  void add\_edge(int, int, int=1);  };  // 自訂邊的排序函式  class compare{  public:  bool operator()(edge &e1, edge &e2){  // 依照邊的權重排序  // 根據 priority\_queue 的規則，這種寫法會讓 top() 拿到  // 權重較小的邊  return e1.weight > e2.weight;  }  };  void MST\_Kruskal(){  int edges\_completed = 0;  MST\_Edges.clear();  // 根據權重把邊排序  priority\_queue<edge, vector<edge>, compare> sorted\_edges;    // 初始化  for (int i=0 ; i<vertex ; i++){  MST\_Set[i] = -1;  // 把邊加入 priority\_queue  for (auto iter = edges[i].begin() ; iter!=edges[i].end() ; iter++){  // 注意要使用 \*\*iter 來取出邊  // iter 本身的類型是 list<edge\*>::iterator  // iter: iterator  // \*iter: edge\*  // \*\*iter: edge  // 把邊 push 進 queue 中  sorted\_edges.push(\*\*iter);  }  }    // 處理完所有邊，或者取的邊數已經到達頂點數-1時結束迴圈  while (!sorted\_edges.empty() && edges\_completed < vertex-1){  edge current = sorted\_edges.top();  sorted\_edges.pop();  if (Find\_Set(current.from) != Find\_Set(current.to)){  Merge\_Set(current.from, current.to);  MST\_Edges.push\_back(current);  edges\_completed++;  }  }  // 算出最小生成樹權重  int sum = 0;  for (int i=0 ; i<vertex ; i++){  // 輸出邊的資訊  cout << (char)(MST\_Edges[i].from+65) << "->" <<  (char)(MST\_Edges[i].to+65) << endl;  sum += MST\_Edges[i].weight;  }  cout << "Weight of this MST = " << sum << endl;  }  // 使用無向圖，加上 from  void Graph::add\_edge(int from, int to, int weight){  edges[from-1].push\_back(new edge[from-1, to-1, weight]);  edges[to-1].push\_back(new edge[to-1, from-1, weight]);  }  int Graph::Find\_Set(int u){ ... }  void Graph::Merge\_Set(int u, int v){ ... }  // 其他函式內容 ...  int main(){  Graph g(7);  g.Add\_Edge(1,2,20);  g.Add\_Edge(1,3,10);  g.Add\_Edge(1,4,15);  g.Add\_Edge(2,5,10);  g.Add\_Edge(3,5,10);  g.Add\_Edge(3,4,10);  g.Add\_Edge(3,6,5);  g.Add\_Edge(4,7,25);  g.Add\_Edge(6,7,10);  g.Add\_Edge(5,7,5);  g.MST\_Kruskal();  return 0;  } |
| 執行結果 | |
| C->F  E->G  C->A  C->E  E->B  D->C  Weight of this MST = 50 | |



如上圖所示，得到的最小生成樹中六個邊的權重總和為 50。

（6）LeetCode#1584. Min Cost to Connect All Points

A. 題目

給定一個陣列 ，代表了二維平面上一些頂點的整數座標，。

連接兩個頂點 和 的成本設定為 ，也就是 座標相減的絕對值與 座標相減的絕對值之和。

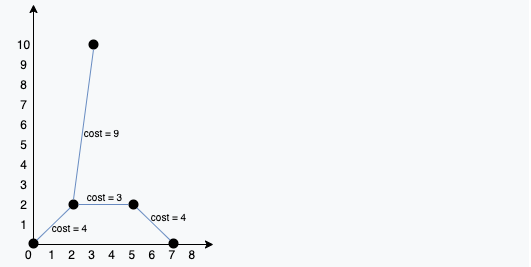
回傳要連接所有頂點的最小成本。

B. 出處

https://leetcode.com/problems/min-cost-to-connect-all-points/

C. 範例輸入與輸出

輸入：points = [[0,0],[2,2],[3,10],[5,2],[7,0]]

