Graph

圖論

大綱

圖的介紹

圖的存法

圖的搜索

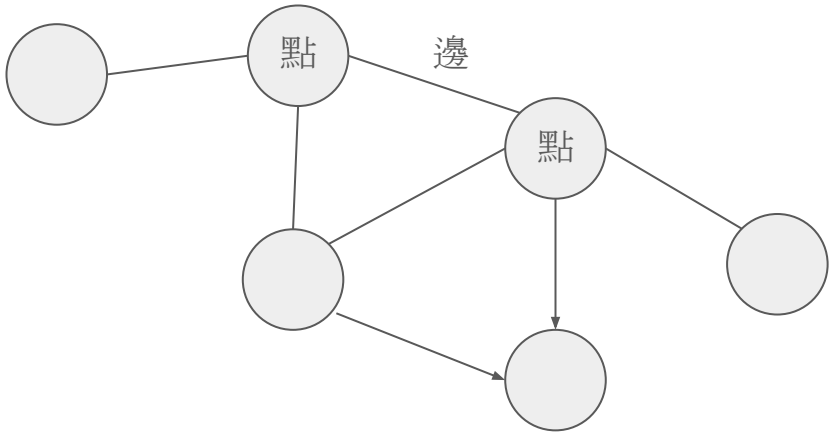
樹

有向無環圖(DAG)

