C/C++ 進階班

演算法

最小生成樹
(Minimal Spanning Tree)

李耕銘

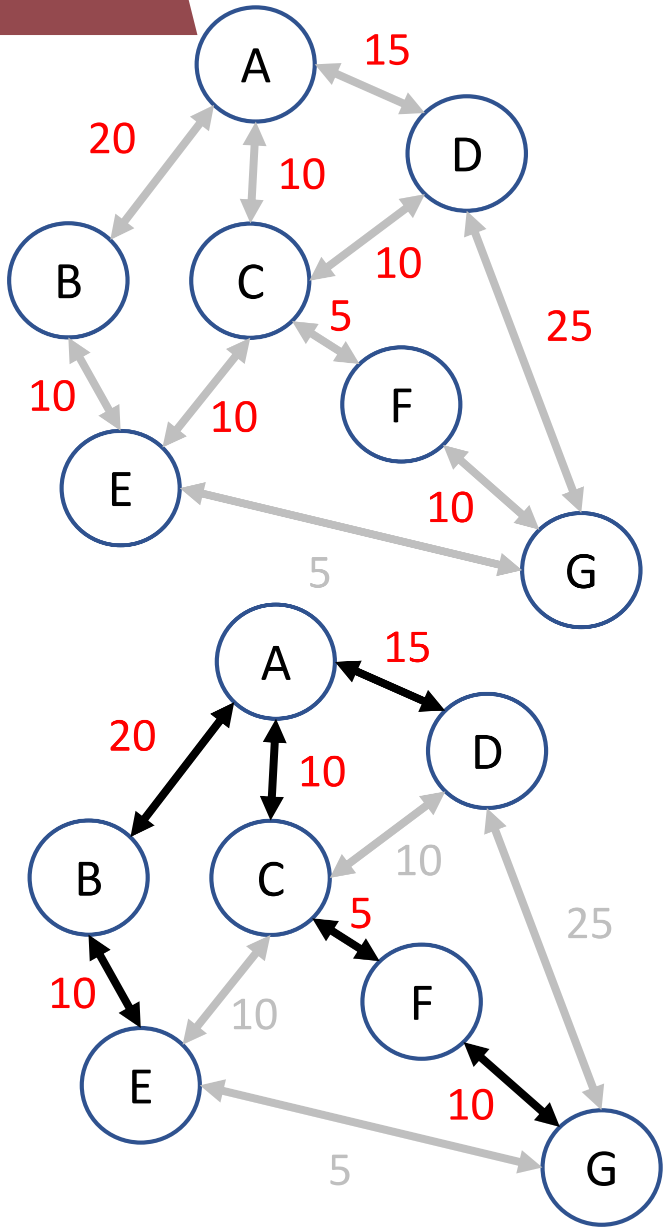
課程大綱

- 最小生成樹定義
- 最小生成樹的原理
- Kruskal's Algorithm
- Prim's Algorithm

最小生成樹定義

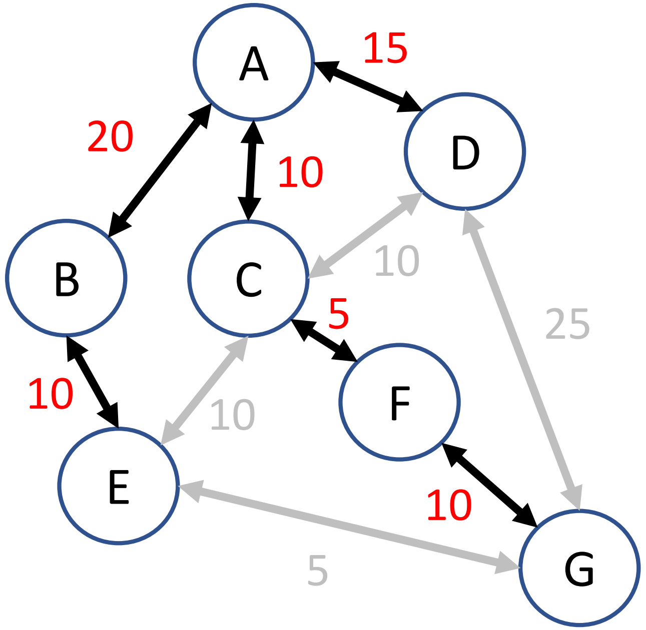
最小生成樹定義

- 生成樹 Spanning Tree
  - ➢ 自圖中取出所有頂點與部分邊，使其形成一棵樹
  - ➢ 圖上的點能互相連通，形成生成樹
  - ➢ 圖上的點無法兩兩互相連通，形成生成森林
  - ➢ 生成樹會有很多組解
  - ➢ 通常是無向圖
- 回憶：
  1. 樹不能有環
  2. 若有 |V| 個頂點，|V|-1 條邊



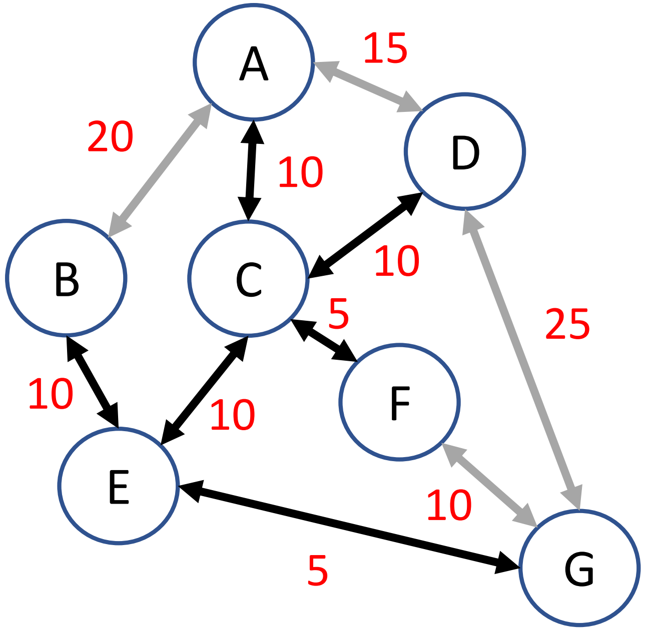
最小生成樹定義

- 生成樹 Spanning Tree 的權重
  - ➢ 是該生成樹所有邊的權重總和
    - ◘ 右圖：20+10+10+5+10+15=70
  - ➢ 選擇不同的邊組成生成樹，會有不同的權重



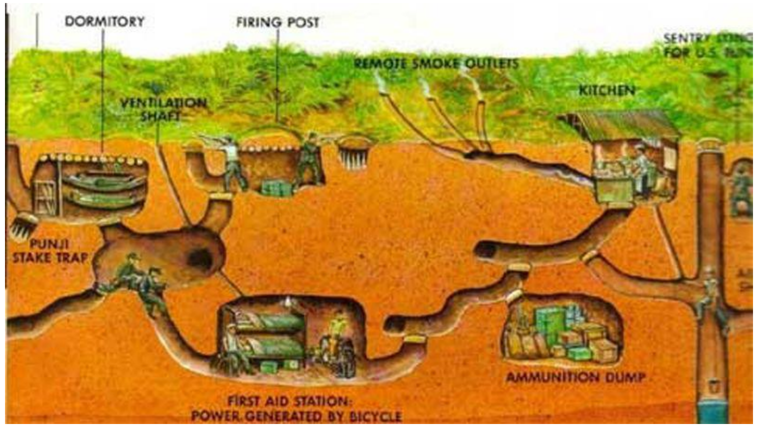
最小生成樹定義

- 最小生成樹 Minimal Spanning Tree
  - ➢ 所有生成樹裡，權重最小的樹為最小生成樹
  - ➢ Ex：5+10+10+10+10+5 = 50
  - ➢ 最小生成樹有 |V|-1 條邊
  - ➢ 最小生成樹可能有多組解



最小生成樹定義

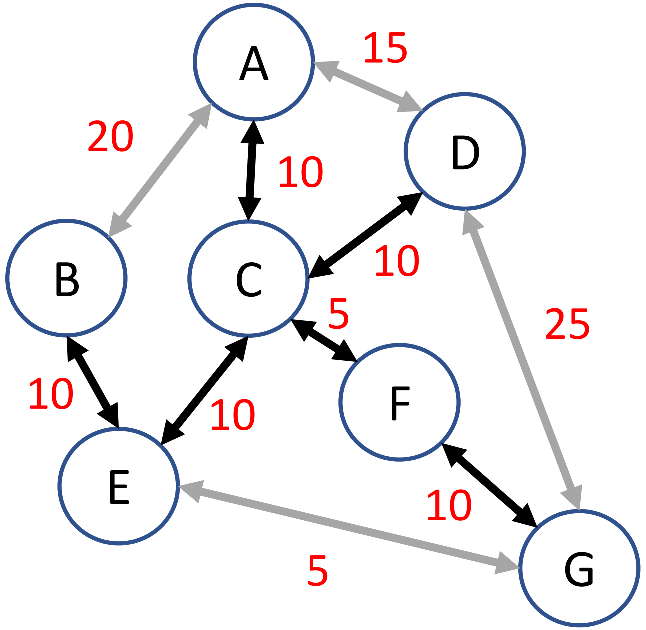
- 最小生成樹 Minimal Spanning Tree 的應用
  - 修路問題
    - 如何修路可以在成本最低下連接所有城市
  - 礦井通風口
    - 如何用最少的花費打通所有礦井
  - 網路通訊
  - 水利工程
  - ...



最小生成樹的原理

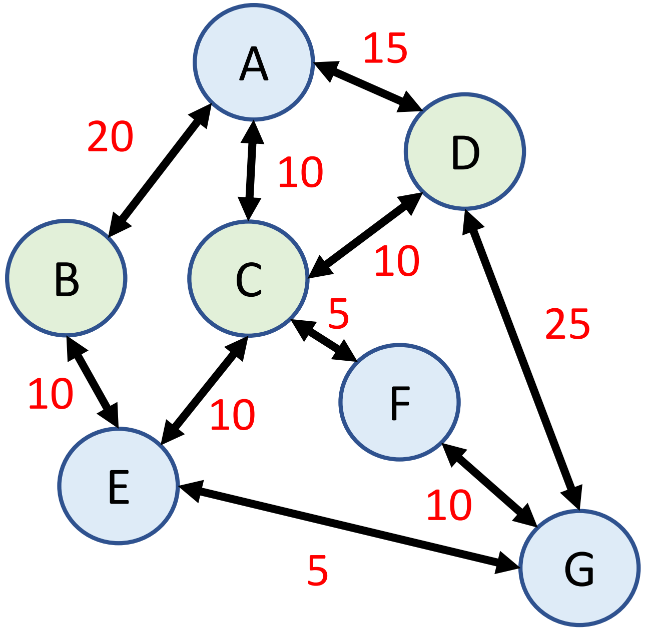
最小生成樹原理

- 最小生成樹 Minimal Spanning Tree (MST) 的規則
  - 自所有的邊中取出 |V|-1 條邊
  - 把所有邊區分成
    1. 屬於最小生成樹 (set A)
    2. 不屬於最小生成樹 (set B)
  - 逐步自 set B 中挑選邊至 set A
  - 安全 (Safe)
    - 把邊加入 set A 後，set A 的邊仍屬於最小生成樹



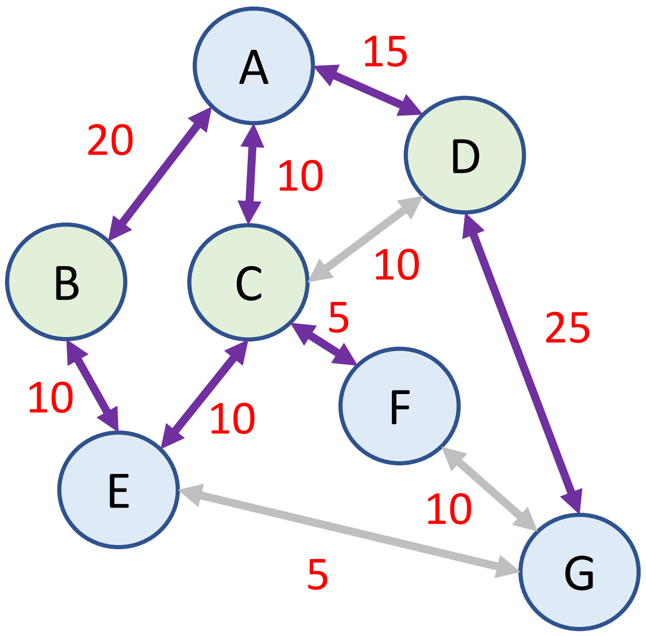
最小生成樹原理

- Cut 割
  - 頂點的集合
  - 把所有**頂點**切割成兩獨立集合
  - Ex：
    - Cut： BCD 與 AEFG



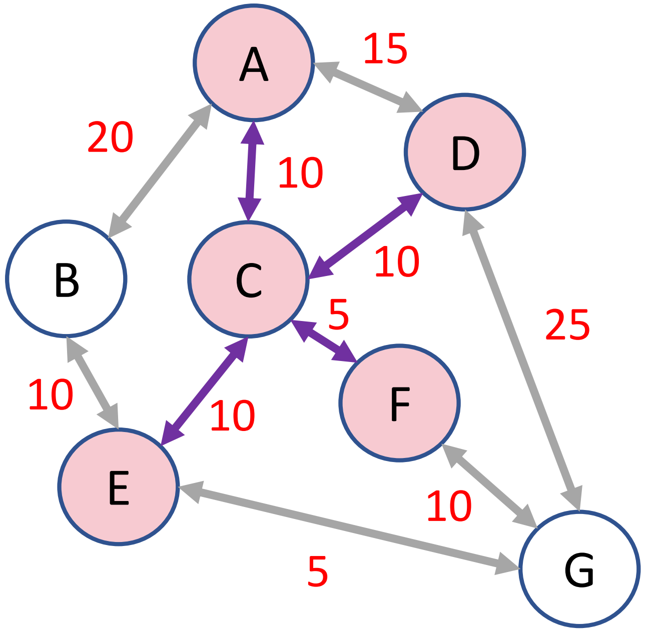
最小生成樹原理

- Crossing edges
  - Cut 把所有頂點切割成兩獨立集合
  - 某邊可連接這兩集合
  - Ex：AB、AC、AD、BE、CE、CF、DG
- Cut 割的權重
  - 所有 crossing edges 的權重和
  - Ex：20+10+15+10+10+5+25=95
- Respect
  - 沒有任何一條 crossing edges
- Light edge
  - 在所有候選邊中，權重最小的邊



