Minimum Spanning Tree

Lecture by baluteshih
Credit by zolution

Sprout

Disjoint Set

- 在介紹 Minimum Spanning Tree 之前，我想先介紹一個方便的資料結構。
- 中文稱為「並查集」。

- 他可以支援以下操作。
  - 1.詢問元素隸屬的集合。
  - 2.合併兩個集合。

- 這裡的集合在圖論上通常會被當成「連通塊」，這也代表著並查集擁有查詢任兩點是否連通的功能。

Disjoint Set

- 當我們想詢問元素隸屬的集合時，我們勢必得回傳這個集合被賦予的編號。

- 並查集的想法便是，他直接從每個集合中找出一個元素，並賦予他「老大」的地位。

- 每個元素只要找到自己的老大就可以知道自己在哪個集合上了。

Disjoint Set

- 令 boss[i] 是節點 i 的老大。
- 一開始每個人都自成一個集合，所以 boss[i] = i。

- 合併集合時，會指定兩個在不同集合的人，把這兩個人所屬的集合合併。
- 我們假設這兩個集合的老大分別是 a 跟 b。

- 只寫 boss[b] = a 就好了嗎？
- 根據我們的定義，可能不夠。

Disjoint Set

- 如果寫 boss[b] = a，那麼表示老大是 b 的所有其他節點都要把老大一起改成 a。
- 不妨我們把定義修正成「上級」

Disjoint Set

- 如果寫 boss[b] = a，那麼表示老大是 b 的所有其他節點都要把老大一起改成 a。
- 不妨我們把定義修正成「上級」

```
int find_boss(int x) {
    if (x == boss[x])
        return x;
    return find_boss(boss[x]);
}
```

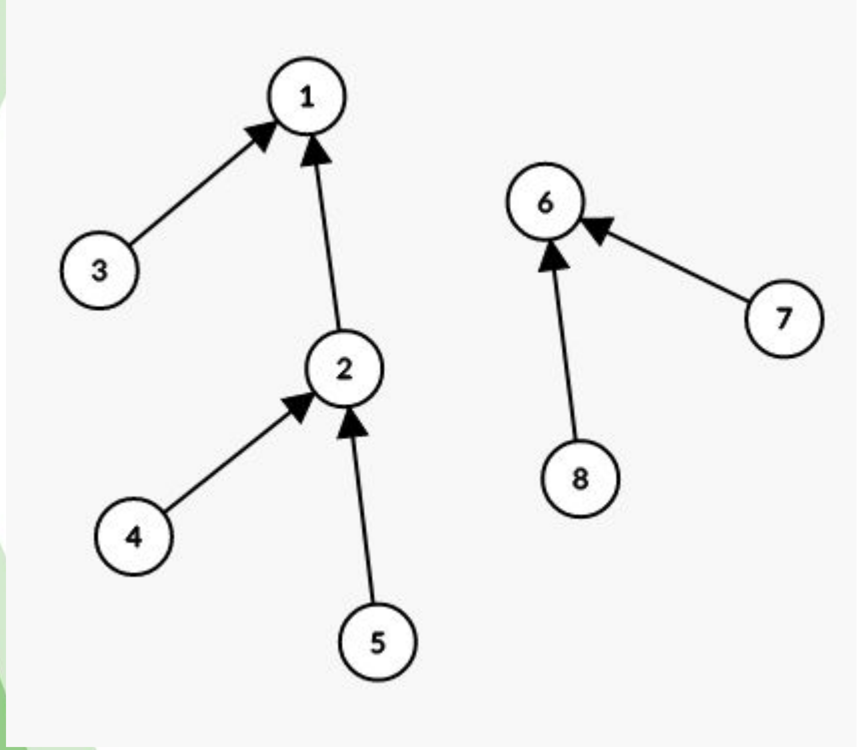
Disjoint Set

- 如果寫 boss[b] = a，那麼表示老大是 b 的所有其他節點都要把老大一起改成 a。
- 不妨我們把定義修正成「上級」

```
int find_boss(int x) {
    if (x == boss[x])
        return x;
    return find_boss(boss[x]);
}
```