輸出：20

D. 解題邏輯

本題根據設定的成本公式計算出每個邊的權重後，應用 Kruskal 演算法得到最小生成樹的權重即可。

|  |  |
| --- | --- |
| Min Cost to Connect All Points [Kruskal] | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96 | // 邊的結構  struct edge{  int from;  int to;  int cost;  };  // 邊的排序函式  class compare{  public:  bool operator()(edge &e1, edge &e2){  return e1.cost > e2.cost;  }  };  class Solution{  vector<int> MST\_set;  public:    // Find\_Set  int find\_set(int current){  int root = current;  while (MST\_set[root]>=0){  root = MST\_set[root];  }  while (current!=root){  int predecessor = MST\_set[current];  MST\_set[current] = root;  current = predecessor;  }  return root;  }  // Merge\_Set  void merge\_set(int u, int v){  int u\_root = find\_set(u);  int v\_root = find\_set(v);  if (u\_root == v\_root){ return ; }  if (MST\_set[u\_root] < MST\_set[v\_root]){  MST\_set[u\_root] += MST\_set[v\_root];  MST\_set[v\_root] = u\_root;  } else {  MST\_set[v\_root] += MST\_set[u\_root];  MST\_set[u\_root] = v\_root;  }  }  int minCostConnectPoints(vector<vector<int>>& points){    // 已加入的邊數  int completed = 0;  // 所需總成本  int total\_cost = 0;  int vertex = points.size();  MST\_set.resize(vertex);  priority\_queue<edge, vector<edge>, compare> sorted\_edges;  // Push and initialize  for (int i=0 ; i<vertex ; i++){  MST\_set[i] = -1;  // 在任兩個點間都加上一條邊edge(i, j)  for (int j=i+1 ; j<vertex ; j++){  // 計算 cost  int current\_cost = abs(points[i][0]-points[j][0]) +  abs(points[i][1]-points[j][1]);  edge current = {i, j, current\_cost};  sorted\_edges.push(current);    } // end of inner for  } // end of outer for  while (!sorted\_edges.empty() && completed < vertex-1){  edge current = sorted\_edges.top();  sorted\_edges.pop();  if (find\_set(current.from) != find\_set(current.to)){  merge\_set(current.from, current.to);  // 要加入邊時，把該邊成本加入總成本中  total\_cost += current.cost;  // 更新目前加入邊數  completed++;  }  }  // 回傳最小成本  return total\_cost;  } // end of minCostConnectPoints  }; // end of Solution |

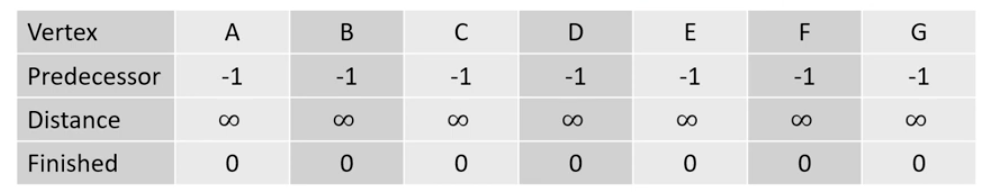
1. Prim 演算法

本節介紹另一種用來得到最小生成樹的演算法：Prim's Algorithm。

Kruskal 演算法一開始是把所有的頂點都看成獨立的集合，並且逐一挑選邊來把集合合併。

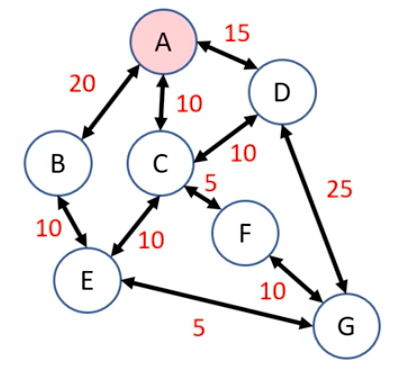
Prim 演算法則是從一個任選的點開始逐漸挑選其他「頂點」來加入集合，挑選的原則是去找距離正在形成的最小生成樹「最近」的頂點，逐一把頂點加入，直到完成整個最小生成樹，進行過程很像圖論章節中找關鍵路徑的演算法。

（1）實作 Prim's Algorithm 的變數

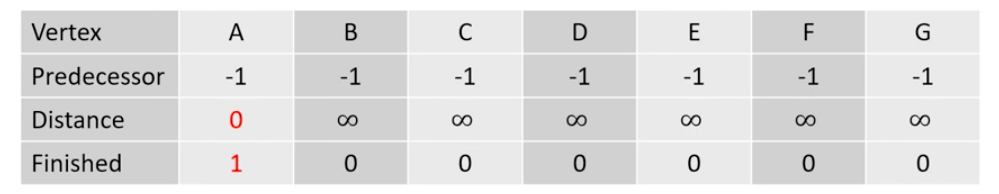


1. ：每個頂點往 MST 移動的最短路徑上，與其相鄰的那個頂點（類似父節點），用來記錄 MST 中有哪些邊。一開始最短路徑未知，初始化為 -1
2. ：目前該節點到 MST 最短的距離，初始值為
3. ：記錄哪些頂點已經加入 MST 中，初始值為 false / 0