最小生成樹原理

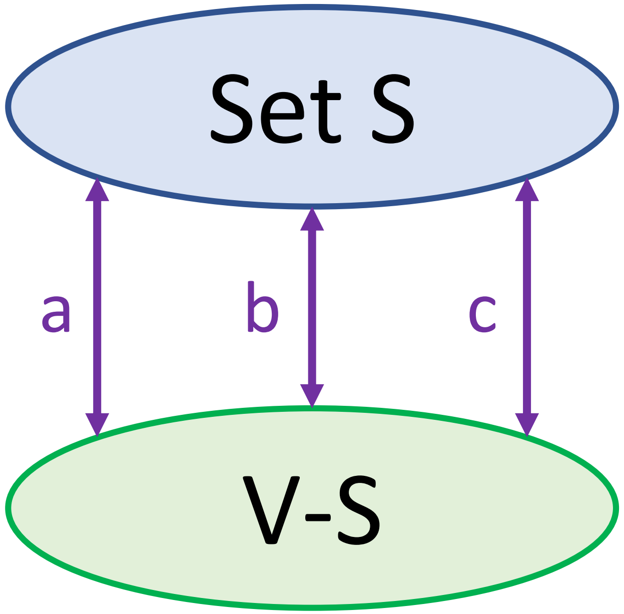
- 定理：已知圖 G(V,E) 是有權重、連接、無向圖
  1. Set A 是當前最小生成樹中，邊的子集合
     - Set A 是邊的集合
     - Set A：AC、CD 、CE 、CF
  2. Set S 是當前最小生成樹中，點的子集合
     - Set S 是點的集合
     - Set S：A、C 、 D、 E、 F
     - Set A 中的所有邊只存在於 set S 內
     - Set A 必須 respect Cut(S,V-S)
  3. 若 e(E,B) 是 crossing edge 與 light edge
     - 把 e 放入 set A 是安全的



最小生成樹原理

證明：已知圖 G(V,E) 是有權重、連接、無向圖

1. Set S 是當前最小生成樹中，點的子集合
   - V-S 是除 S 外，其餘點的集合
2. 若 S 與 V-S 彼此間都已經形成 MST
   - 生成樹的權重分別為：W_S, W_{V-S}
3. 其中有 a、b、c 三條 crossing edges
   - 已知 a 是 light edge
4. 最小生成樹是樹的一種，任兩點間只有唯一一條路徑
   - 選擇 a、b、c 後，都能連接兩集合後形成生成樹
   - 其中 a 的權重最低，最小生成樹的權重為：
   - $W_S + a + W_{V-S}$
5. 依序選擇 crossing edges 中的 light edge 就能夠擴展



最小生成樹原理

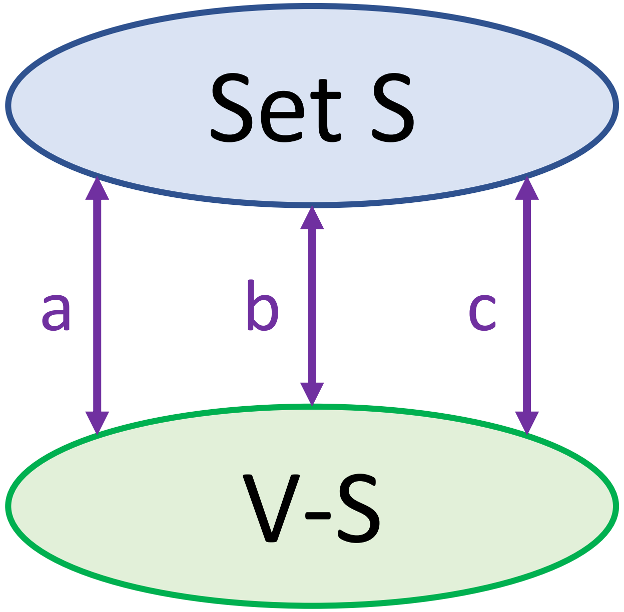
已知圖 G(V,E) 是有權重、連接、無向圖，換句話說：

- 若圖 G 中有多個最小生成樹的集合 T
- T 間的 crossing edge 之中的 light edge 是 safe 的！

set A 是當前最小生成樹的邊，把頂點用 Cut 分成兩集合：

1. 已經是最小生成樹 S
2. 其餘頂點 V - S

- 再從 crossing edges 中挑選 light edge 加入 set A
- 不斷重複便可以得到最小生成樹
- 本質上是一種貪婪演算法 (Greedy Algorithm)



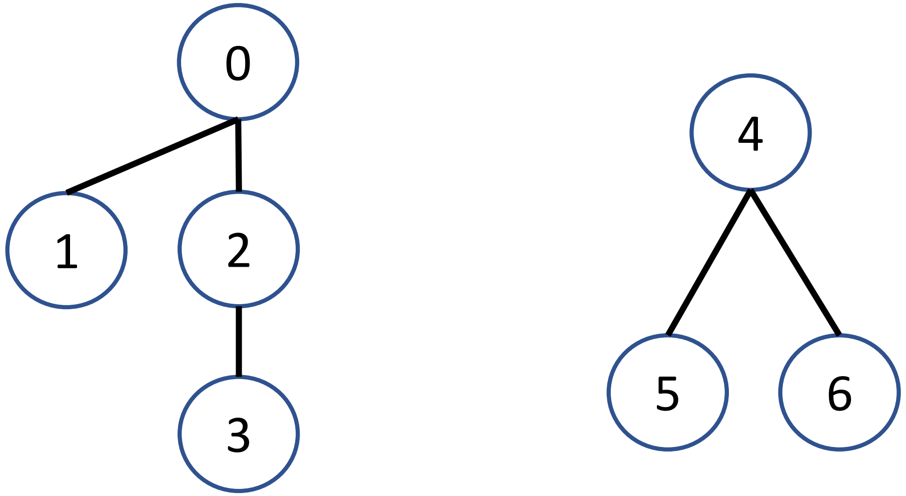
Kruskal's Algorithm

集合

Q：如何有效率的區分 MST/集合/分組？

A：把每一組建立成一棵樹

根節點決定了屬於哪個 MST



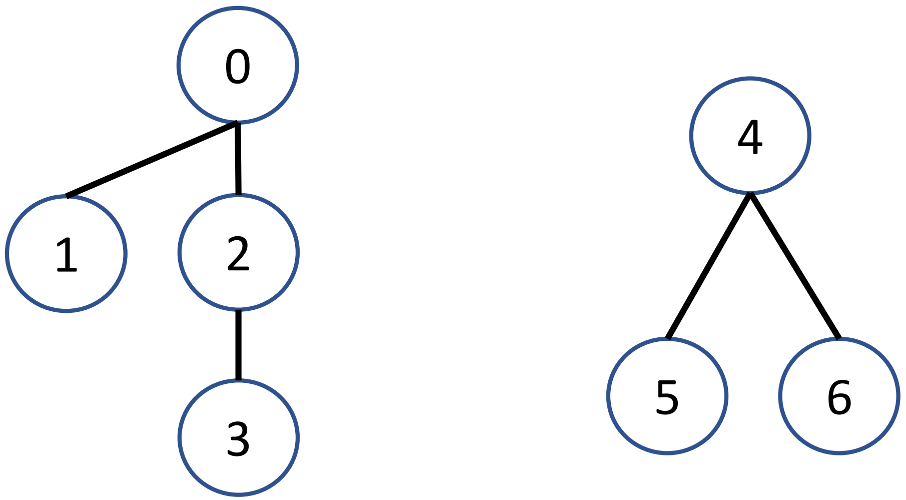
集合

以陣列 set 表示

1. set[v] ≥ 0時，表示頂點 v 的 predecessor 編號
2. set[v] < 0 時，表示該 MST/set 有 |set[v]| 個頂點

EX

Vertex	0	1	2	3	4	5	6
set	-4	0	0	2	-3	4	4



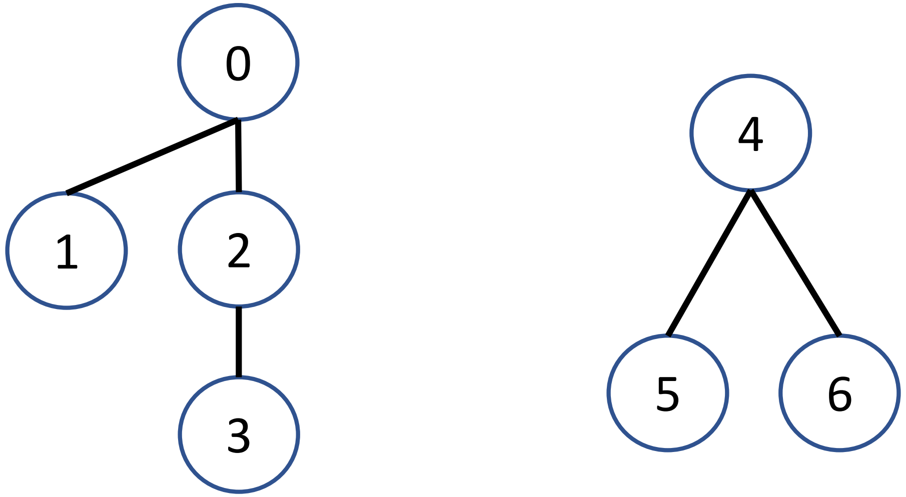
集合

如何找到 v 從屬於哪個 MST/set ？

1. 若 set[v] ≥ 0，則令 v = set[v]
2. 繼續往上找 set[v]
3. 直到 set[v] < 0

find_set

```
find_set(set, v){
    while(set[v] ≥ 0)
        v = set[v]
    return v
}
```

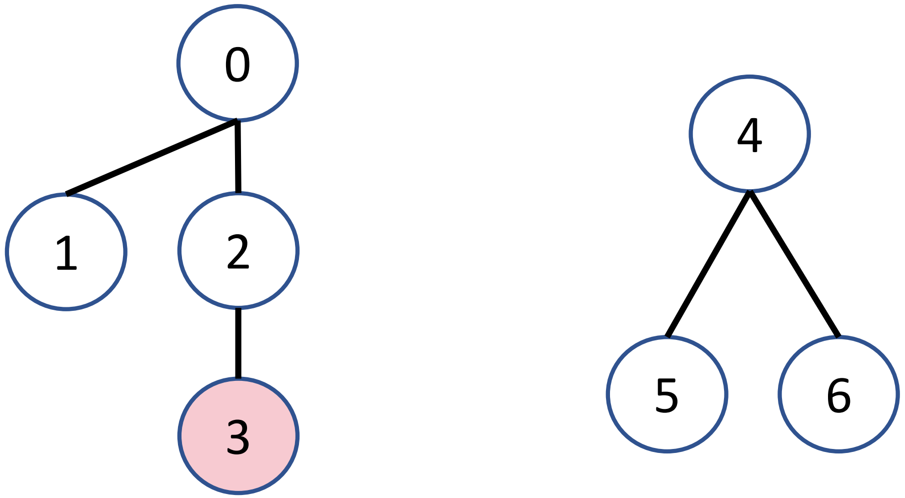


Vertex	0	1	2	3	4	5	6
set	-4	0	0	2	-3	4	4

集合

Collapsing

- 為了增進效能，把樹高塌陷成 1
- 只要經過 1 次搜尋就能知道該點屬於哪個集合
- 往根節點 root 中經過的頂點 v 都設定成：
  - set[v] = root



Find_Set

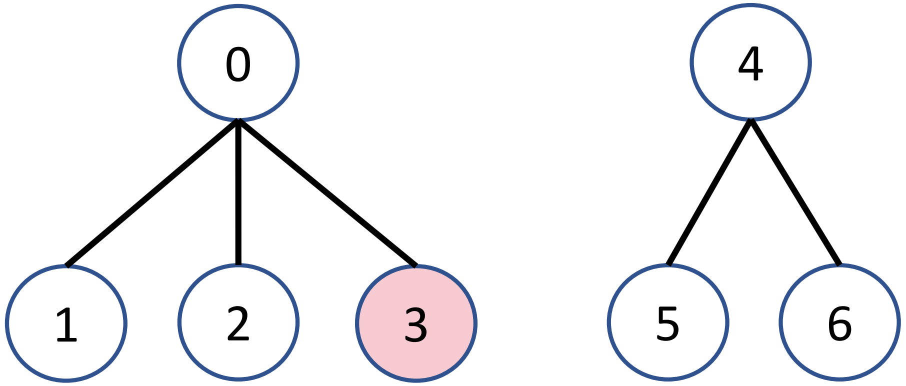
```
Find_Set (set, v){
    root = v
    while(set[root] ≥ 0)
        root = set[root]
    while(v != root)
        predecessor = set[v]
        set[v] = root
        v = predecessor
    return root
}
```