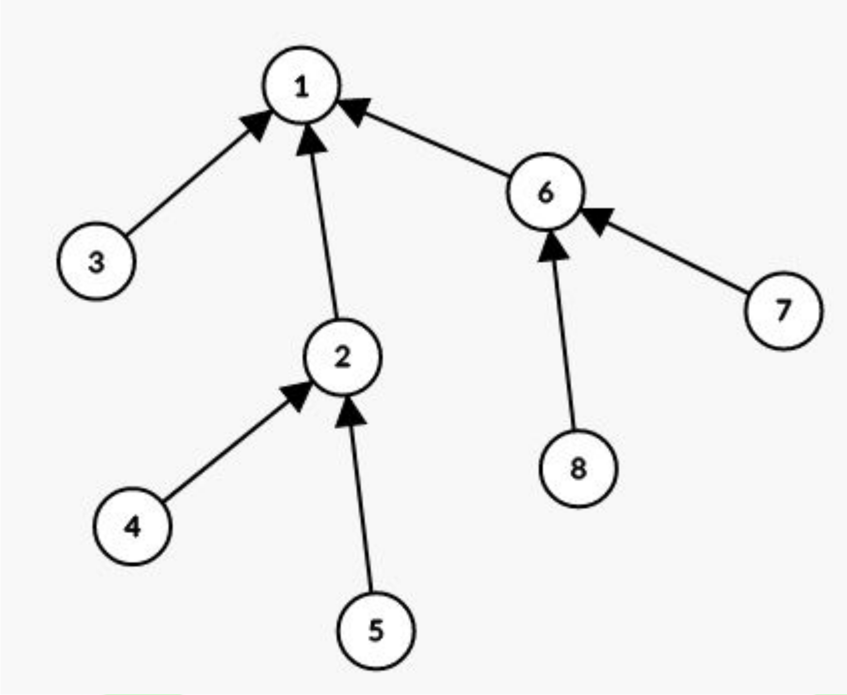
- 每次詢問老大的時候，就遞迴找到最上級的人！

Disjoint Set

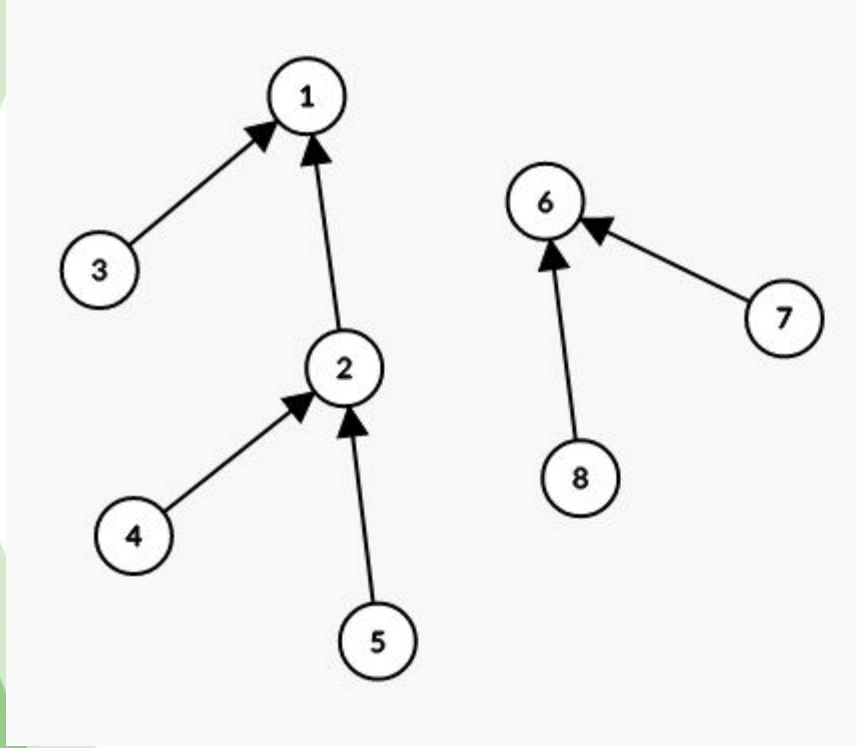


Merge 1,6

-------------->

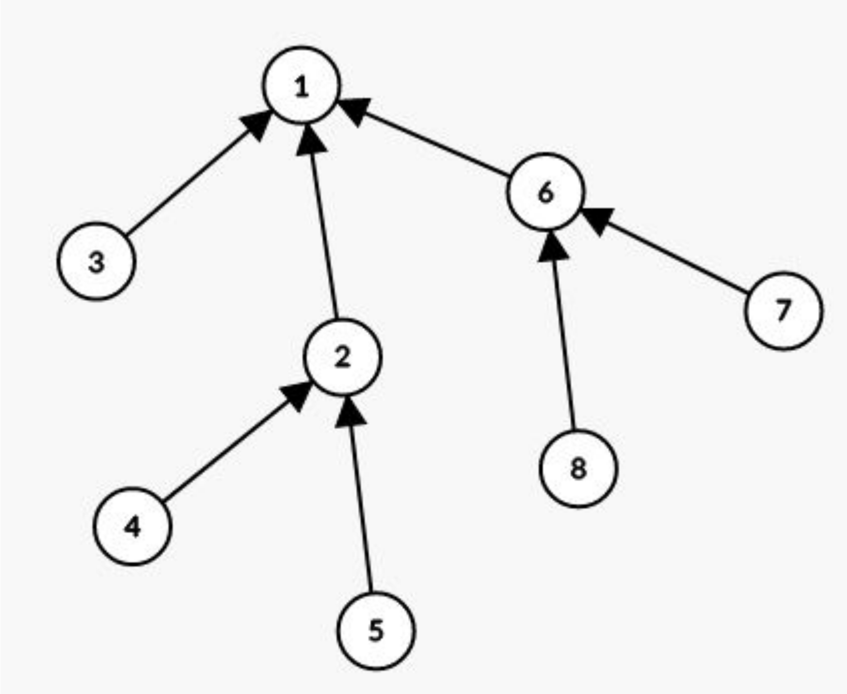


Disjoint Set



Merge 1,6

-------------->



「合併」本身 boss[6] = 1 -> O(1)!

Disjoint Set

- 可是查詢呢？
- 一條鏈的狀況下好像會變 O(N) QQ
- 有沒有辦法讓樹的高度不要太高？

- 我們來看看以下做法

Disjoint Set

- 一開始除了 boss 之外，也建立一個陣列叫 rank[i]
- rank[i] 代表「當自己是老大，那麼遞迴走到我的人最慘要走幾步」
- 簡單來說，就是紀錄自己這棵樹的高度，所以預設 rank[i] = 1

- 合併的時候，比較兩個老大的 rank，把 rank 比較小老大的接在比較大老大的下面。
- 這樣會多好呢？

Disjoint Set

- 這一招叫做「啟發式合併」
- 由於要讓一棵樹的 rank 提升，就必須要有一個 rank 一樣大的樹。
- 所以要讓最大 rank 提升至 x，就至少要有 2^x 次合併。
- -----> 樹高不超過 logN！
- 所以我們 find_boss 的複雜度就降為 O(log N) 了！

- 能不能再好一些？

Disjoint Set

- 其實仔細想想，我們把 boss 的定義改成上級，但那只是因為我們希望讓合併 O(1) 而已。
- 實際上我們只需要「老大是誰」的資訊就可以了。
- 那，何不妨耍一點小聰明額外把一些節點指到定位呢？

Disjoint Set

- 其實仔細想想，我們把 boss 的定義改成上級，但那只是因為我們希望讓合併 O(1) 而已。
- 實際上我們只需要「老大是誰」的資訊就可以了。
- 那，何不妨耍一點小聰明額外把一些節點指到定位呢？

```
int find_boss(int x) {
    if (x == boss[x])
        return x;
    return find_boss(boss[x]);
}
```