一般圖

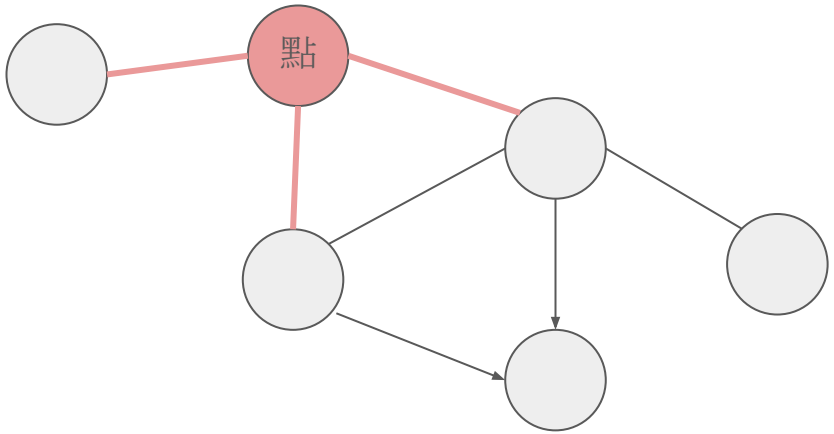
圖的介紹

- 由點(圓圈) 跟 邊(線條) 所成的結構



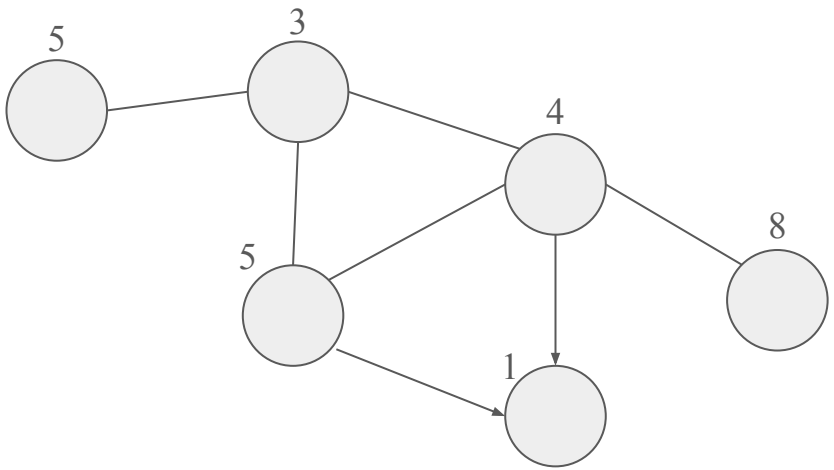
點 Node

- 一個點可以連出去很多邊
- 連出去多少邊稱為該點的Degree



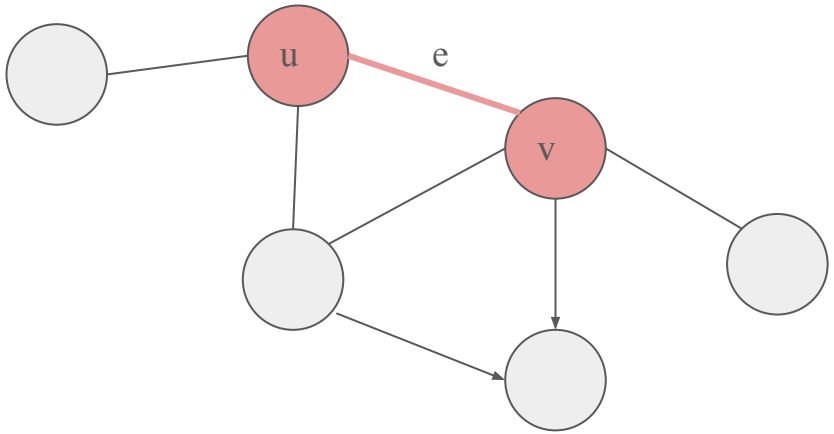
點 Node

- 點上有數字, 稱為點權重

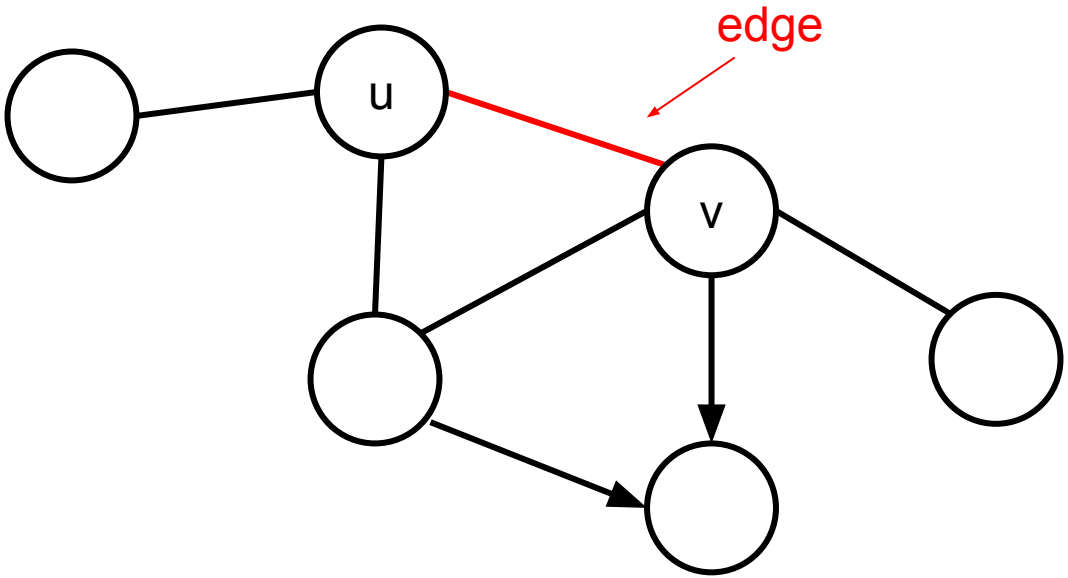


邊 Edge

- 一條邊只能連到兩個點(相同或相異)