Vertex	0	1	2	3	4	5	6
set	-4	0	0	2	-3	4	4

集合

Collapsing

- 為了增進效能，把樹高塌陷成 1
- 只要經過 1 次搜尋就能知道該點屬於哪個集合
- 往根節點 root 中經過的頂點 v 都設定成：
    - set[v] = root



Find_Set

```
Find_Set (set, v){
    root = v
    while(set[root] ≥ 0)
        root = set[root]
    while(v != root)
        predecessor = set[v]
        set[v] = root
        v = predecessor
    return root
}
```

Vertex	0	1	2	3	4	5	6
set	-4	0	0	0	-3	4	4

集合

Q：如何新增/合併兩 MST/set？

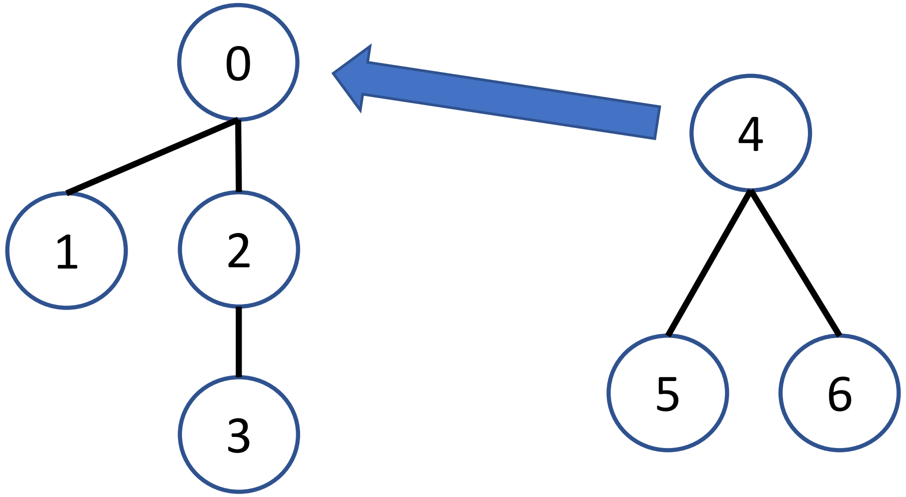
A：把 root 的 set 指向另一 root 即可！

Q：如何決定誰要從屬於誰？

A：通常個數越多的 set，樹高越大 (但不一定 :( )

讓個數少的 set(4) 從屬於個數多的 set(0)

最後記得讓 root 的 set 更新成 set(0)+set(4)



Vertex	0	1	2	3	4	5	6
set	-4	0	0	2	-3	4	4

集合

Q：如何新增/合併兩 MST/set？

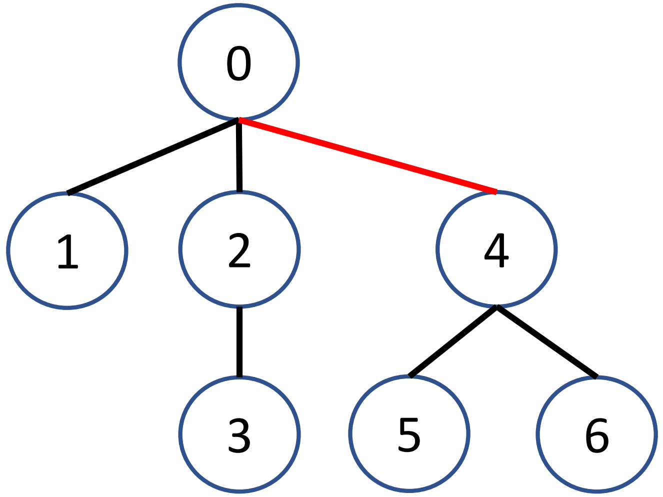
A：把 root 的 set 指向另一 root 即可！

Q：如何決定誰要從屬於誰？

A：通常個數越多的 set，樹高越大 (但不一定 :( )

讓個數少的 set(4) 從屬於個數多的 set(0)

最後記得讓 root 的 set 更新成 set(0)+set(4)



Vertex	0	1	2	3	4	5	6
set	-7	0	0	2	0	4	4

集合

Q：如何新增/合併兩 MST/set？

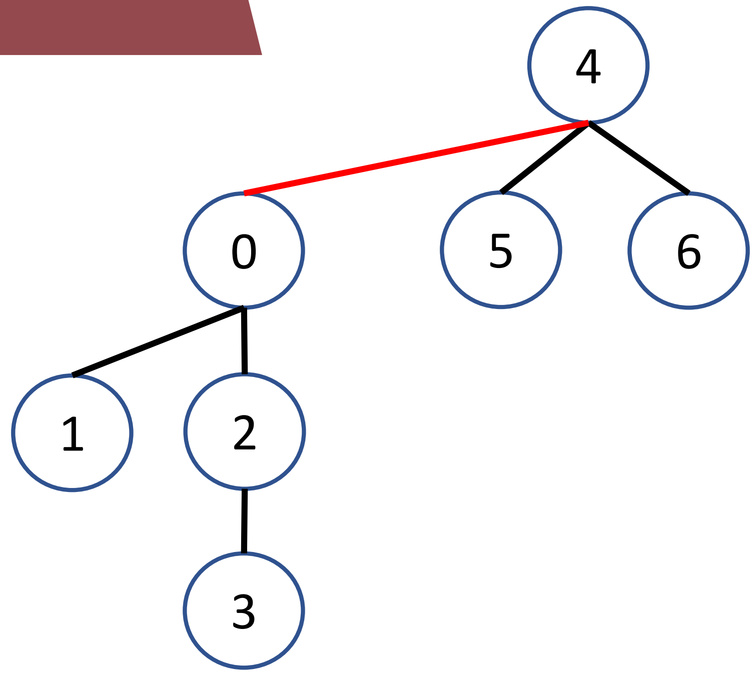
A：把 root 的 set 指向另一 root 即可！

Q：如何決定誰要從屬於誰？

A：通常個數越多的 set，樹高越大 (但不一定 :( )

讓個數少的 set(4) 從屬於個數多的 set(0)

最後記得讓 root 的 set 更新成 set(0)+set(4)



Vertex	0	1	2	3	4	5	6
set	4	0	0	2	-7	4	4

反過來樹高通常會更高 :(

集合

<u>Q：如何新增/合併兩 MST/set？</u>

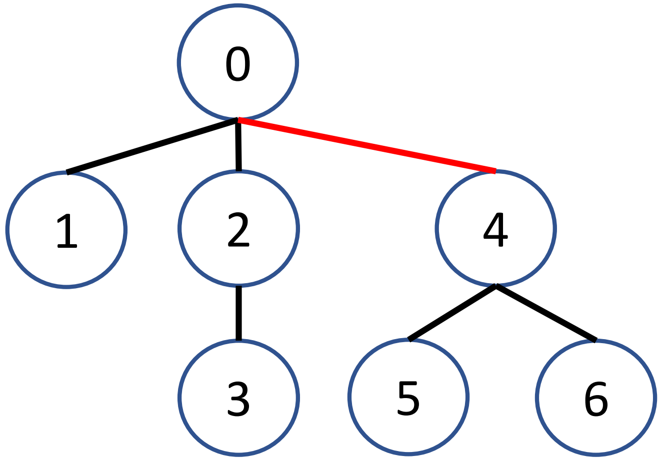
A：把 root 的 set 指向另一 root 即可！

<u>Q：如何決定誰要從屬於誰？</u>

A：通常個數越多的 set，樹高越大 (但不一定 :( )

讓個數少的 set(4) 從屬於個數多的 set(0)

最後記得讓 root 的 set 更新成 set(0)+set(4)



merge_set

```
merge_set(set, u, v){
    u_root = Find_Set(set,u)
    v_root = Find_Set(set,v)
    if(set[u_root] ≤ set[v_root])
        set[u_root]  += set[v_root]
        set[v_root] = u_root
    else
        set[v_root]  += set[u_root]
        set[u_root] = v_root
}
```

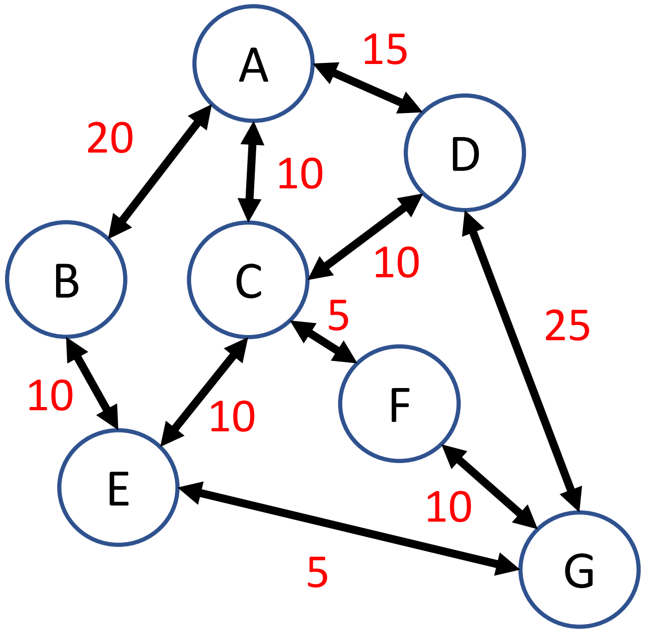
Vertex	0	1	2	3	4	5	6
set	-7	0	0	2	0	4	4

Example Code

Mission

完成以下兩函式備用：

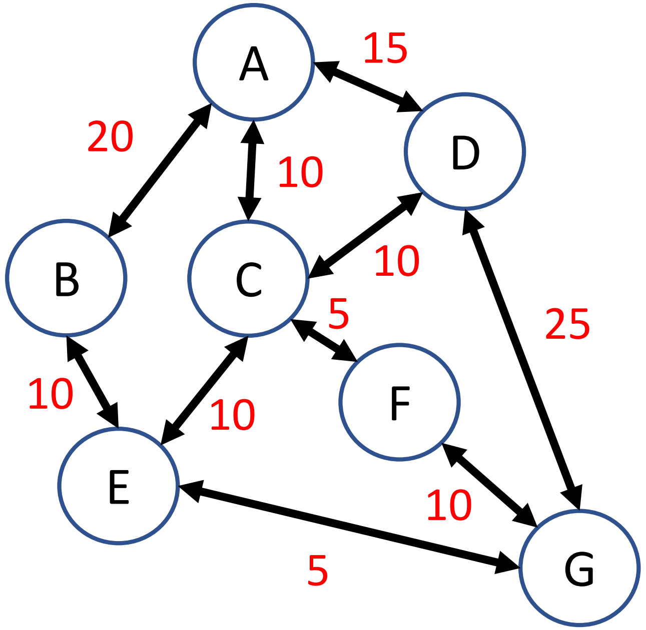
1. Find_Set
2. Merge_Set



Kruskal's Algorithm

Kruskal's Algorithm

1. 初始時將所有頂點視為獨立的 Set
   - ➢ 把每個頂點視為一棵 MST
2. 從連結各 set 或最小生成樹間的邊中挑選
   - ➢ 從中挑選權重最小的邊
   - ➢ 挑選後確認該邊是否為 crossing edge
     - ◘ 不是的話挑下一個！
   - ➢ 可記錄頂點當下所屬的 MST 來確認 crossing edge
   - ➢ 依序融合各最小生成樹，直到剩下最後一棵
3. 按照權重由小到大依序加入 set A
   - ➢ 但須注意/避免形成環 (樹沒有環！)