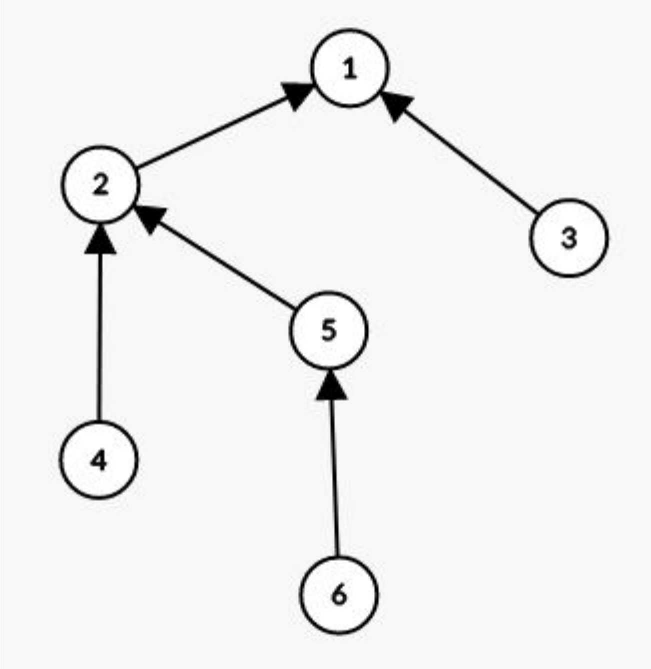
Disjoint Set

- 其實仔細想想，我們把 boss 的定義改成上級，但那只是因為我們希望讓合併 O(1) 而已。
- 實際上我們只需要「老大是誰」的資訊就可以了。
- 那，何不妨耍一點小聰明額外把一些節點指到定位呢？

```
int find_boss(int x) {
    if (x == boss[x])
        return x;
    return boss[x] = find_boss(boss[x]);
}
```

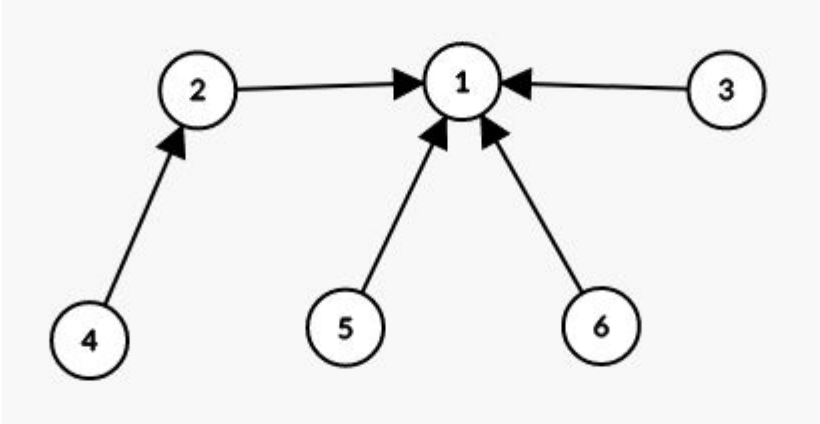
- 利用遞迴的機制直接順路把 boss 指上去！

Disjoint Set



find_boss(6)

--------------->



Disjoint Set

- 這一個做法又被稱為「路徑壓縮」
- 這樣不是只是偷吃步嗎？是能多快？
- 超快！
- 有人證明出來，這樣的複雜度平均起來，每次查詢只需要 O(log_{2+f/n} n)
- 其中 f 是查詢次數。

- 證明很難，有興趣可以上網查查

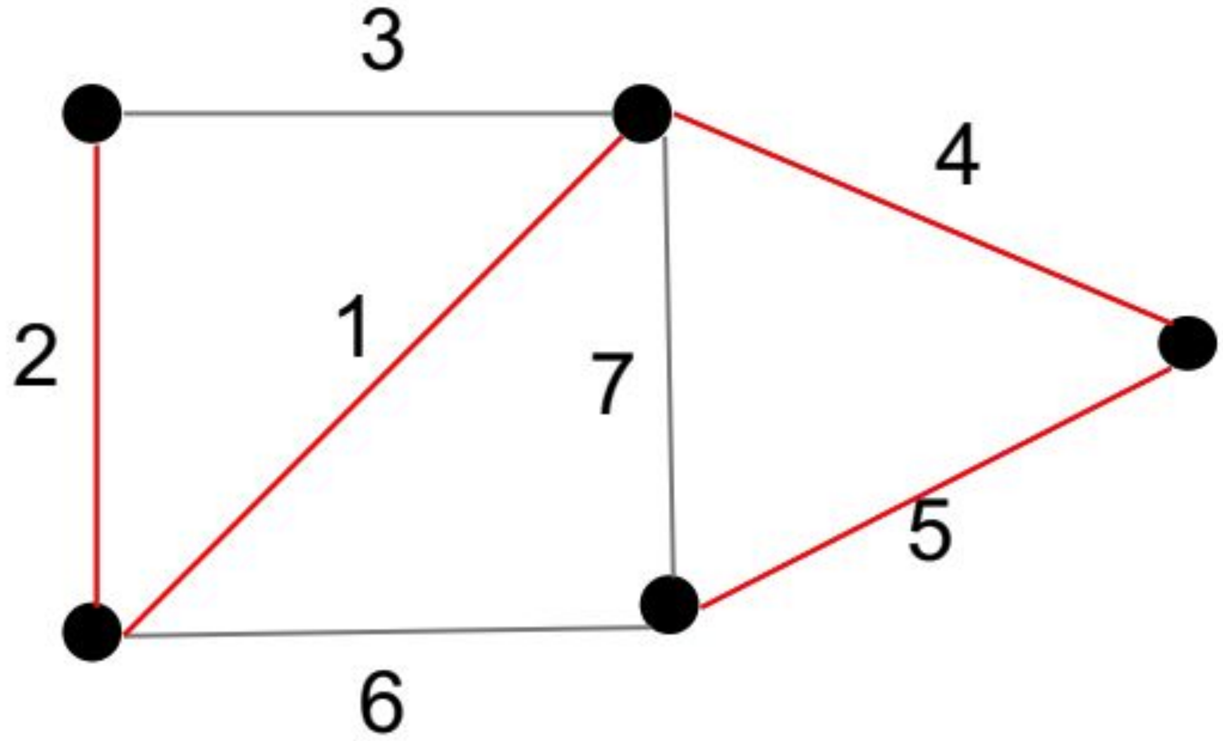
Disjoint Set

- 那如果加上前面的啟發式合併呢？
- 平均查詢複雜度會變成 O(α(N))
- α(N)？
- α(N) 是阿克曼函數 A(N,N) 的反函數。
- A(4,4) 大約是 2^2^10^19729，可見其增長之快
- 反函數之後當然變超級小，跟常數沒兩樣！
- 證明也很難(X

- ~~不過由於路徑壓縮就夠優秀了，所以只寫路徑壓縮也是可以。~~
- 題外話，啟發式合併會在「可回溯並查集」被強迫用到。

Minimum Spanning Tree

- 最小生成樹



Minimum Spanning Tree

- N 個點的連通圖上，選出一些邊留著使得他還是連通的
  - 我們先叫他「生成圖」好了
  - 假設所有邊權都不相同
- 如果我們要找的是最小的一張生成圖
  - 最小：生成圖上的所有邊權總和最小
- 性質一：他一定會是生成「樹」
  - 想一想：如果不是「樹」？