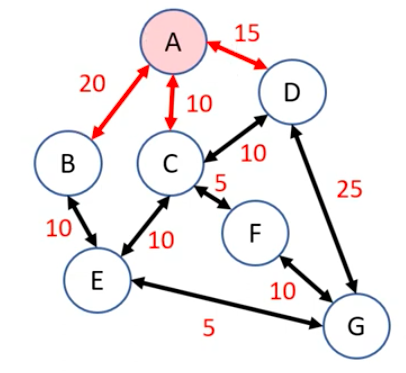
（2）Prim 演算法的進行過程



一開始，任意選擇頂點 A 加入 MST，A 的角色類似根節點，因此 值維持 -1，且因為已經在 MST 中，與 MST 的距離 值設為 0，另外， 值設為 1 代表此頂點已處理。



根據 A 點的全部三條出邊，更新 B、C、D 這三個相鄰點與 MST 的距離，如果從 A 走到這幾個相鄰頂點的距離，比原本的距離（初始值 ）來的短，那就把該值更新。

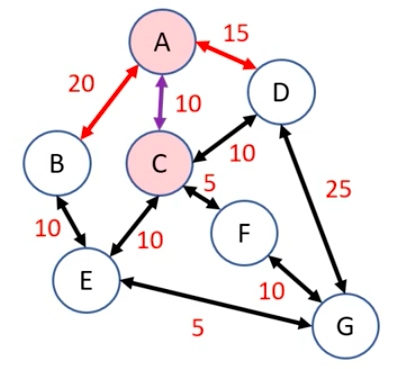
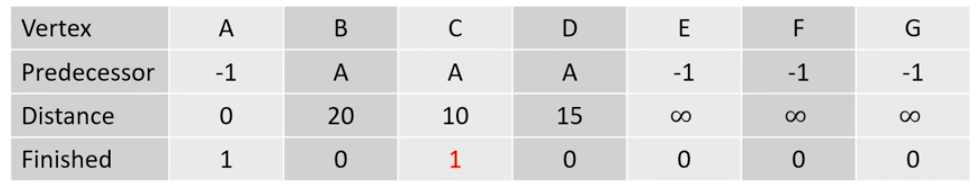


比如從 MST 中的頂點 A 只要透過 就可以走到頂點 B，因此 被更新為 的權重 20，類似的，C 和 D 點的 被更新為 10 和 15，過程中 B、C、D 點的 都設為 A。



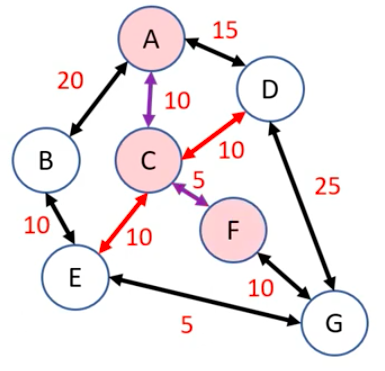
這樣一來對 A 點的處理就完成了，挑選此時「未完成」（ 值為 0）的頂點中 最小者作為下一個處理的頂點，此時候選頂點中以 最小，因此接續處理頂點 C。

同樣的，把頂點 C 加入 MST，再來，將其 值設為 1。



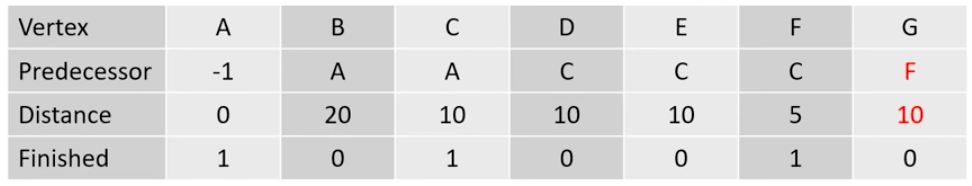
同樣檢視 C 的所有出邊，除了 A 已經處理完成，因此跳過以外，要更新 D、E、F 的距離，比如這時從 MST 中的 C 點出發只要透過 就可以到達 D 點，因此 D 的距離被更新為 的權重 10，類似的，E 與 F 的距離被更新為 10 和 5，因為 D、E、F 三點的距離都更新為較小的值，因此 都設為 C。