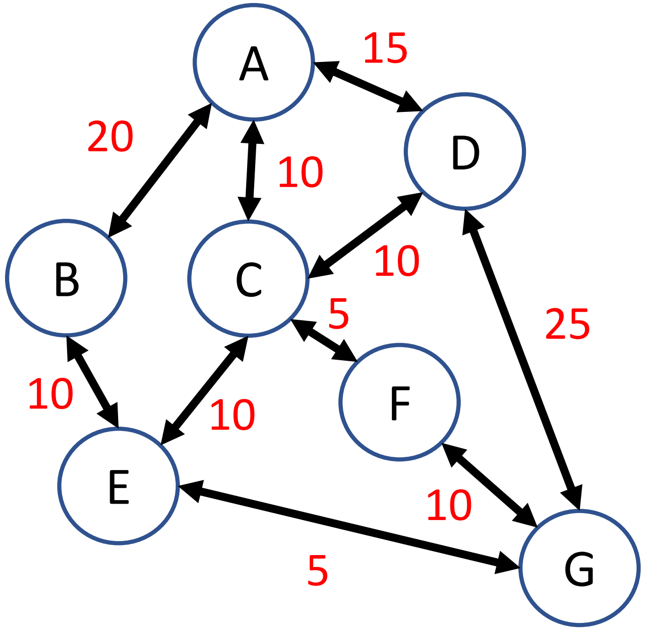


Kruskal's Algorithm

Kruskal's Algorithm

➢ 準備三種變數

1. MST_Edges：紀錄所有最小生成樹的邊
   ➢ 即剛提過的 set A，這些邊必在最後的 MST 內
2. MST_Set：紀錄目前每個頂點所屬的 MST
3. Sorted_Edges：把邊依照權重依小到大排列

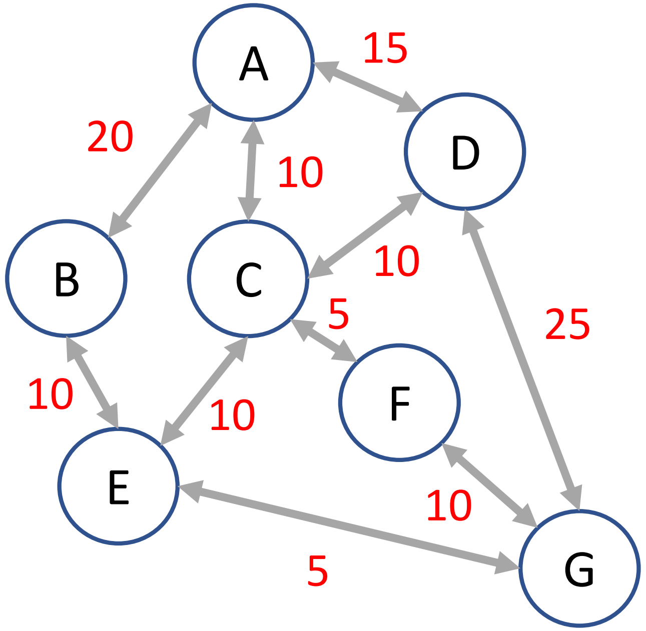


Kruskal's Algorithm

MST_Edges = {}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	-1	-1	-1	-1	-1	-1	-1



Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

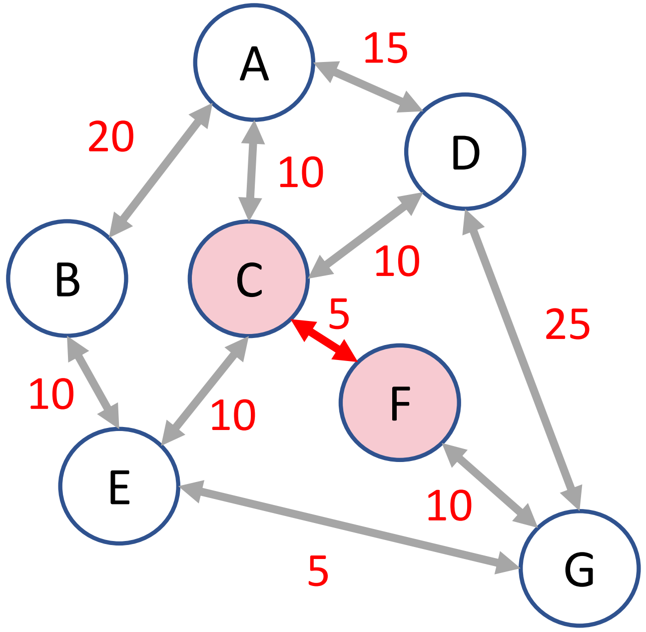
MST_Edges = {}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	-1	-1	-1	-1	-1	-1	-1

Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

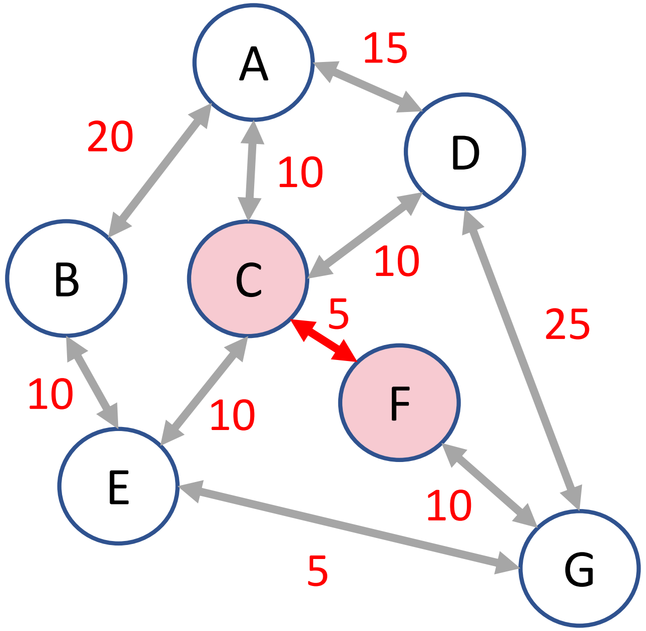


Kruskal's Algorithm

MST_Edges = {CF}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	-1	-1	-2	-1	-1	C	-1



Sorted_Edges

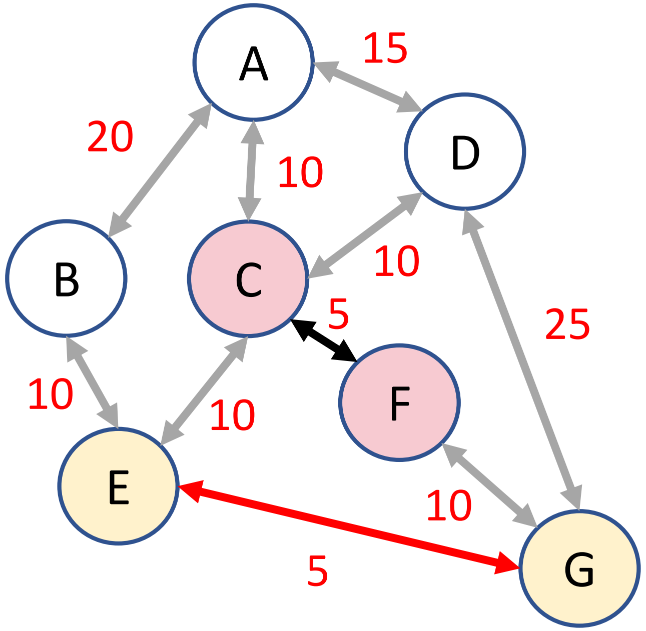
Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

MST_Edges = {CF}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	-1	-1	-2	-1	-1	C	-1



Sorted_Edges

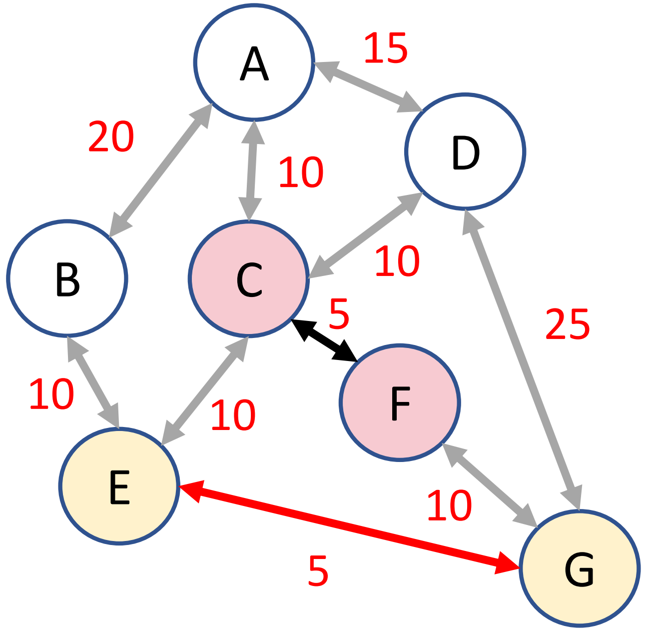
Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

MST_Edges = {CF,EG}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	-1	-1	-2	-1	-2	C	E



Sorted_Edges

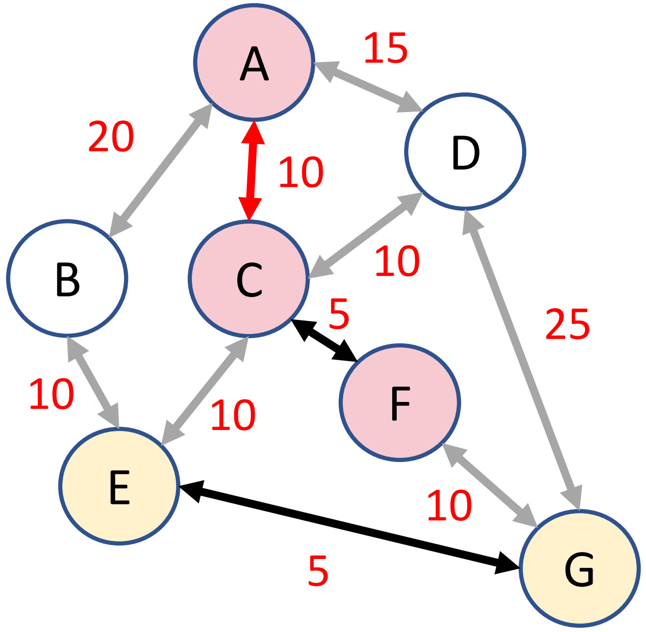
Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

MST_Edges = {CF,EG}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	-1	-1	-2	-1	-2	C	E



Sorted_Edges

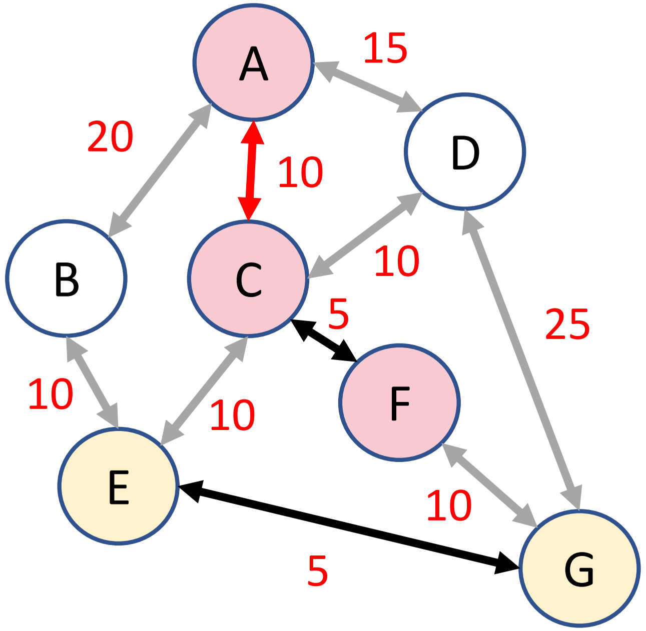
Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

MST_Edges = {CF,EG,AC}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-3	-1	-2	C	E



Sorted_Edges

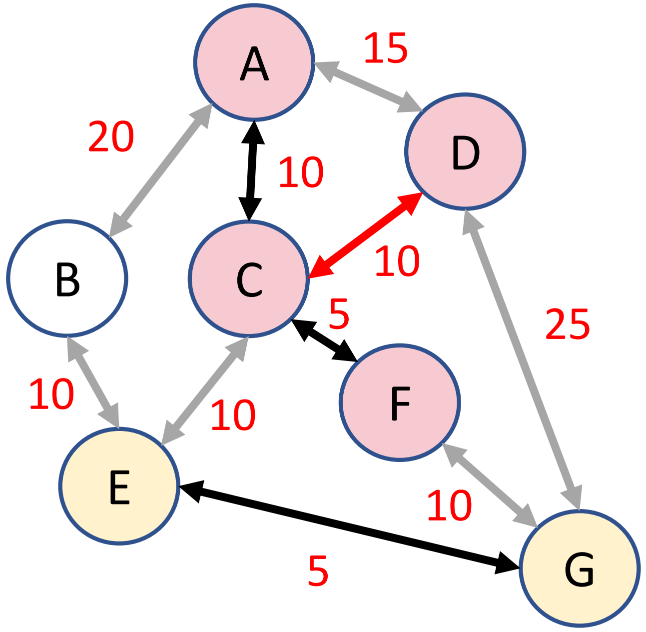
Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

MST_Edges = {CF,EG,AC}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-3	-1	-2	C	E



Sorted_Edges

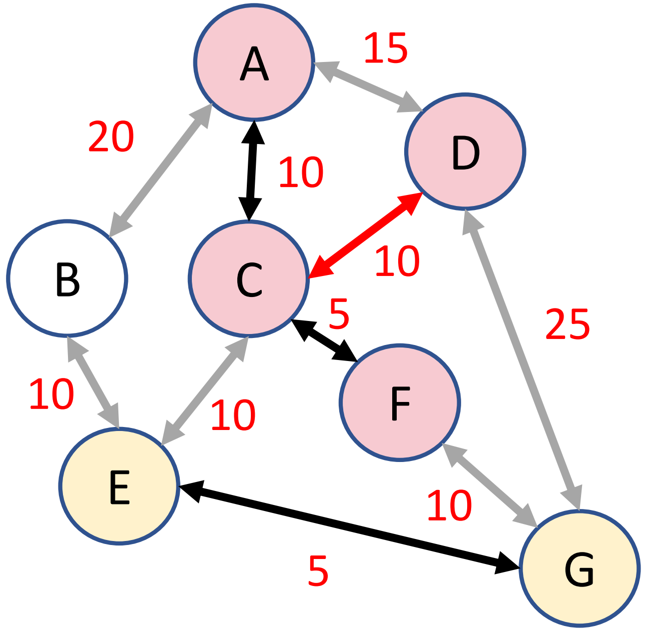
Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

MST_Edges = {CF,EG,AC,CD}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-4	C	-2	C	E