Minimum Spanning Tree

- 性質二:Cycle Property

- C 是圖上的一個環，e 是 C 上權重唯一最大的一條邊
- 那 e 必然不會是MST的一部分

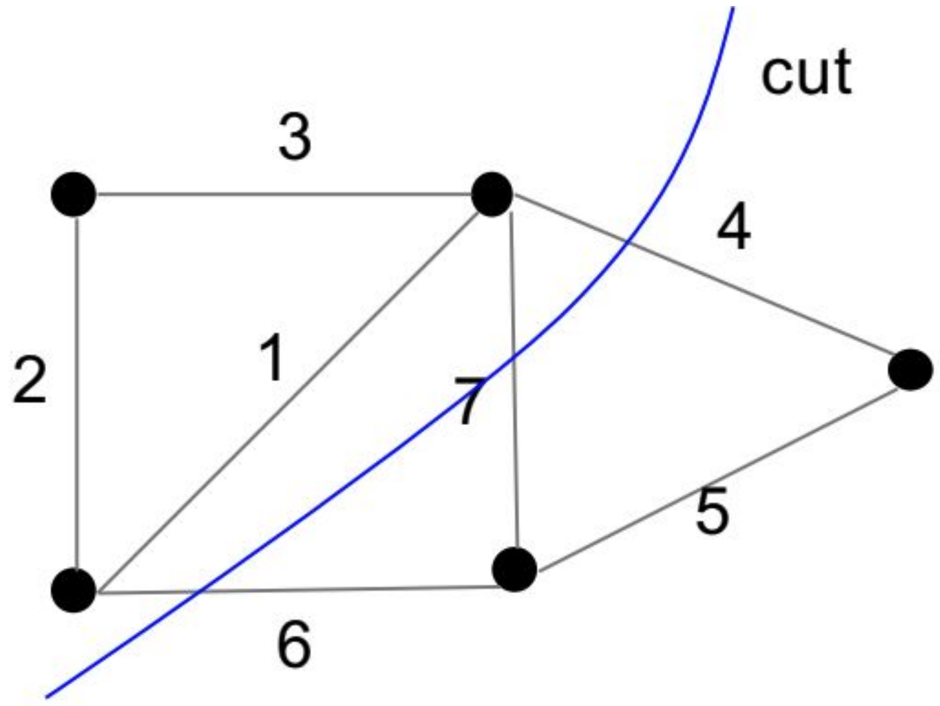
- 證明？

Minimum Spanning Tree

- 反證法
- 假設 e 在 MST 裡面
  - 把 e 移開會讓 MST 分裂成兩塊
  - 但 C 上會有另外一條邊可以把 MST 接回來，同時 cost 還比較小
- 所以原 MST 不是最小權重，矛盾！
- 想一想：如果有多條最大的邊該怎麼修正這個性質？

Minimum Spanning Tree

- 性質三:Cut Property
- 先來定義什麼是 Cut



Minimum Spanning Tree

- 性質三:Cut Property
- 任一個 Cut 上唯一最小權重的邊會是 MST 的一部分

- 證明？

Minimum Spanning Tree

- 性質三:Cut Property
- 反證法

- 假設 e 不是 MST 的一部分
  - 那把 e 加進去就會形成環
  - 就可以把另一條權重比 e 大的邊拔掉
  - 但這樣 MST 權重變更小
  - 矛盾

- 想一想:如果有多條最小的邊該怎麼修正這個性質？

Minimum Spanning Tree

- 啊所以到底要怎麼找他？

Kruskal Algorithm

- 依序從最小的邊開始試
- 如果那個邊不會形成環就可以
- Disjoint set

- Complexity: O(ElogE)

- 為什麼這是對的？

Kruskal Algorithm

- Case 1: 如果加了這條邊形成環
- Case 2: 加了這條邊不會形成環

Kruskal Algorithm

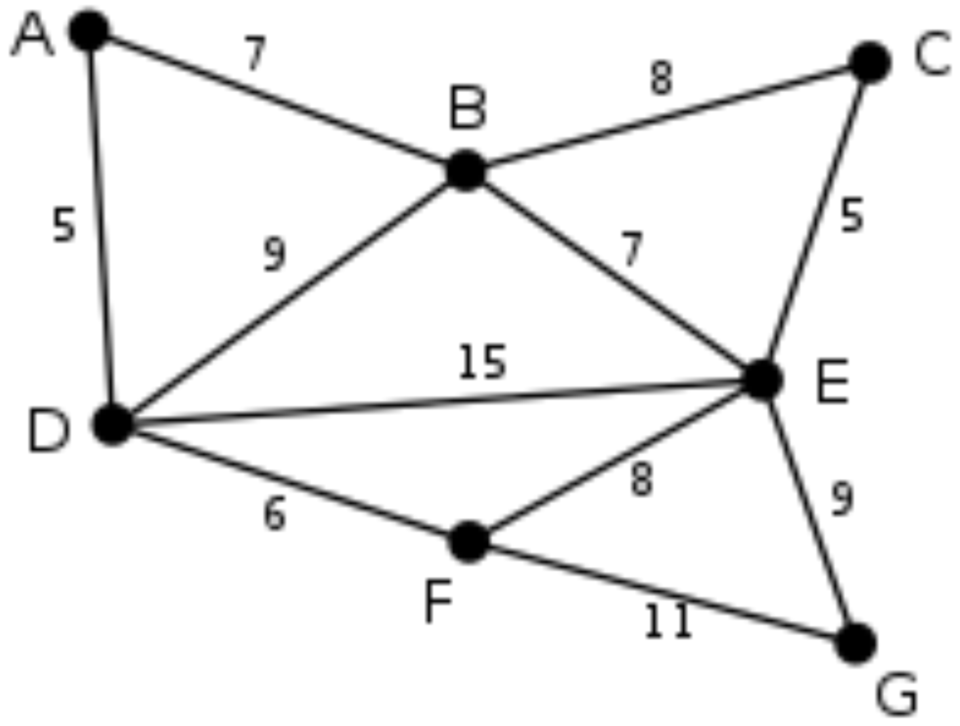
- Case 1: 如果加了這條邊形成環
  - 那這條邊會是這個環上的最大邊
  - 根據 Cycle property, 這條邊不會是 MST 的一部分
- Case 2: 加了這條邊不會形成環