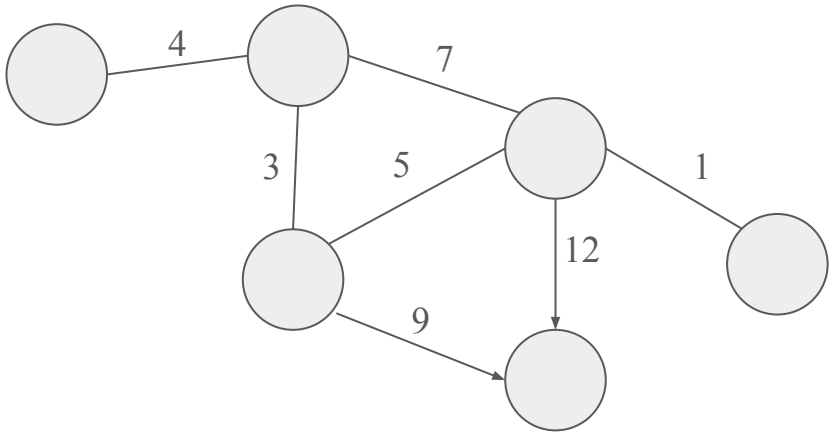


邊 Edge



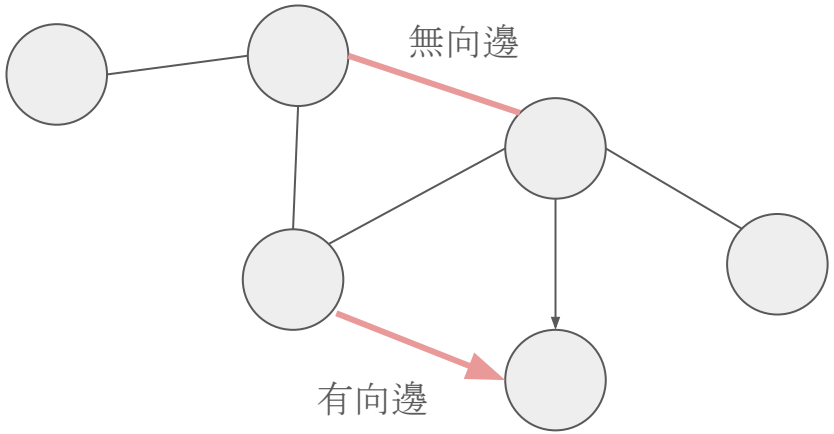
邊

- 邊上有數字, 稱為邊權重



邊

- 有方向的邊叫有向邊
- 沒方向的邊叫無向邊

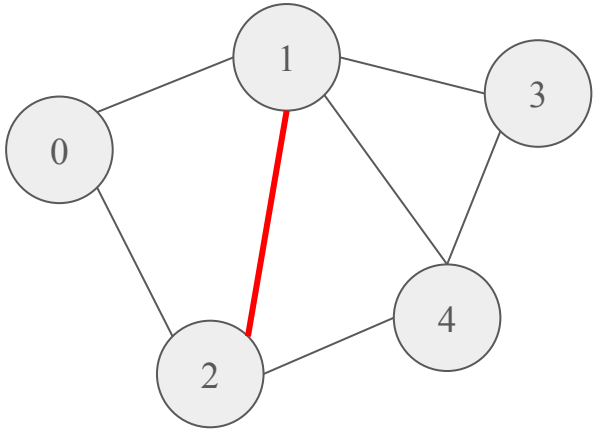


圖的存法

- 鄰接矩陣 Adjacency matrix
- 鄰接串列 Adjacency list
- 邊列表 Edge list

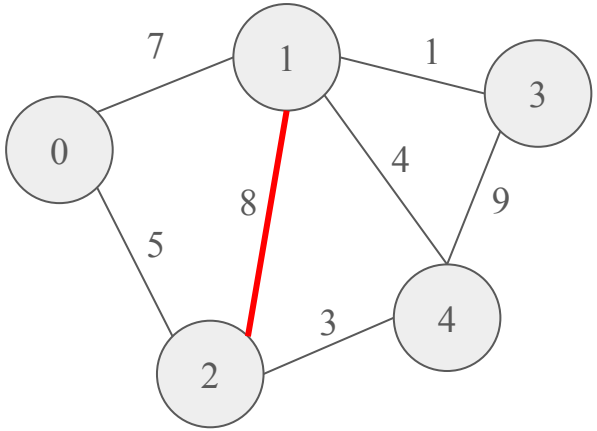
鄰接矩陣 Adjacency Matrix

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	1	1
2	1	1	0	0	1
3	0	1	0	0	1
4	0	1	1	1	0