Sorted_Edges

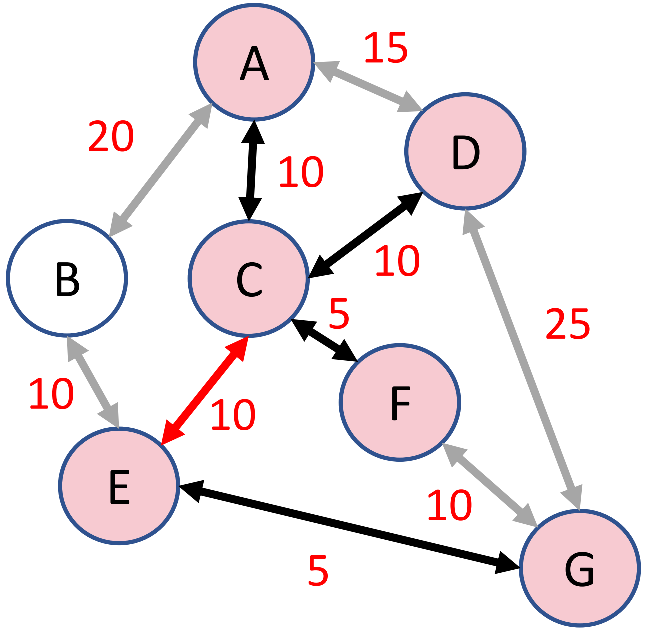
Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

MST_Edges = {CF,EG,AC,CD}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-4	C	-2	C	E



Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

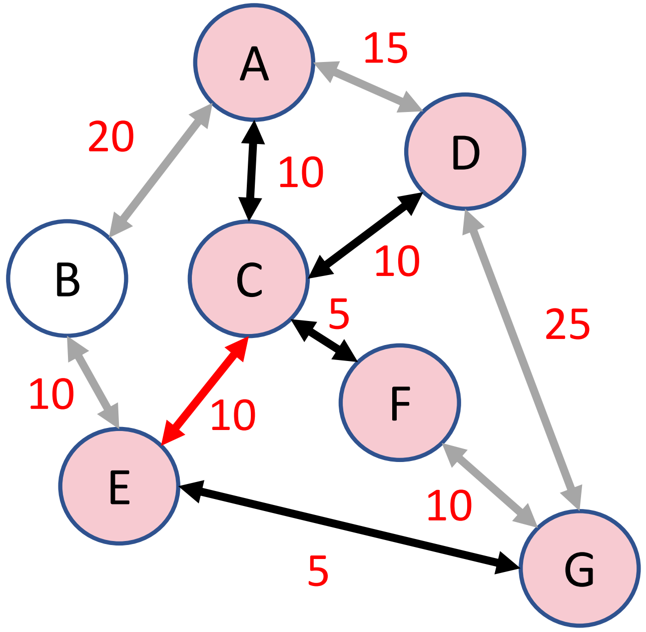
MST_Edges = {CF,EG,AC,CD,CE}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-6	C	C	C	E

Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G



Kruskal's Algorithm

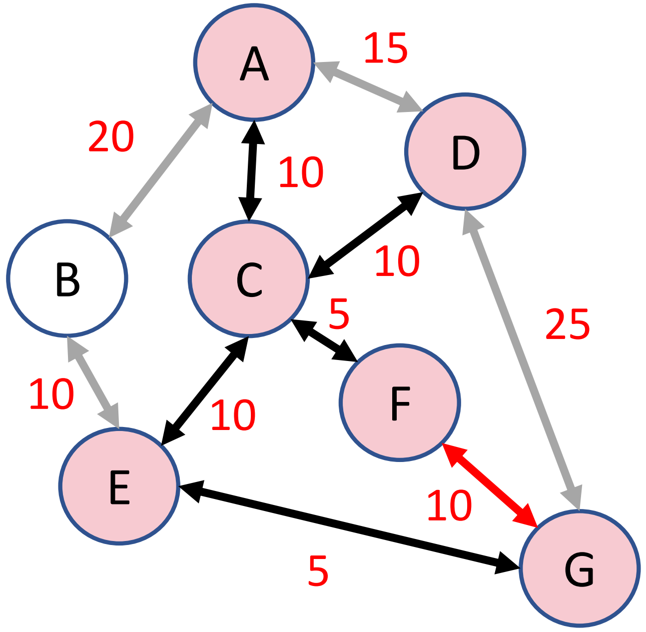
MST_Edges = {CF,EG,AC,CD,CE}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-6	C	C	C	E

Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G



Kruskal's Algorithm

MST_Edges = {CF,EG,AC,CD,CE}

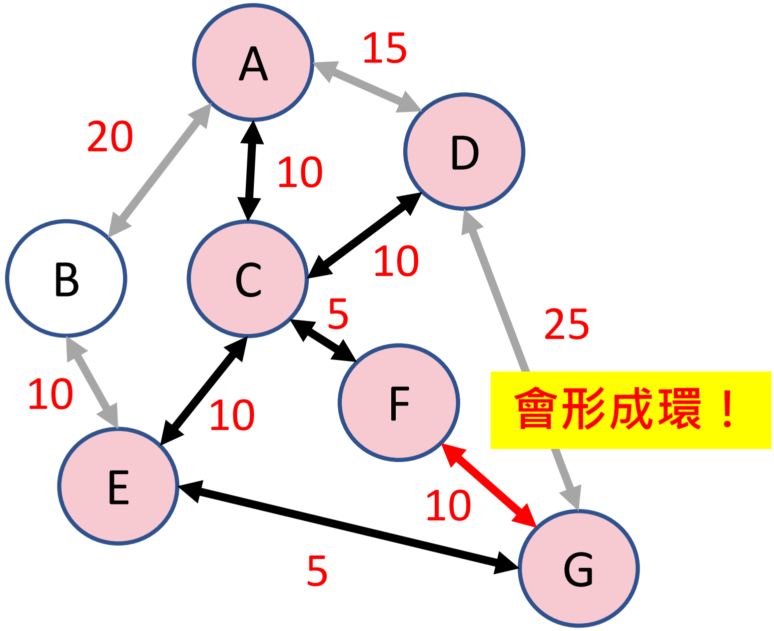
MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-6	C	C	C	E

Sorted_Edges

F →C
G→E→C

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G



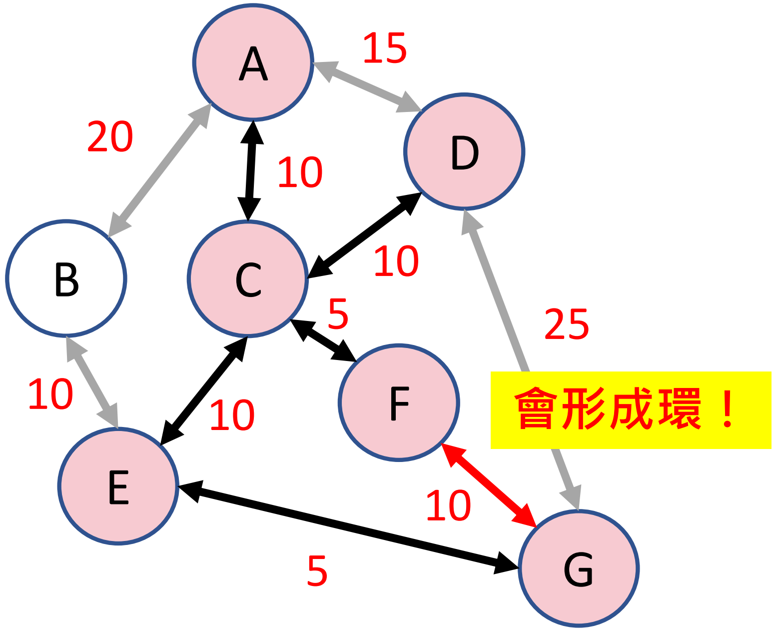
Kruskal's Algorithm

MST_Edges = {CF,EG,AC,CD,CE}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-6	C	C	C	C

順便把MST of G 改成 C



Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

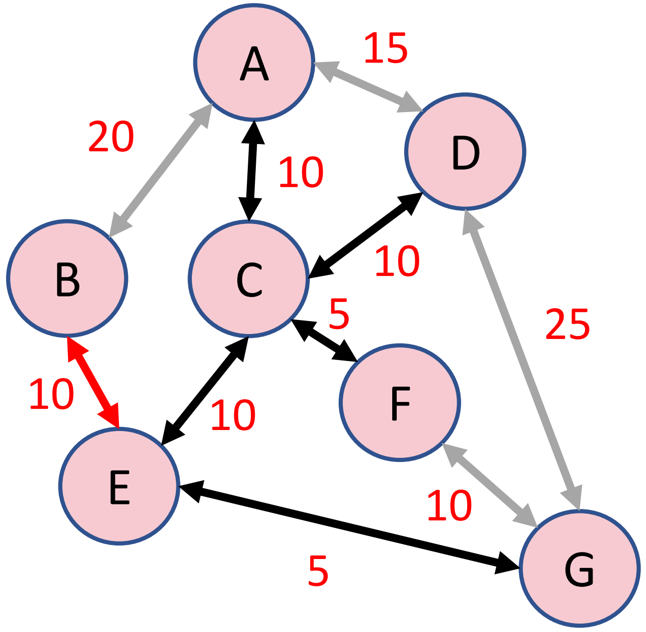
MST_Edges = {CF,EG,AC,CD,CE}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	-1	-6	C	C	C	C

Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G



Kruskal's Algorithm

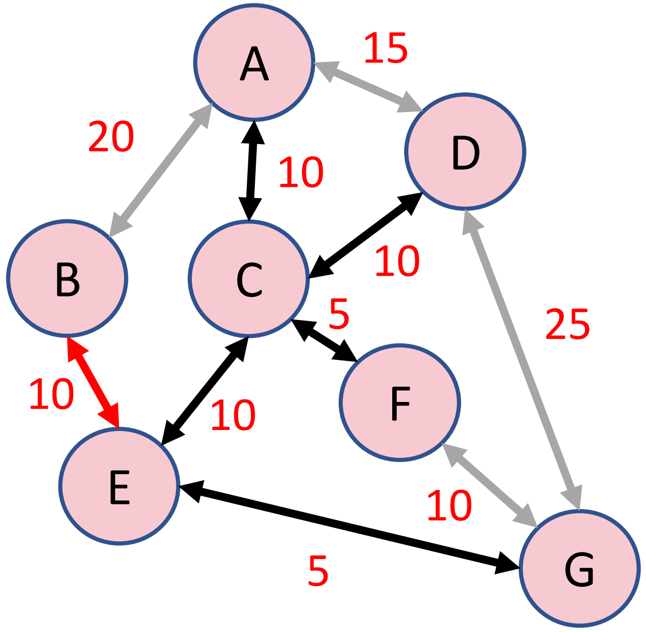
MST_Edges = {CF,EG,AC,CD,CE,BE}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	C	-6	C	C	C	C

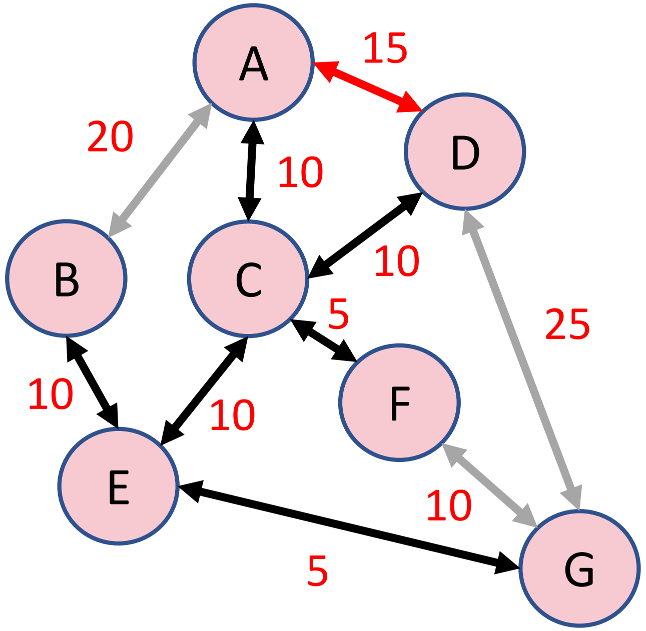
Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G



Kruskal's Algorithm

會形成環！



MST_Edges = {CF,EG,AC,CD,CE,BE}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	C	-6	C	C	C	C