Kruskal Algorithm

- Case 1: 如果加了這條邊形成環
  - 那這條邊會是這個環上的最大邊
  - 根據 Cycle property，這條邊不會是 MST 的一部分
- Case 2: 加了這條邊不會形成環
  - 那這條邊是條橋，連接左右兩棵樹
  - 根據 Cut Property，因為這條邊是這個 cut 上最小的邊
  - 所以這條邊會是 MST 的一部分

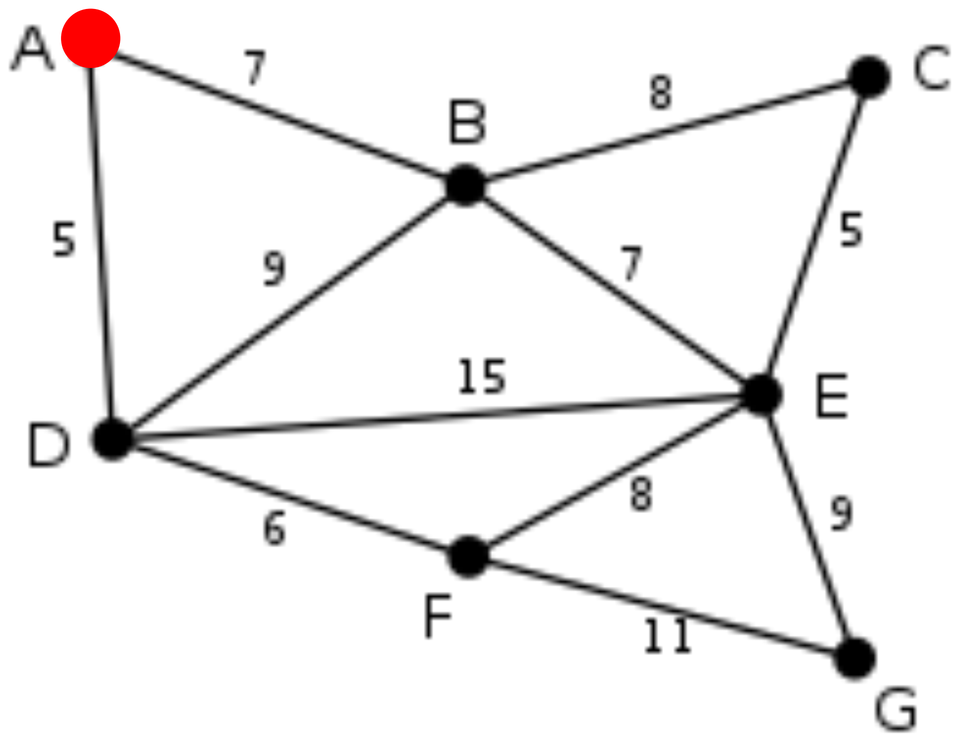
Prim’s Algorithm

- 質數演算法
- 從任意一個點開始，每次從cost最小的點找可以延伸的最小邊



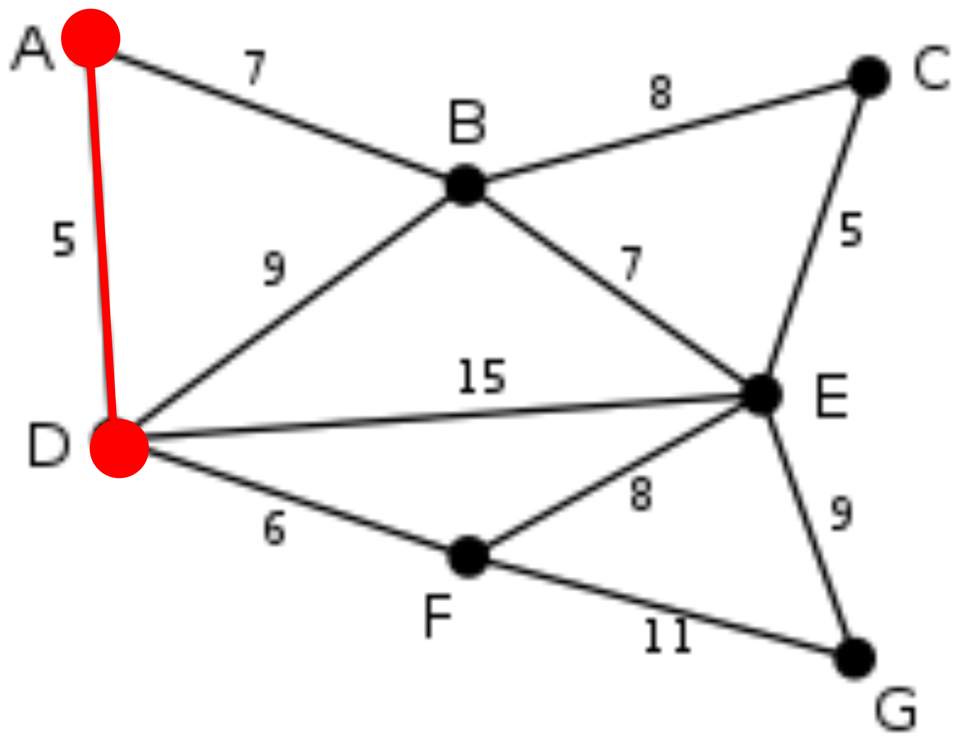