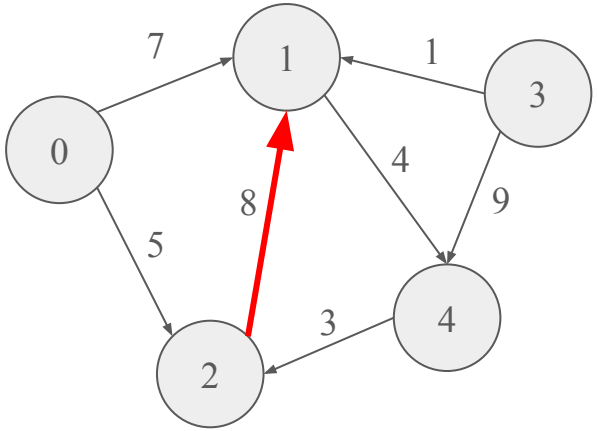
鄰接矩陣 存無向邊帶權重

	0	1	2	3	4
0	0	7	5	0	0
1	7	0	8	1	4
2	5	8	0	0	3
3	0	1	0	0	9
4	0	4	3	9	0



鄰接矩陣 存有向邊帶權重

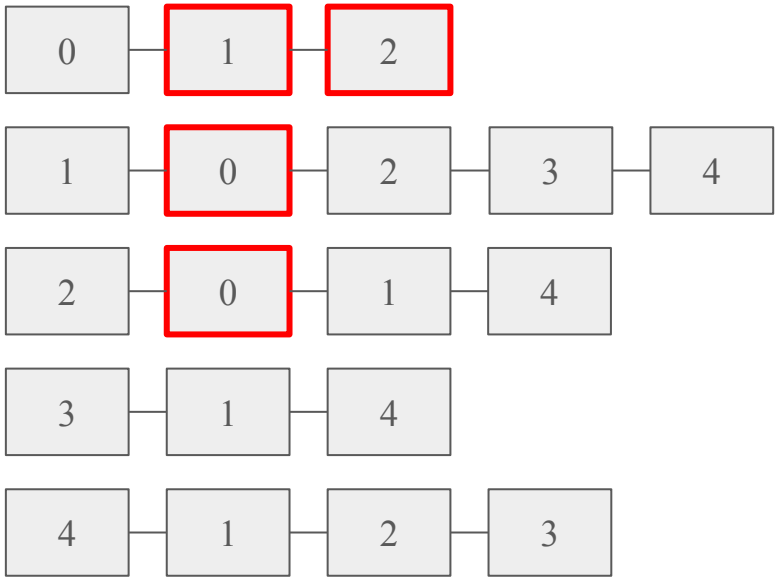
	0	1	2	3	4
0	0	7	5	0	0
1	0	0	0	0	4
2	0	8	0	0	0
3	0	1	0	0	9
4	0	0	3	0	0

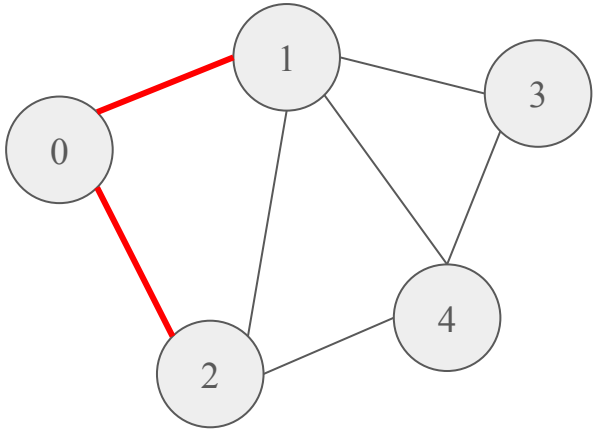


鄰接矩陣 存有向邊帶權重 Code

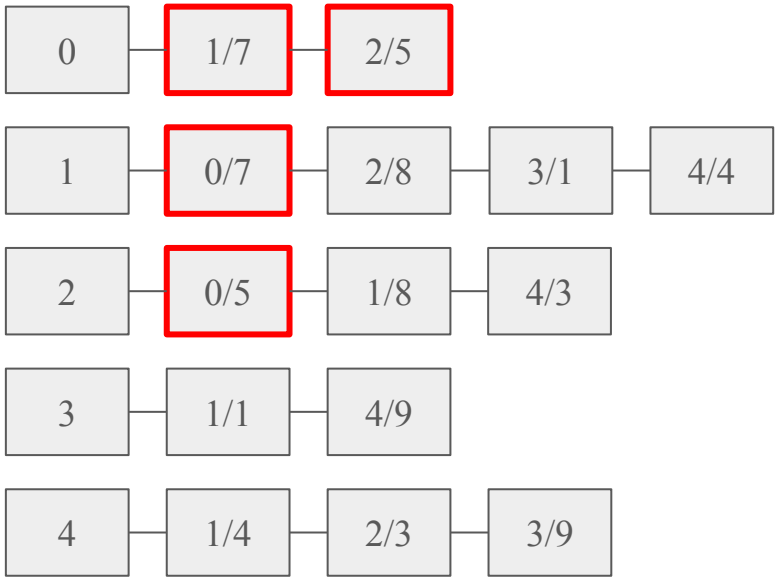
```
 5 const int MAXN = 1e3 + 5;
 6 int Graph[MAXN][MAXN];
 7 void addEdge(int u, int v, int w) {
 8     Graph[u][v] = w;
 9     Graph[v][u] = w; // 無向邊記得加
10 }
```