Sorted_Edges

A →C
D →C

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

Kruskal's Algorithm

MST_Edges = {CF,EG,AC,CD,CE,BE}

MST_Set：-1 表示該 MST 內只有自己一個點

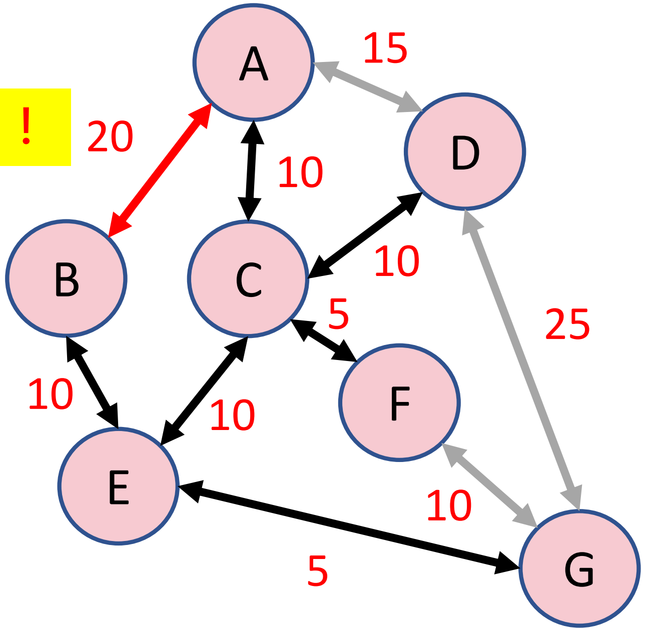
Vertex	A	B	C	D	E	F	G
MST	C	C	-6	C	C	C	C

Sorted_Edges

A →C
B →C

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G

會形成環！



Kruskal's Algorithm

MST_Edges = {CF,EG,AC,CD,CE,BE}

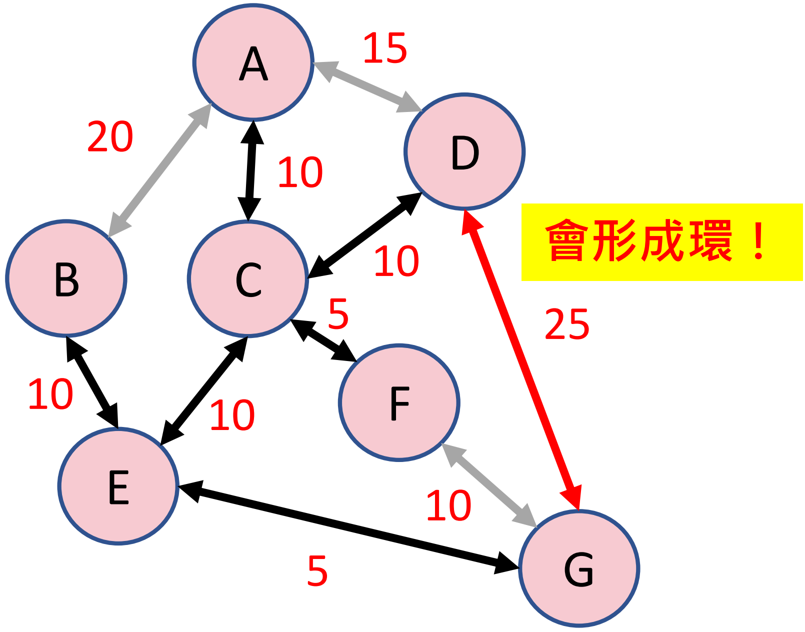
MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	C	-6	C	C	C	C

Sorted_Edges

D →C
G →C

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G



Kruskal's Algorithm

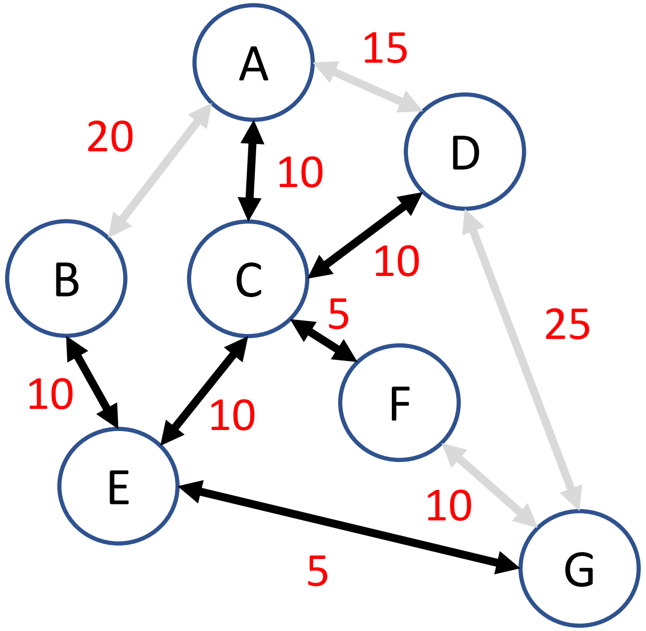
MST_Edges = {CF,EG,AC,CD,CE,BE}

MST_Set：-1 表示該 MST 內只有自己一個點

Vertex	A	B	C	D	E	F	G
MST	C	C	-6	C	C	C	C

Sorted_Edges

Vertex	C	E	A	C	C	F	B	A	A	D
Weight	5	5	10	10	10	10	10	15	20	25
Vertex	F	G	C	D	E	G	E	D	B	G



Kruskal's Algorithm

Kruskal's Algorithm

```
Kruskal (G,V,E){
    edges_completed = 0
    Sorted_Edges = priority_queue
    MST_Edges = array of edge
    for each v in V:
        MST_Set[v] = -1
    for each e in E:
        Sorted_Edges.push(e)
    for each e(u,v) in Sorted_Edges:
        if(Find_Set(u)!=Find_Set(v)):
            MST_Edges[edges_completed++] = e
            merge_set(set,u,v)
    return edges_MST
}
```

$O(V)$

$O(Elog_2E)$

$O(E)$

時間複雜度

初始化： $O(Elog_2E+V)=O(Elog_2E)$

把所有邊掃過：$O(E)$

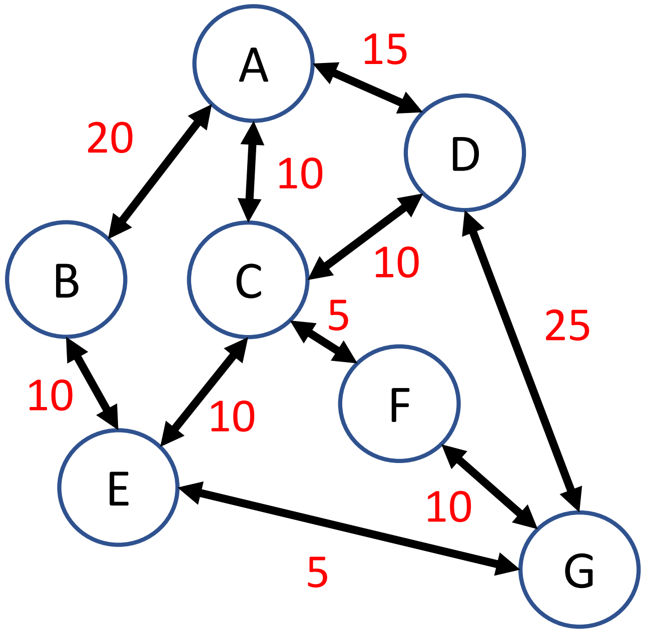
總和：$O(Elog_2E)$

又 $E=O(V^2)$；$O(Elog_2E)=O(Elog_2V)$

Example Code

Mission

寫出 Kruskal ‘s Algorithm！



Practice

Mission

LeetCode #1584. Min Cost to Connect All Points

You are given an array points representing integer coordinates of some points on a 2D-plane, where points[i] = [xi, yi].

The cost of connecting two points [xi, yi] and [xj, yj] is the manhattan distance between them: |xi - xj| + |yi - yj|, where |val| denotes the absolute value of val.

Return the minimum cost to make all points connected. All points are connected if there is exactly one simple path between any two points.

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

Prim's Algorithm

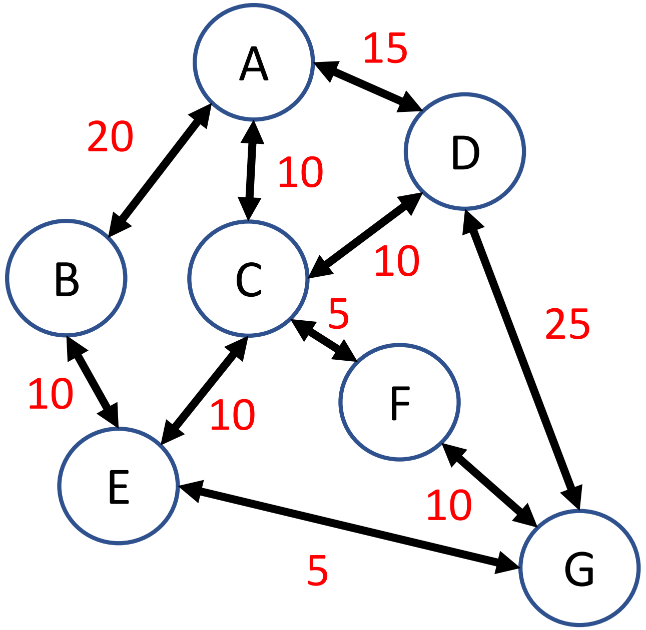
Prim's Algorithm

Kruskal's Algorithm

- 按照權重由小到大依序加入 set A
  - 但須注意/避免形成環 (樹沒有環！)

Prim's Algorithm

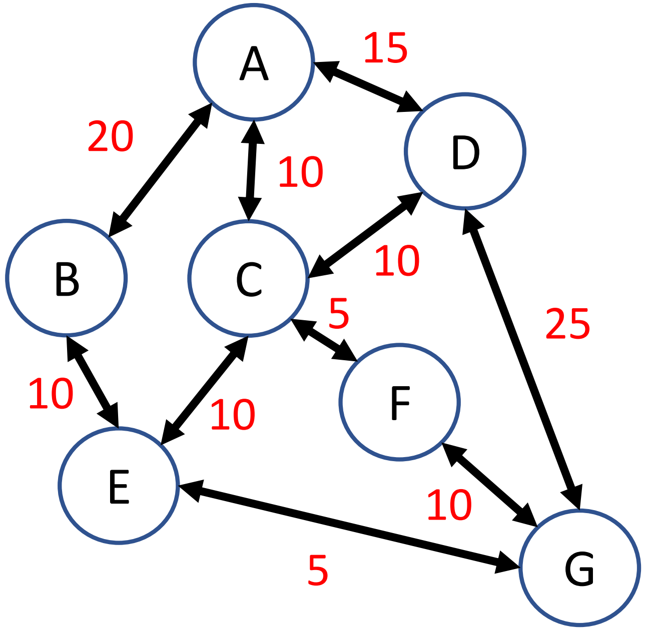
- 選定任一頂點作為 MST 的起點
- 從該點開始擴展，找尋每一點到 MST 的最短路徑！
- 實作上很像最短路徑演算法



Prim's Algorithm

Prim's Algorithm

- 準備三種變數
  1. Predecessor：每個頂點到 MST 最短路徑的 Parent
     - 初始化成 -1
  2. Distance：目前 MST 到該點的最短距離
     - 初始化成 ∞
  3. Finished：哪些頂點確定已被放入 MST
     - 初始化成 false/0

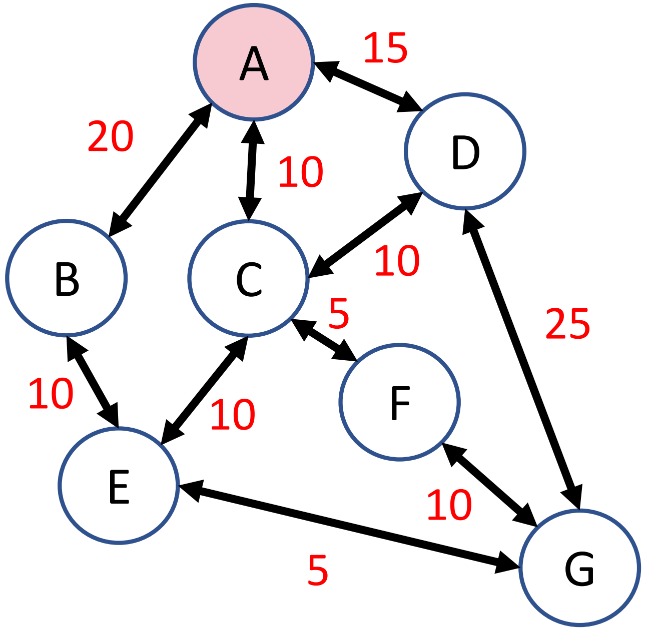


Vertex	A	B	C	D	E	F	G
Predecessor	-1	-1	-1	-1	-1	-1	-1
Distance	∞	∞	∞	∞	∞	∞	∞
Finished	0	0	0	0	0	0	0

Prim's Algorithm

Prim's Algorithm

- 選 A 點作為出發點，初始化：
  - Predecessor = -1 (不變)
  - Distance = 0
  - Finished = 1

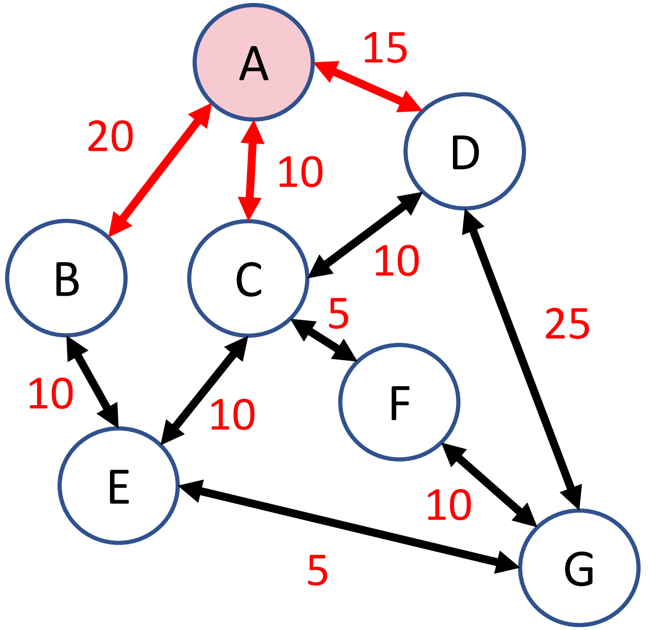


Vertex	A	B	C	D	E	F	G
Predecessor	-1	-1	-1	-1	-1	-1	-1
Distance	0	∞	∞	∞	∞	∞	∞
Finished	1	0	0	0	0	0	0