Prim’s Algorithm

- 質數演算法
- 從任意一個點開始，每次從cost最小的點找可以延伸的最小邊



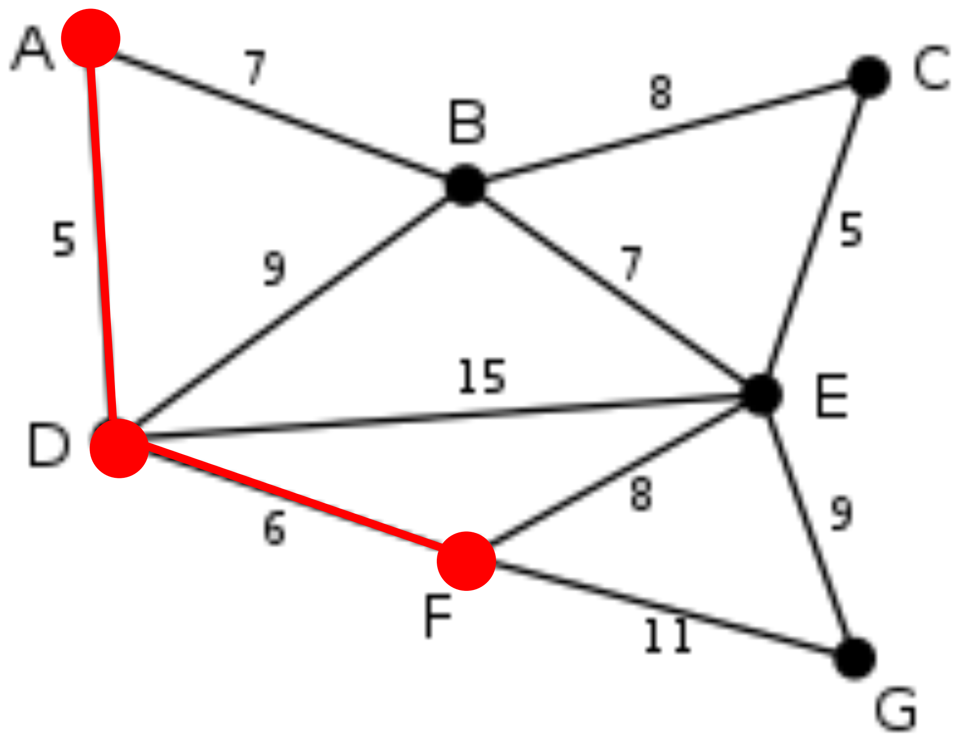
Prim's Algorithm

- 質數演算法
- 從任意一個點開始，每次從cost最小的點找可以延伸的最小邊



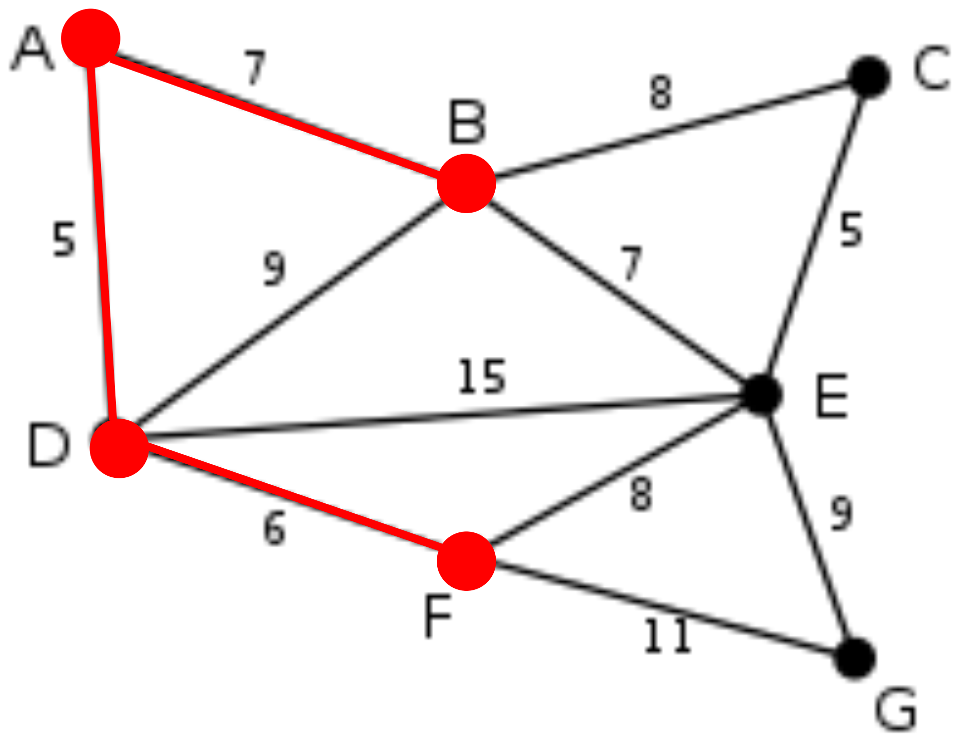
Prim’s Algorithm

- 質數演算法
- 從任意一個點開始，每次從cost最小的點找可以延伸的最小邊



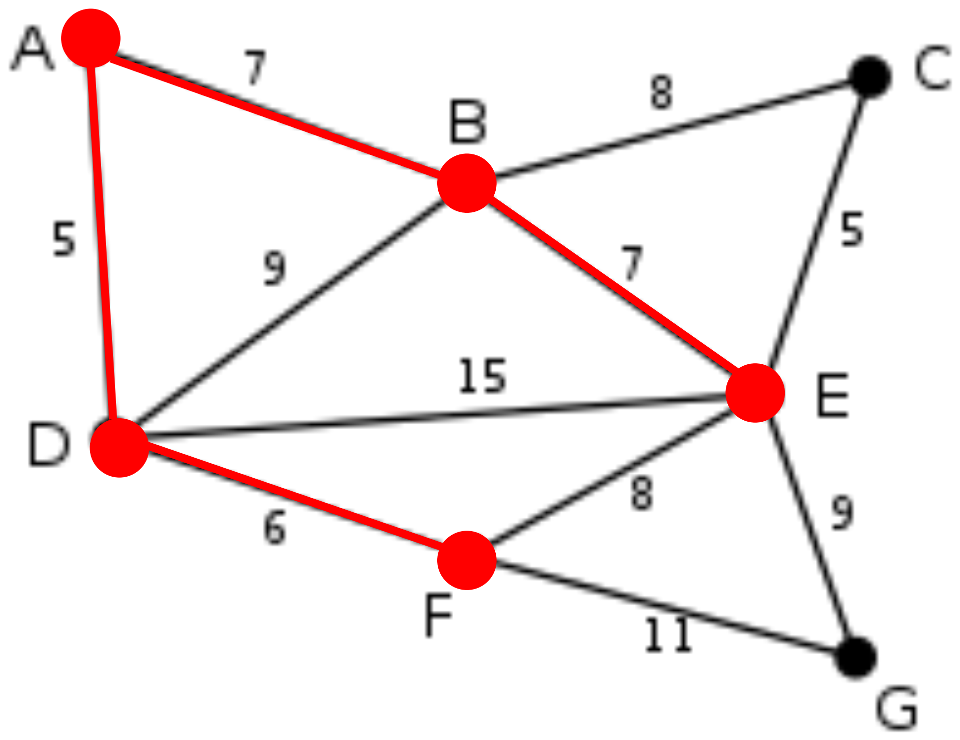
Prim's Algorithm