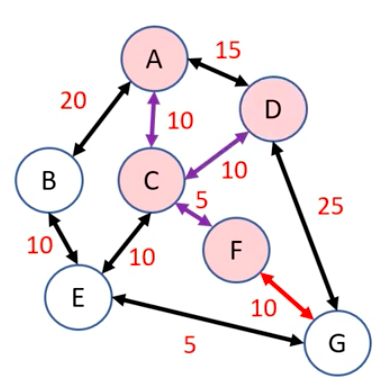
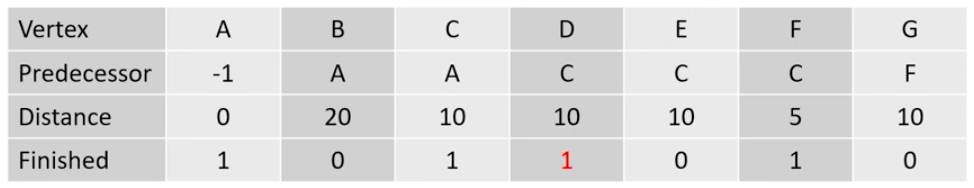


完成對 C 的處理後，重新檢視所有 值還是 0 的頂點，發現此時 最小的點是 F，因此對 F 點進行處理。



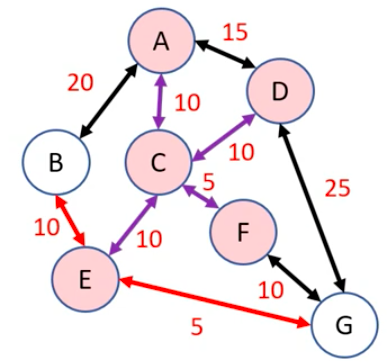
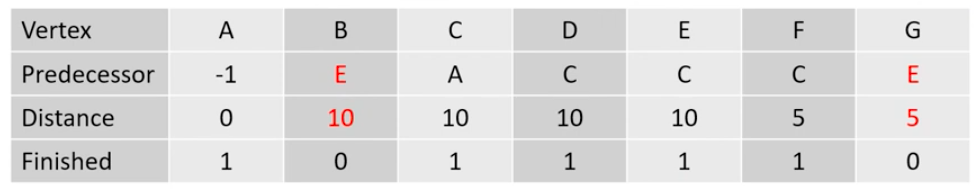
F 的 值設為 1，因為有出邊 ，所以把 G 點的距離值更新為 10、 更新為 F。



再來，任選此時距離最小、同為 10 的 D、E、G 點之一加入 MST，比如選擇 D 加入。

更新 D 點的 值為 1，並檢查出邊 和 ，因為 A 已經在 MST 中，不做處理，而 不能使 G 點與 MST 的距離更近，因此也不做處理。

再挑選 E、G 中一點納入，比如納入 E 並做完對應處理後，B 點的距離變為 10、G 點的距離變為 5。



隨後依序將 G、B 兩點加入 MST 中，完成整個最小生成樹。

觀察整個過程，所有 Finished 的頂點被看作是已經加入正在擴展的最小生成樹中，而所有由這些頂點向外連接的邊，都是由 Finished 連到 non-Finished 的 crossing edges。

在 crossing edges 間，需要挑選權重最小的那個 light edge 來加入，每輪加入一個邊並多連接到一個頂點，總共需要進行 輪。

（3）Prim' Algorithm 的複雜度

|  |  |
| --- | --- |
| Prim' Algorithm 的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | Prim(G,V,E,s){  // 初始化  for each v in V:  Predecessor[v]=-1  Distance[v] =  Finished[v] = 0  // 起點 s 的距離設為 0  Distance[s] = 0  // 共要加上 V-1 條邊  for i = 0~V-2:  // 找到目前距離最近且尚未加入的頂點  u = get\_min\_distance()  Finished[u] = 1  // 更新所有 u 的相鄰頂點的距離  for each vertex(v) in u.adjacent():  // 需要更新距離的情況  if (!Finished[v] && w(u,v)<=Distance[v]):  Predecessor[v] = u  Distance[v] = w(u,v)  } |