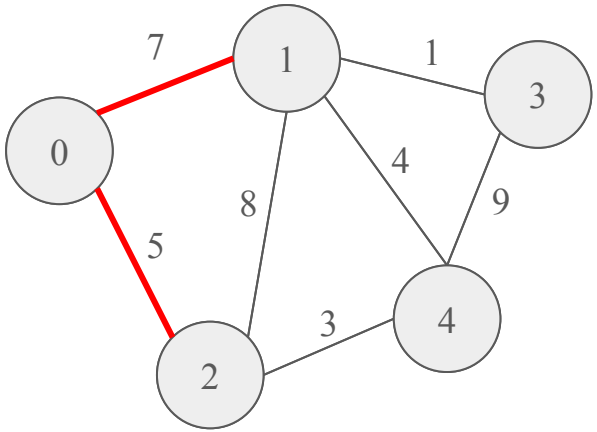
鄰接串列 Adjacency List



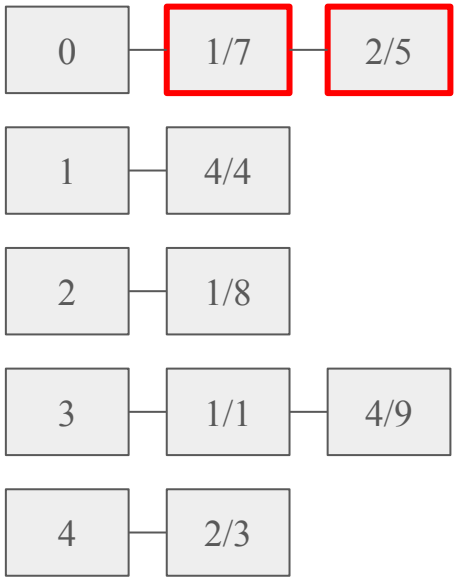


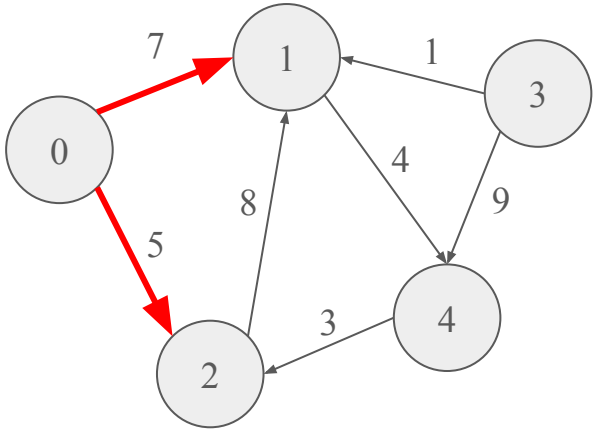
鄰接串列 存無向邊帶權重





鄰接串列 存有向邊帶權重



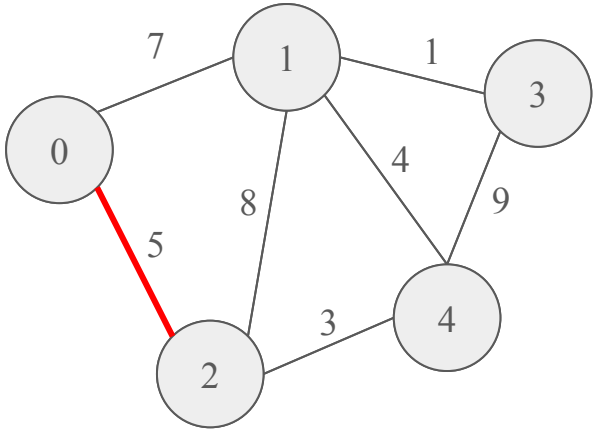


鄰接串列 存有向邊帶權重 Code

```cpp
const int MAXN = 1e3 + 5;
struct Edge {
    int v, w;
};
vector<Edge> Graph[MAXN];
void addEdge(int u, int v, int w) {
    Graph[u].push_back({v, w});
    Graph[v].push_back({u, w}); // 無向邊記得加
}
```

邊列表 存邊權重

u	v	w
0	1	7
0	2	5
1	2	8
1	3	1
1	4	4
2	4	3
3	4	9



邊列表 存邊權重 Code

```
 5 struct Edge {
 6     int u, v, w;
 7 };