- 質數演算法
- 從任意一個點開始，每次從cost最小的點找可以延伸的最小邊



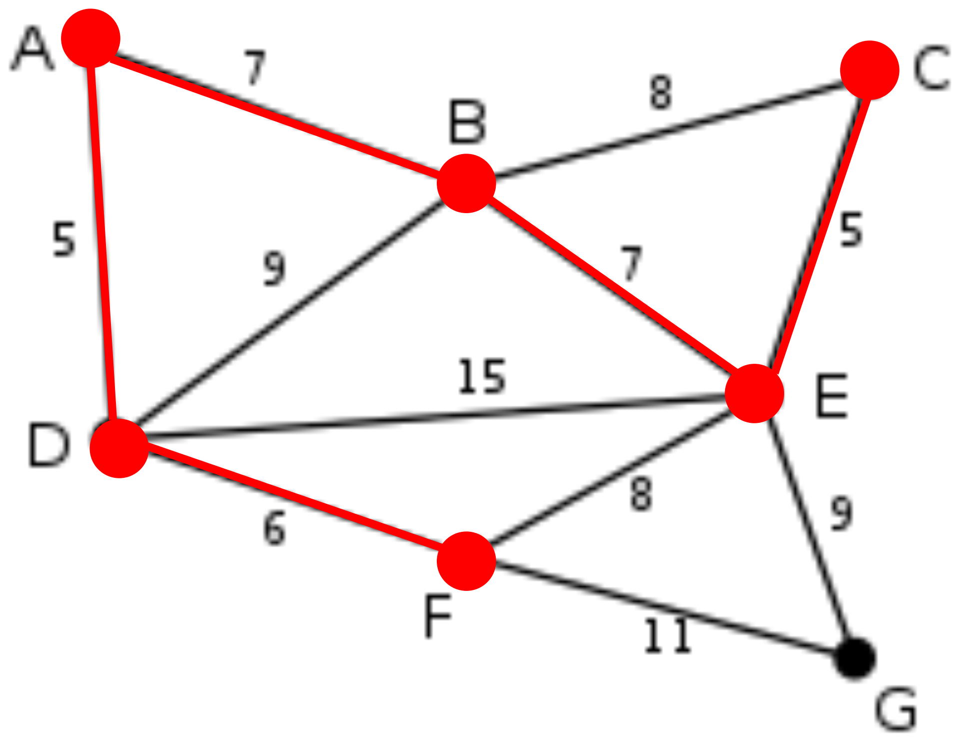
Prim's Algorithm

- 質數演算法
- 從任意一個點開始，每次從cost最小的點找可以延伸的最小邊



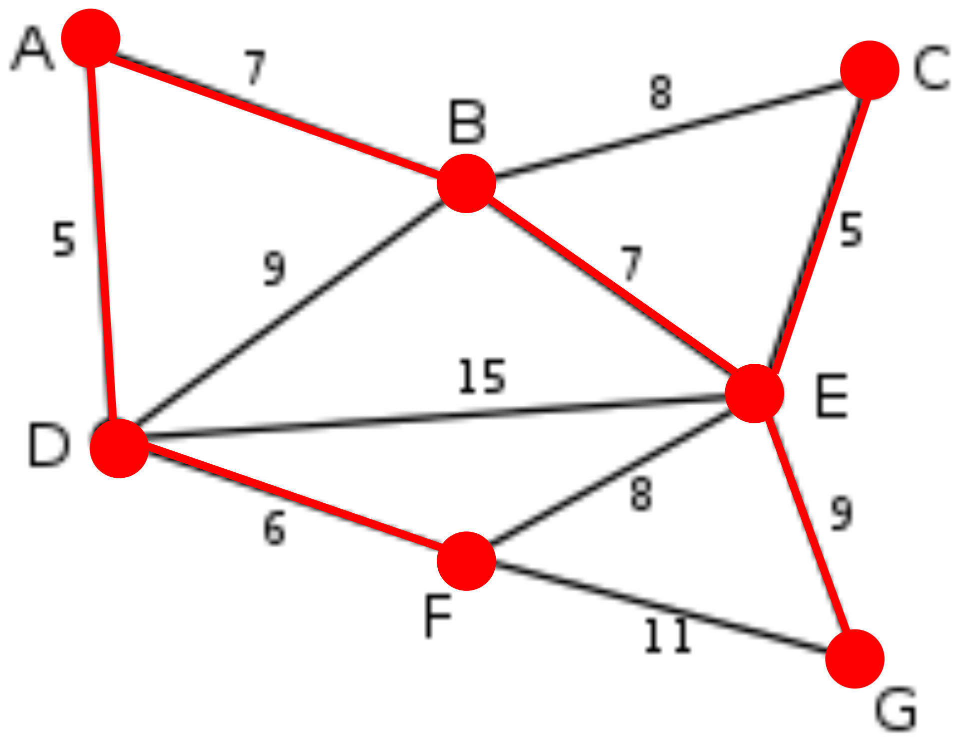
Prim’s Algorithm

- 質數演算法
- 從任意一個點開始，每次從cost最小的點找可以延伸的最小邊



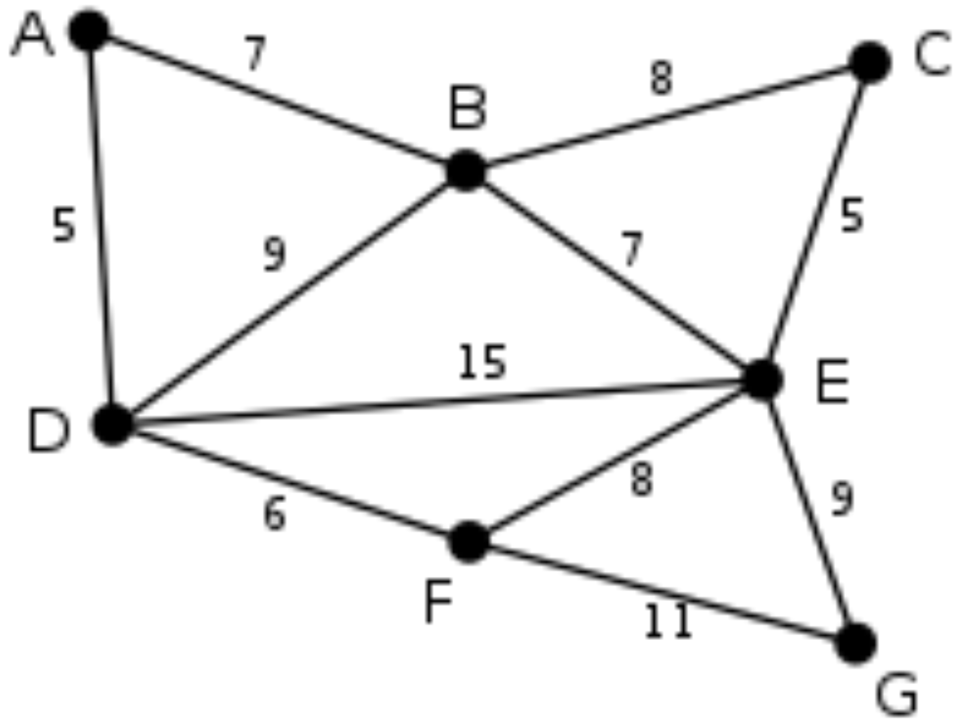
Prim’s Algorithm

- 質數演算法