1. 建立資料結構並初始化：
2. 找到最近頂點，共找 次，

找到「一次」最近頂點的複雜度

* 矩陣：
* Binary heap：
* Fibonacci heap：

1. 更新頂點距離，需要更新 次

更新「一次」頂點的複雜度

* 矩陣：
* Binary heap：
* Fibonacci heap：

1. 總計

* 矩陣：
* Binary heap：
* Fibonacci heap：

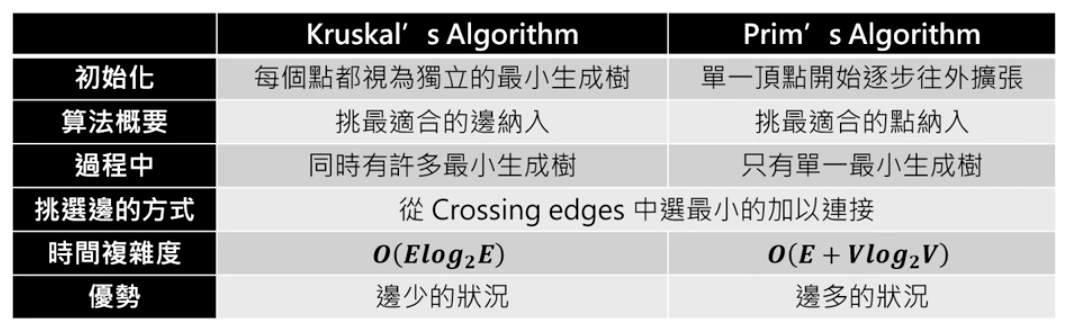
（4）實作 Prim’s Algorithm

以上一節的程式碼為基礎，加上 Prim 演算法的函式：

|  |  |
| --- | --- |
| Prim’s Algorithm | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146 | #include<iostream>  #include<stdlib.h>  #include<vector>  #include<list>  #include<queue> // for priority\_queue  using namespace std;  typedef struct{  // !注意要加上 from  int from;  int to;  int weight;  }edge;  class Graph{  private:  // 往哪個方向走與 MST 距離最短  vector<int> predecessor;  // 目前頂點與 MST 的距離  vector<int> distance;  // 目前頂點是否已處理完成  vector<bool> finished;  // 找到目前距離最短的頂點  int Find\_Minimal\_Distance();  vector<edge> MST\_Edges;  vector<int> MST\_Set;  int vertex;  vector<list<edge\*>> edges;  int Find\_Set(int);  void Merge\_Set(int, int);  public:  Graph(int);  void MST\_Prim();  void MST\_Kruskal();  void print\_edges();  void add\_edge(int, int, int=1);  };  // 找到目前距離 MST 最近的點  int Graph::Find\_Minimal\_Distance(){  int minimal = INT\_MAX;  int index = -1;  for (int i=0 ; i<vertex ; i++){  // 必須是 finish 值為 false 的點  if (distance[i]<minimal && !finished[i]){  // 回傳的頂點編號  index = i;  minimal = distance[i];  }  }  return index;  }  void Graph::MST\_Prim(){  // 初始化  predecessor.resize(vertex);  distance.resize(vertex);  finished.resize(vertex);  for (int i=0 ; i<vertex ; i++){  predecessor[i]=-1;  distance[i] = INT\_MAX;  finished[i] = false;  }  // 任意選一個起點，這裡選 0  distance[0] = 0;  int index=0;  // 目前的總權重  int sum = 0;  for (int i=0 ; i<vertex ; i++){  // 找到目前最近的未加入頂點  index = Find\_Minimal\_Distance();  // 把要加入的最近頂點標為 true  finished[index] = true;  // 加上到該點的邊權重  sum += distance[index];  // 檢查 index 的所有出邊以更新距離值  for (auto iter=edges[index].begin() ; iter!=edges[index].end() ;  iter++){  // 出邊指到的頂點  int target = (\*iter)->to;  // 該頂點目前的距離  int current\_weight = distance[target];  // 從 index 走到該頂點的距離  cross\_weight = (\*iter)->weight;  // 出邊指向的點已完成時不處理  if (finished[target]){ continue; }    // 需要更新距離時  if (cross\_weight < current\_weight){  // 更新距離  distance[target] = cross\_weight;  // 更新 predecessor  predecessor[target] = index;  }  } // end of inner for  } // end of outer for  // 透過 predecessor 得知 MST 中有哪些邊  for (int i=0 ; i<vertex ; i++){  if (predecessor[i] != -1){  cout << (char)(i+65) << "->" << (char)(predecessor[i]+65)  << endl;  }  }  cout << "Weight of this MST = " << sum << endl;  } // end of MST\_Prim  // 使用無向圖，加上 from  void Graph::add\_edge(int from, int to, int weight){  edges[from-1].push\_back(new edge[from-1, to-1, weight]);  edges[to-1].push\_back(new edge[to-1, from-1, weight]);  }  // 其他函式內容 ...  int main(){  Graph g(7);  g.Add\_Edge(1,2,20);  g.Add\_Edge(1,3,10);  g.Add\_Edge(1,4,15);  g.Add\_Edge(2,5,10);  g.Add\_Edge(3,5,10);  g.Add\_Edge(3,4,10);  g.Add\_Edge(3,6,5);  g.Add\_Edge(4,7,25);  g.Add\_Edge(6,7,10);  g.Add\_Edge(5,7,5);  g.MST\_Kruskal();  g.MST\_Prim();  return 0;  } |
| 執行結果 | |
| // Kruskal  C->F  E->G  C->A  C->E  E->B  D->C  Weight of this MST = 50  // Prim  B->E  C->A  D->C  E->C  F->C  G->E  Weight of this MST = 50 | |