 8 vector<Edge> EdgeList
 9 void addEdge(int u, int v, int w) {
10     EdgeList.push_back({u, v, w});
11 }
```

圖的存法比較

- 鄰接串列 Adjacency list
  - 空間複雜度 O(V + E)
  - 列舉該點的每條邊 O(d), d : 該點的degree
- 鄰接矩陣 Adjacency matrix
  - 空間複雜度 O(V^2)
  - 列舉該點的每條邊 O(V)
- 邊列表 Edge list
  - 空間複雜度 O(E)

圖的存法比較

- 每一種圖的存法都有不一樣的時空複雜度
- 每一種演算法適合的圖存法也不相同
- 選擇適合的存法是重要的第一步

圖的搜索

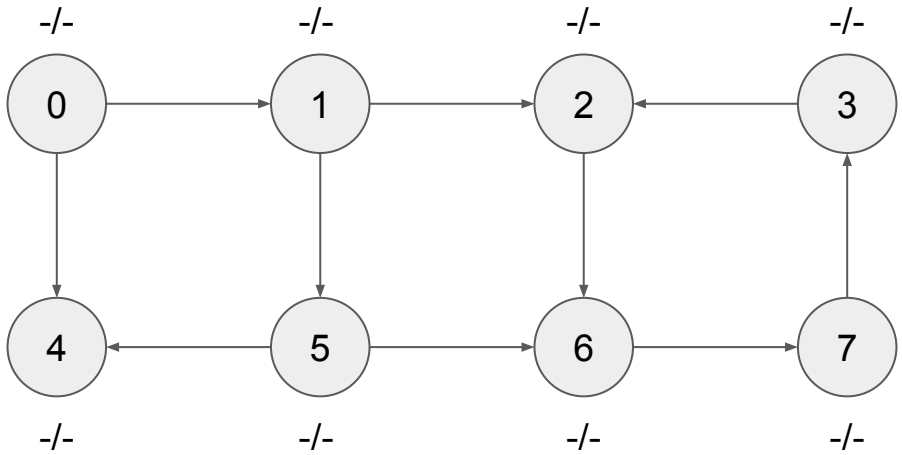
- 深度優先搜索(DFS)
- 廣度優先搜索(BFS)

深度優先搜索 DFS

- 又名 **D**epth **F**irst **S**earch
- 像人在走迷宮
  - 先選一條路走
  - 走到死路退回來, 重選一條路
- 重要的資訊
  - In Stamp 入戳章
  - Out Stamp 出戳章
  - 這兩個資訊可以來解決很多問題
- 大多用遞迴實作, 也可以用Stack