Prim's Algorithm

Prim's Algorithm

- 更新 A 的所有相鄰節點(v)
  - 如果 e(A,v) ≤ Distance[v]
    1. Predecessor[v] = A
    2. Distance[v] = e(A,v)
- 選目前 Distance 最小的點 C
  - Finished[C] = 1

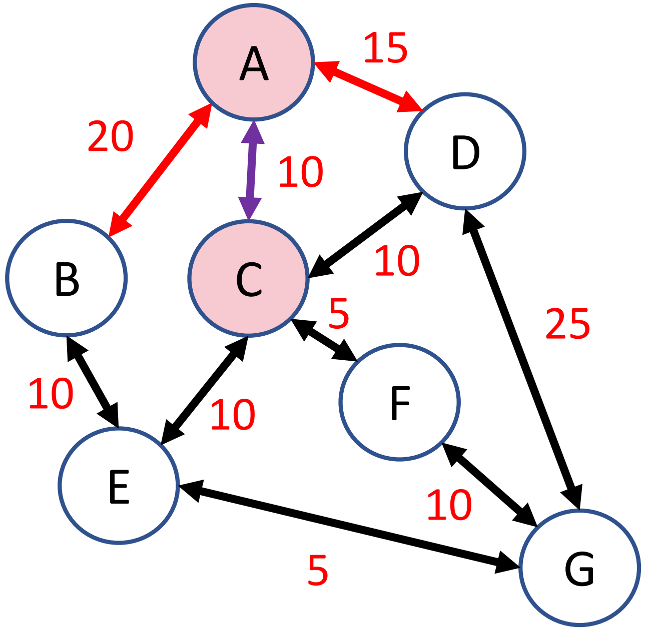


Vertex	A	B	C	D	E	F	G
Predecessor	-1	A	A	A	-1	-1	-1
Distance	0	20	10	15	∞	∞	∞
Finished	1	0	0	0	0	0	0

Prim's Algorithm

Prim's Algorithm

- ➢ 更新 A 的所有相鄰節點(v)
  - ◻ 如果 $e(A,v) \leq Distance[v]$
    1. Predecessor[v] = A
    2. Distance[v] = e(A,v)
- ➢ 選目前 Distance 最小的點 C
  - ◻ Finished[C] = 1

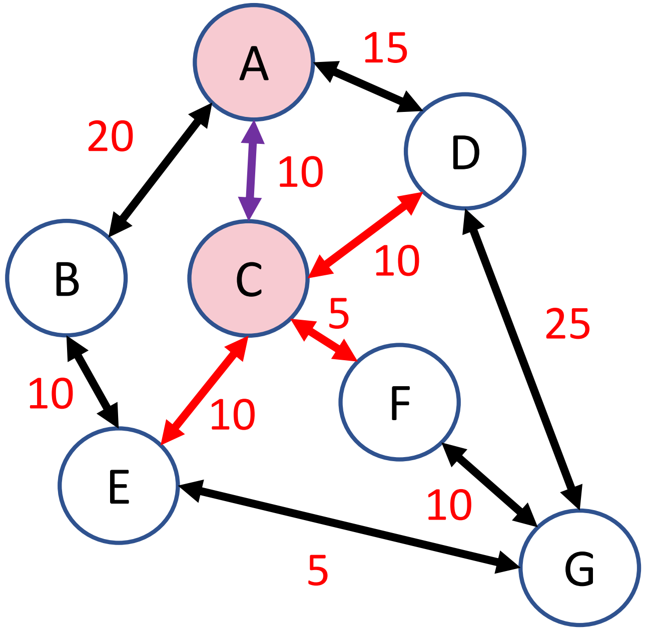


Vertex	A	B	C	D	E	F	G
Predecessor	-1	A	A	A	-1	-1	-1
Distance	0	20	10	15	∞	∞	∞
Finished	1	0	1	0	0	0	0

Prim's Algorithm

Prim's Algorithm

- 更新 C 的所有相鄰節點(v)
  - 如果 e(C,v) ≤ Distance[v]
    1. Predecessor[v] = C
    2. Distance[v] = e(C,v)
- 選目前 Distance 最小的點 F
  - Finished[F] = 1

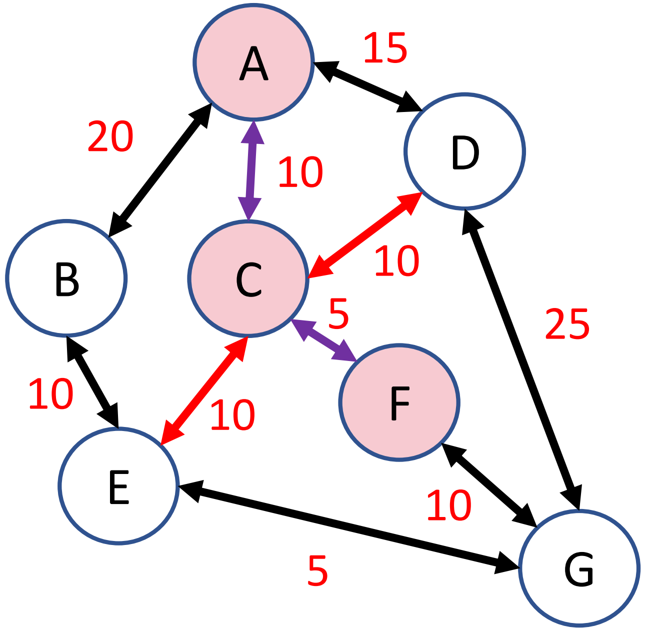


Vertex	A	B	C	D	E	F	G
Predecessor	-1	A	A	C	C	C	-1
Distance	0	20	10	10	10	5	∞
Finished	1	0	1	0	0	0	0

Prim's Algorithm

Prim's Algorithm

- 更新 C 的所有相鄰節點(v)
  - 如果 e(C,v) ≤ Distance[v]
    1. Predecessor[v] = C
    2. Distance[v] = e(C,v)
- 選目前 Distance 最小的點 F
  - Finished[F] = 1

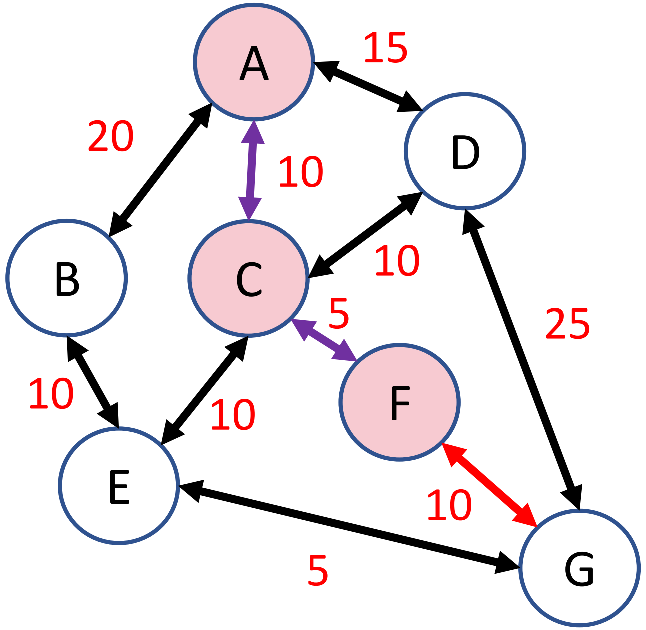


Vertex	A	B	C	D	E	F	G
Predecessor	-1	A	A	C	C	C	-1
Distance	0	20	10	10	10	5	∞
Finished	1	0	1	0	0	1	0

Prim's Algorithm

Prim's Algorithm

- 更新 F 的所有相鄰節點(v)
  - 如果 e(F,v) ≤ Distance[v]
    1. Predecessor[v] = F
    2. Distance[v] = e(F,v)
- 選目前 Distance 最小的點 D
  - Finished[D] = 1

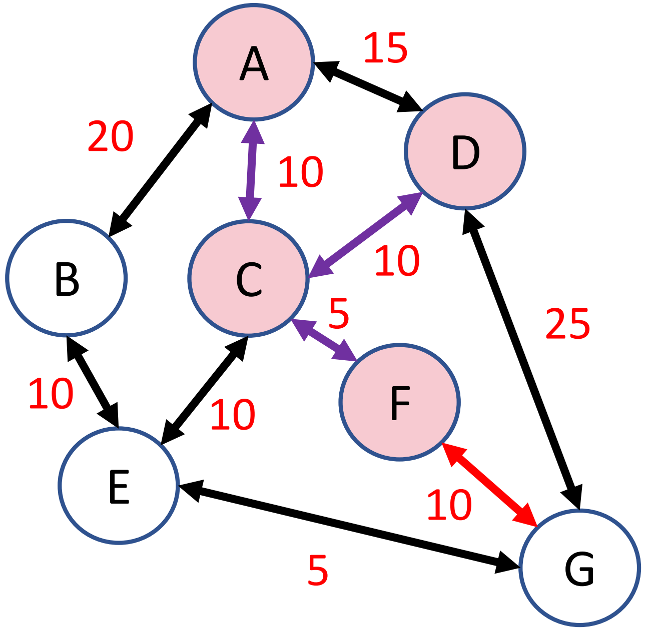


Vertex	A	B	C	D	E	F	G
Predecessor	-1	A	A	C	C	C	F
Distance	0	20	10	10	10	5	10
Finished	1	0	1	0	0	1	0

Prim's Algorithm

Prim's Algorithm

- 更新 F 的所有相鄰節點(v)
  - 如果 e(F,v) ≤ Distance[v]
    1. Predecessor[v] = F
    2. Distance[v] = e(F,v)
- 選目前 Distance 最小的點 D
  - Finished[D] = 1

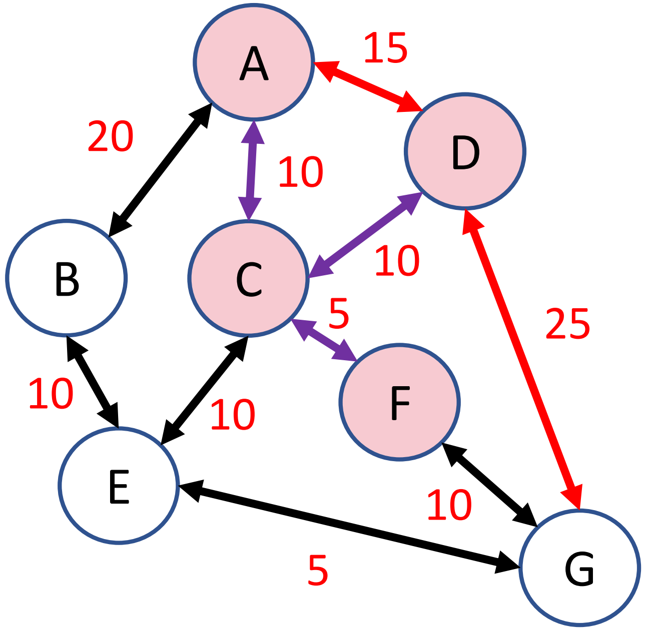


Vertex	A	B	C	D	E	F	G
Predecessor	-1	A	A	C	C	C	F
Distance	0	20	10	10	10	5	10
Finished	1	0	1	1	0	1	0

Prim's Algorithm

Prim's Algorithm

- ➢ 更新 D 的所有相鄰節點(v)
  - ◘ 如果 $e(D,v) \leq Distance[v]$
    1. Predecessor[v] = D
    2. Distance[v] = e(D,v)
- ➢ 選目前 Distance 最小的點 E
  - ◘ Finished[E] = 1

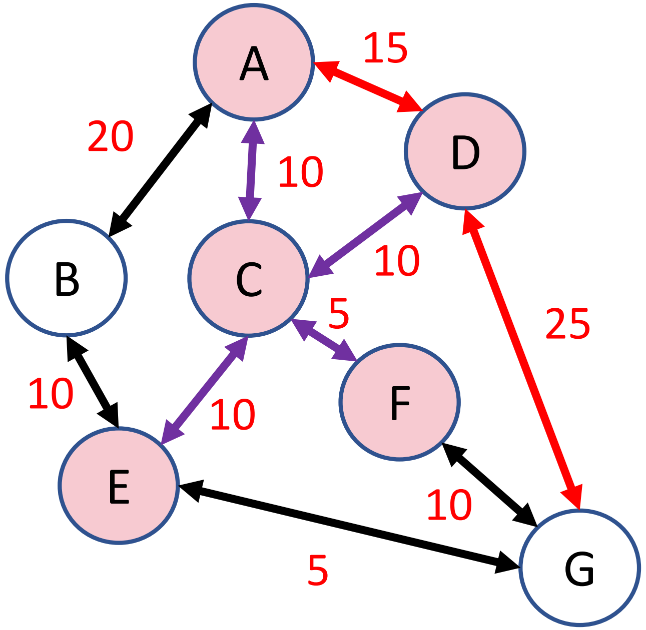


Vertex	A	B	C	D	E	F	G
Predecessor	-1	A	A	C	C	C	F
Distance	0	20	10	10	10	5	10
Finished	1	0	1	1	0	1	0