- 從任意一個點開始，每次從cost最小的點找可以延伸的最小邊



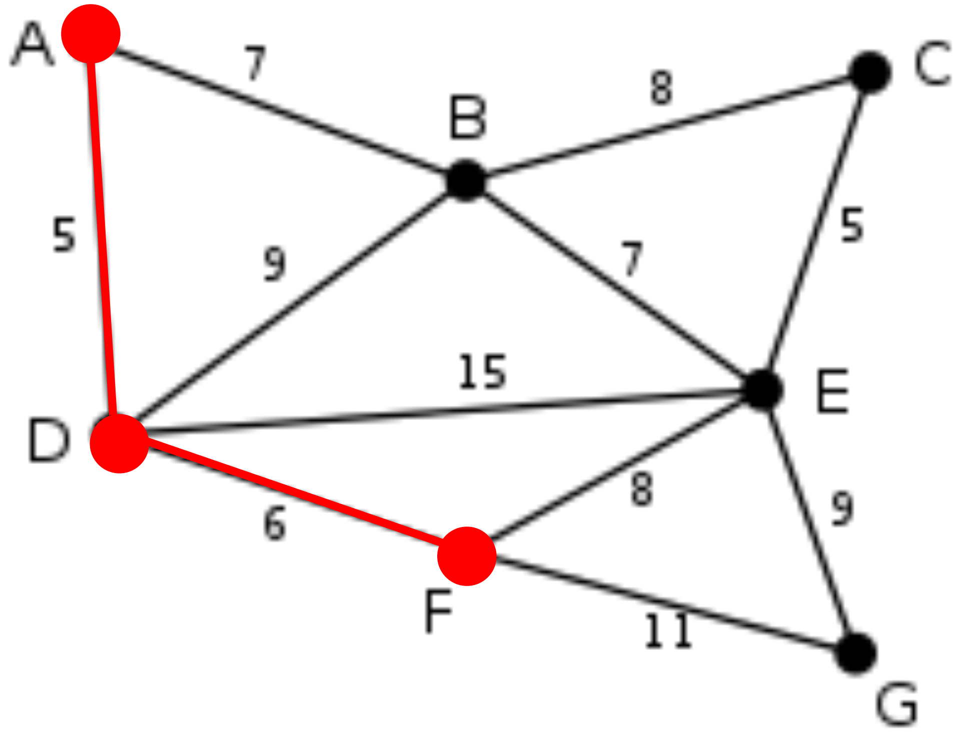
Prim’s Algorithm

- 質數演算法
- 證明？



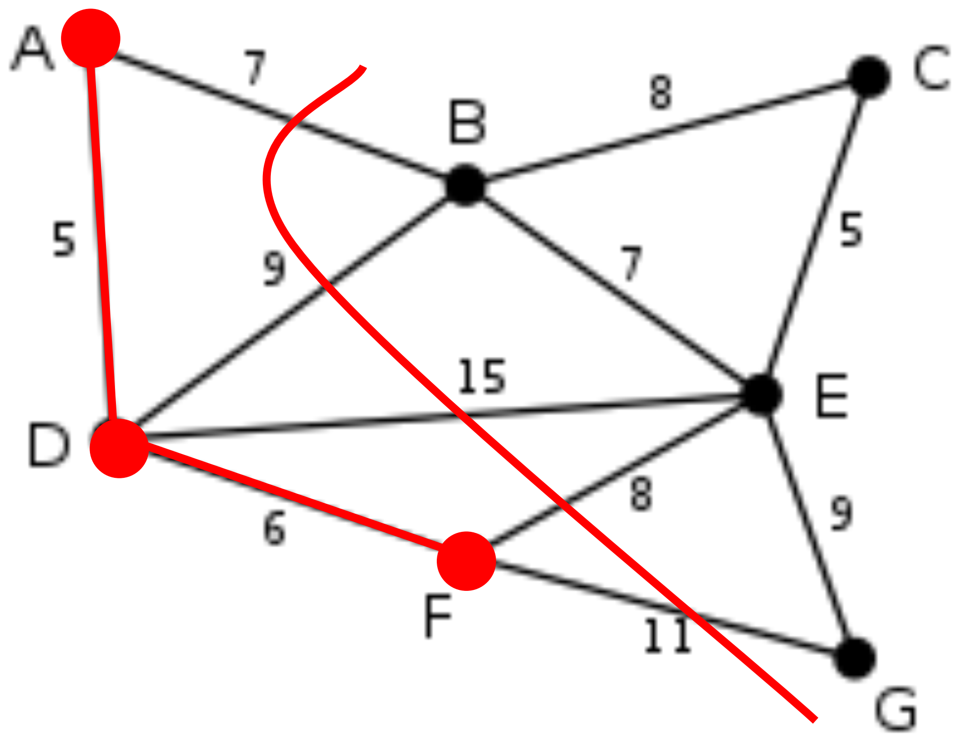
Prim's Algorithm

- 質數演算法
- 證明？Cut Property!



Prim's Algorithm

- 質數演算法
- 證明？Cut Property!



Prim’s Algorithm

- 質數演算法
- 時間複雜度:O(E+V^2) 或 Priority Queue O((E + V) lg (E + V)) (How?)