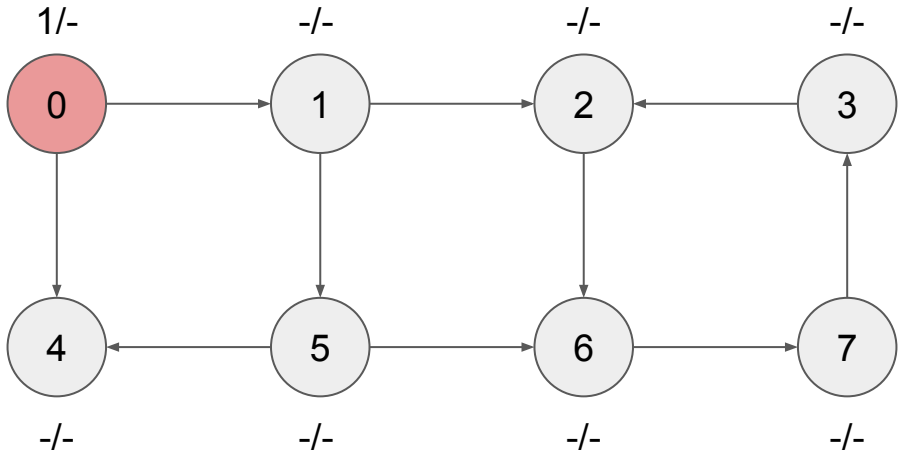
深度優先搜索 DFS

對這張圖做DFS



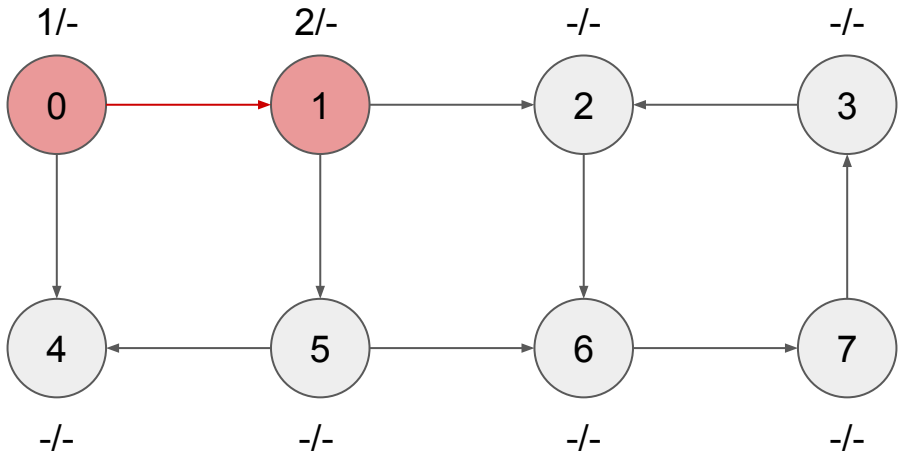
深度優先搜索 DFS

起點為節點0



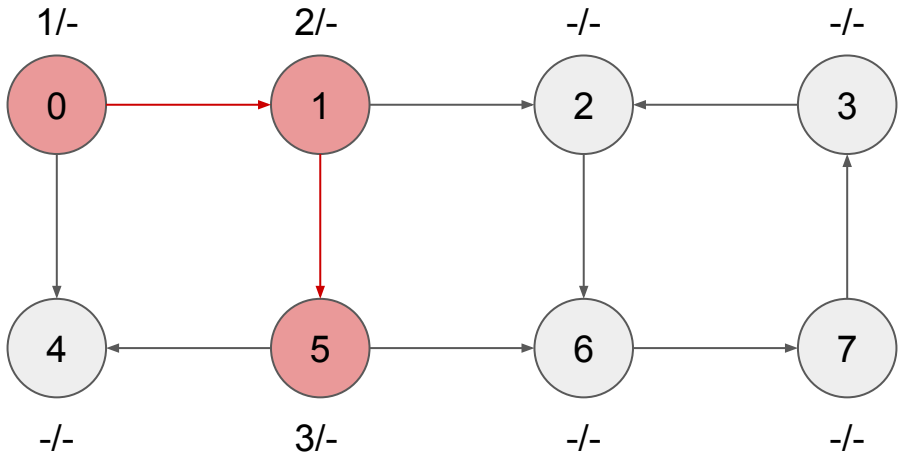
深度優先搜索 DFS

選出節點0連出去的邊中還未走訪的點 (節點1)