可以看出不管使用的是 Kruksal 還是 Prim 演算法，得到的最小生成樹權重皆相同。

（5）比較 Kruskal 與 Prim 演算法



邊少時， 較小，可以使用 Kruskal 演算法（時間大多花在邊的排序上），邊多時則 Prim 演算法有優勢。

（6）LeetCode#1584. Min Cost to Connect All Points

A. 題目

給定一個陣列 ，代表了二維平面上一些頂點的整數座標，。

連接兩個頂點 和 的成本設定為 ，也就是 座標相減的絕對值與 座標相減的絕對值之和。

回傳要連接所有頂點的最小成本。

B. 出處

https://leetcode.com/problems/min-cost-to-connect-all-points/

C. 範例輸入與輸出

輸入：points = [[0,0],[2,2],[3,10],[5,2],[7,0]]

輸出：20

D. 解題邏輯

改用 Prim 演算法得到最小生成樹的權重即可。

|  |  |
| --- | --- |
| Min Cost to Connect All Points [Prim] | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | // 邊的結構  struct edge{  int from;  int to;  int cost;  };  class Solution{    vector<int> predecessor;  vector<int> distance;  vector<bool> finished;  vector<list<edge>> edges;  int vertex;    public:  int Find\_Minimal\_Distance(){  int minimal = INT\_MAX;  int index = -1;  for (int i=0 ; i<vertex ; i++){  if (distance[i]<minimal && !finished[i]){  index = i;  minimal = distance[i];  }  }  return index;  }  int minCostConnectPoints(vector<vector<int>>& points){    // 初始化  int total\_cost = 0;  vertex = points.size();  predecessor.resize(vertex);  distance.resize(vertex);  finished.resize(vertex);  edges.resize(vertex);  // 加入邊與權重  for (int i=0 ; i<vertex ; i++){  for (int j=i+1 ; j<vertex ; j++){  int current\_cost = abs(points[i][0]-points[j][0]) +  abs(points[i][1]-points[j][1]);  edge current1 = {i, j, current\_cost};  edge current2 = {j, i, current\_cost};  edges[i].push\_back(current1);  edges[j].push\_back(current2);  }  }  for (int i=0 ; i<vertex ; i++){  predecessor[i] = -1;  distance[i] = INT\_MAX;  finished[i] = false;  }  // 開始把頂點加入 MST 中  distance[0] = 0;  int index = 0;  for (int i=0 ; i<vertex ; i++){  index = Find\_Minimal\_Distance();  finished[index] = true;  total\_cost += distance[index];  auto iter = edges[index].begin();  for ( ; iter!=edges[index].end() ; iter++){  int target = iter->to;  int current\_weight = distance[target];  int cross\_weight = iter->cost;  if (finished[target]){ continue; }  if (cross\_weight < current\_weight){  distance[target] = cross\_weight;  predecessor[target] = index;  }  }  }    return total\_cost;    } // end of minCostConnectPoints  }; // end of Solution |