Prim's Algorithm

Prim's Algorithm

- 更新 D 的所有相鄰節點(v)
  - 如果 e(D,v) ≤ Distance[v]
    1. Predecessor[v] = D
    2. Distance[v] = e(D,v)
- 選目前 Distance 最小的點 E
  - Finished[E] = 1

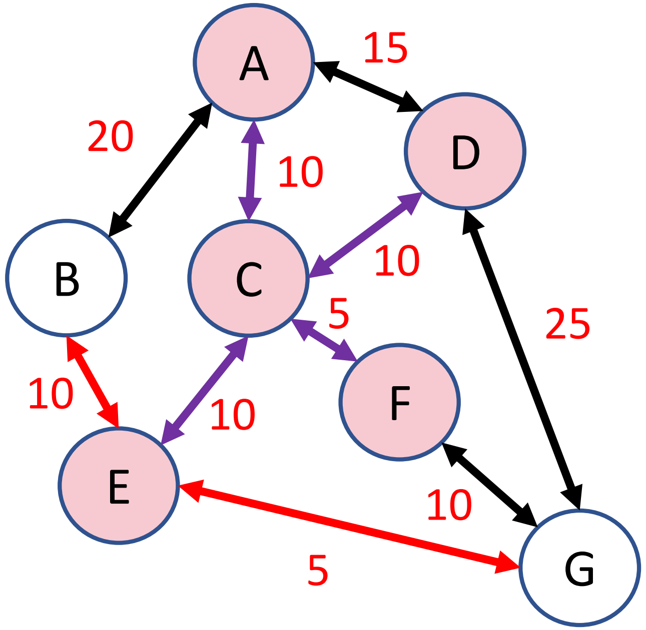


Vertex	A	B	C	D	E	F	G
Predecessor	-1	A	A	C	C	C	F
Distance	0	20	10	10	10	5	10
Finished	1	0	1	1	1	1	0

Prim's Algorithm

Prim's Algorithm

- ➢ 更新 E 的所有相鄰節點(v)
  - ☐ 如果 e(E,v) ≤ Distance[v]
    1. Predecessor[v] = E
    2. Distance[v] = e(E,v)
- ➢ 選目前 Distance 最小的點 G
  - ☐ Finished[G] = 1

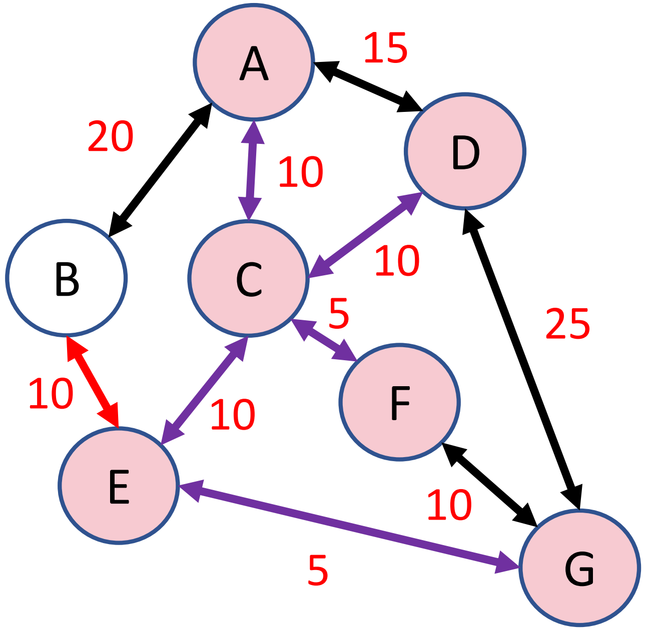


Vertex	A	B	C	D	E	F	G
Predecessor	-1	E	A	C	C	C	E
Distance	0	10	10	10	10	5	5
Finished	1	0	1	1	1	1	0

Prim's Algorithm

Prim's Algorithm

- 更新 E 的所有相鄰節點(v)
  - 如果 e(E,v) ≤ Distance[v]
    1. Predecessor[v] = E
    2. Distance[v] = e(E,v)
- 選目前 Distance 最小的點 G
  - Finished[G] = 1

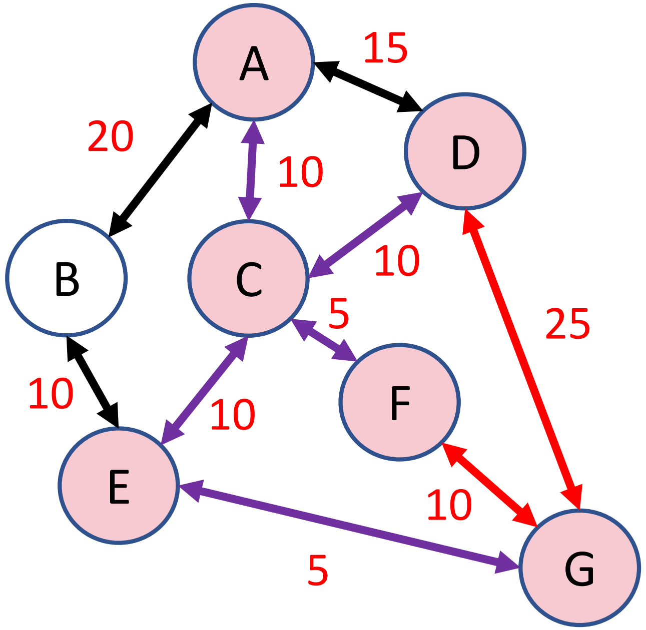


Vertex	A	B	C	D	E	F	G
Predecessor	-1	E	A	C	C	C	E
Distance	0	10	10	10	10	5	5
Finished	1	0	1	1	1	1	1

Prim's Algorithm

Prim's Algorithm

- 更新 G 的所有相鄰節點(v)
  - 如果 e(G,v) ≤ Distance[v]
    1. Predecessor[v] = G
    2. Distance[v] = e(G,v)
- 選目前 Distance 最小的點 B
  - Finished[B] = 1

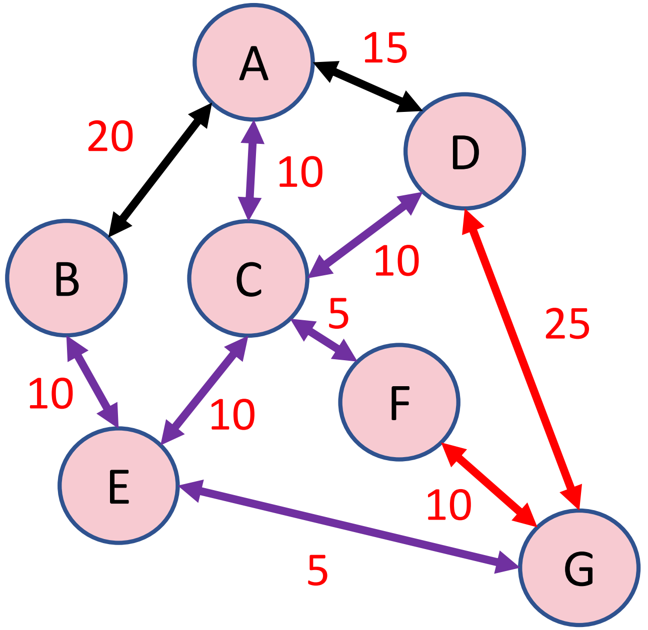


Vertex	A	B	C	D	E	F	G
Predecessor	-1	E	A	C	C	C	E
Distance	0	10	10	10	10	5	5
Finished	1	0	1	1	1	1	1

Prim's Algorithm

Prim's Algorithm

- ➢ 更新 G 的所有相鄰節點(v)
  - ◘ 如果 e(G,v) ≤ Distance[v]
    1. Predecessor[v] = G
    2. Distance[v] = e(G,v)
- ➢ 選目前 Distance 最小的點 B
  - ◘ Finished[B] = 1



Vertex	A	B	C	D	E	F	G
Predecessor	-1	E	A	C	C	C	E
Distance	0	10	10	10	10	5	5
Finished	1	1	1	1	1	1	1

Prim's Algorithm

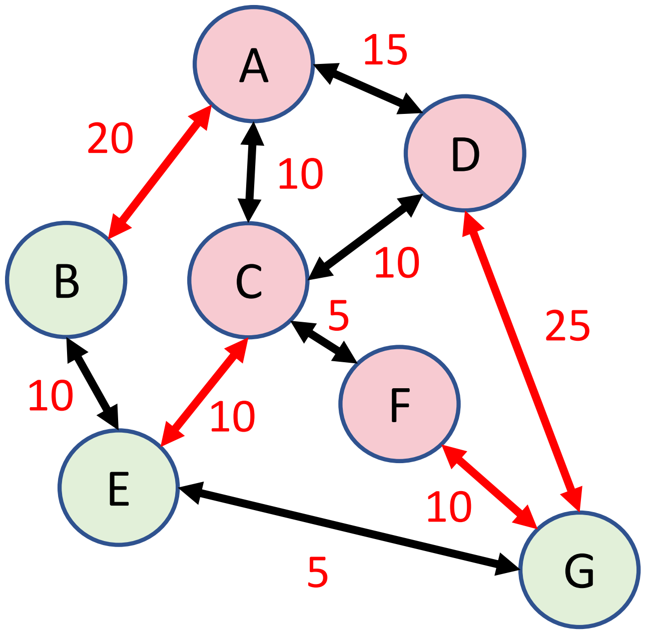
Prim's Algorithm 的原理

- 把所有 Finished 的頂點視為 MST
- 所有從 finished 連到 non-finished 的邊都是 crossing edges
- 從 crossing edges 中挑出最小的
  - 為 light edge，回憶：light edge 是 safe 的！
  - 把 light edge 置入 MST

Q：為何需要重複 V-1 輪？

A：每一輪步驟都能找到一條屬於 MST 的邊

而 MST 有 V-1 條邊，故需重複 V-1 次



Prim's Algorithm

Prim's Algorithm

```
Prim (G,V,E,s){
    for each v in V:                          O(V)
        Predecessor[v] = -1
        Distance[v] = ∞
        Finished[v] = 0
    Distance[s] = 0
    for i = 0~V-2:        O(V)
        u = get_min_distance()      O(?)
        Finished[u] = 1
        for each vertex(v) in u.adjacent():      O(2E)
            if(!Finished[v] && w(u,v)≤Distance[v]):
                Predecessor[v] = u
                Distance[v] = w(u,v)      O(?)
}
```

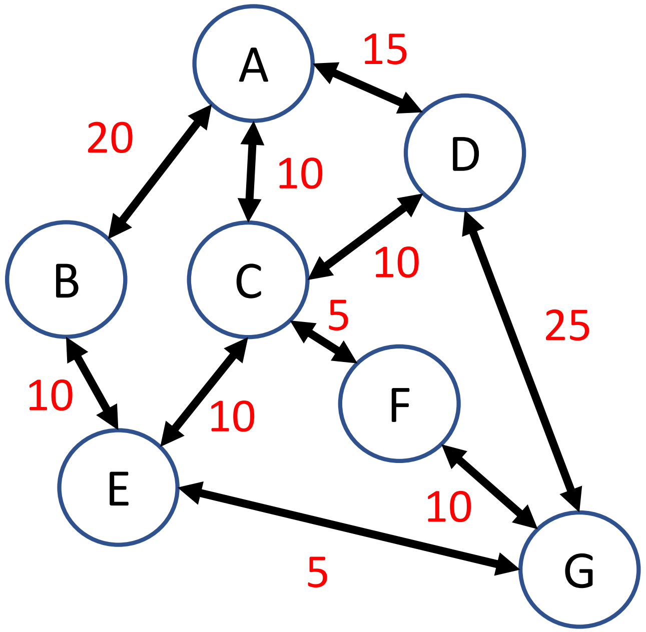
時間複雜度

- 初始化：$O(V)$，並建立資料結構 O(V)
- 取出並刪除最近頂點, V-1 次, $O(V-1)=O(V)$
  - 矩陣：O(V)
  - Binary heap： O(log_2V)
  - Fibonacci heap：O(log_2V)
- 修改 Distance, 2E 次, $O(2E)=O(E)$
  - 矩陣：O(1)
  - Binary heap： O(log_2V)
  - Fibonacci heap：O(1)
- 總計
  - 矩陣：$O(V^2)$
  - Binary heap：O($(E+V)log_2V$)=O($Elog_2V$)
  - Fibonacci heap：$O(E+Vlog_2V)$

Example Code

Mission

試著寫出 Prim's Algorithm！



Kruskal 與 Prim 比較

	Kruskal’s Algorithm	Prim’s Algorithm
初始化	每個點都視為獨立的最小生成樹	單一頂點開始逐步往外擴張
算法概要	挑最適合的邊納入	挑最適合的點納入
過程中	同時有許多最小生成樹	只有單一最小生成樹
挑選邊的方式	從 Crossing edges 中選最小的加以連接	
時間複雜度	$O(E\log_2 E)$	$O(E+V\log_2 V)$
優勢	邊少的狀況	邊多的狀況

Practice

Mission

LeetCode #1584. Min Cost to Connect All Points

You are given an array points representing integer coordinates of some points on a 2D-plane, where points[i] = [xi, yi].

The cost of connecting two points [xi, yi] and [xj, yj] is the manhattan distance between them: |xi - xj| + |yi - yj|, where |val| denotes the absolute value of val.

Return the minimum cost to make all points connected. All points are connected if there is exactly one simple path between any two points.

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