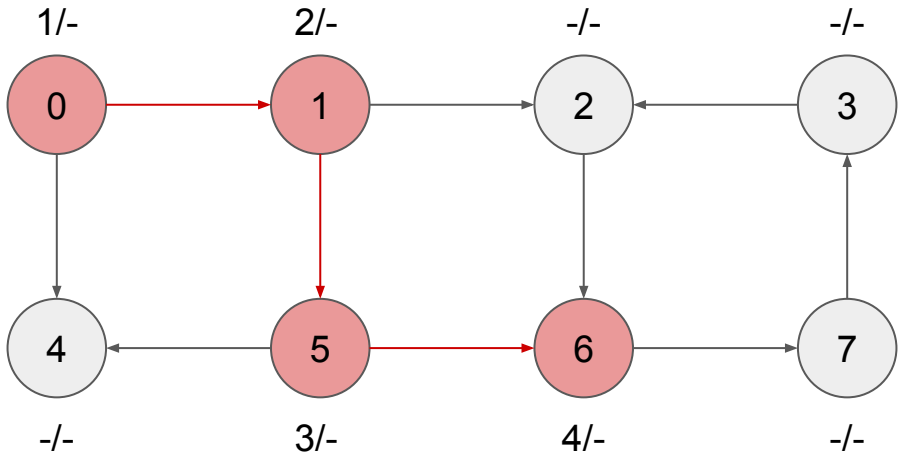
深度優先搜索 DFS

選出節點1連出去的邊中還未走訪的點 (節點5)



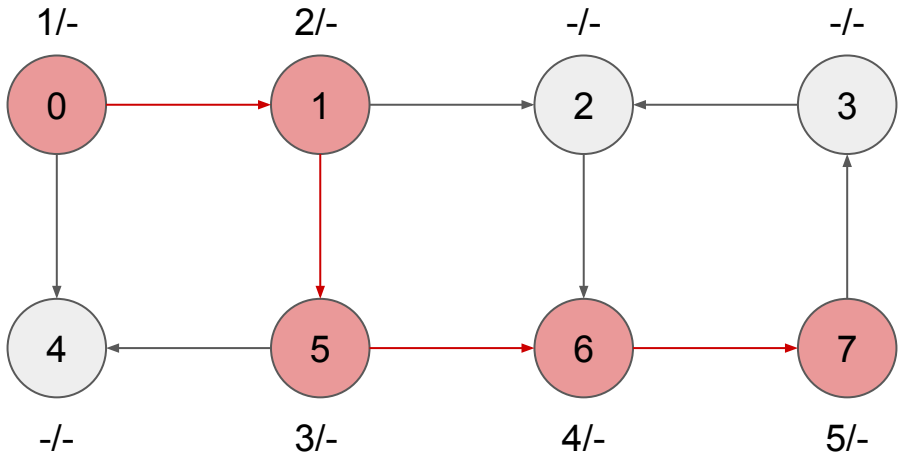
深度優先搜索 DFS

選出節點5連出去的邊中還未走訪的點 (節點6)



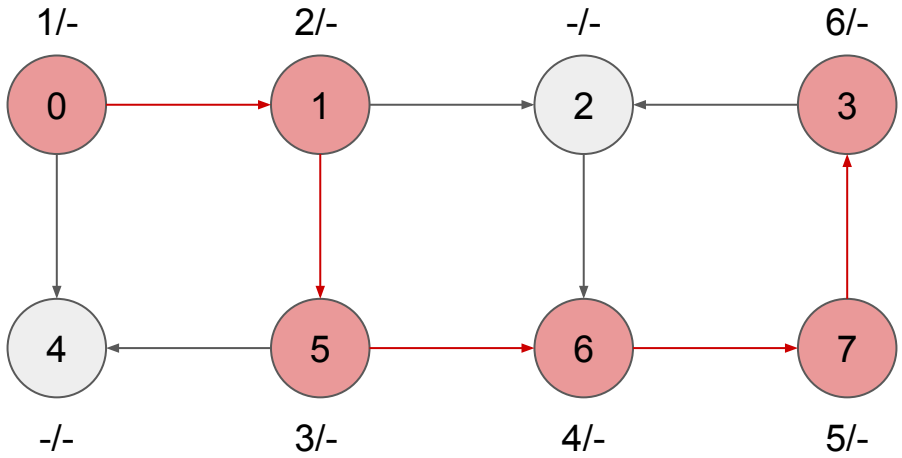
深度優先搜索 DFS

選出節點6連出去的邊中還未走訪的點 (節點7)



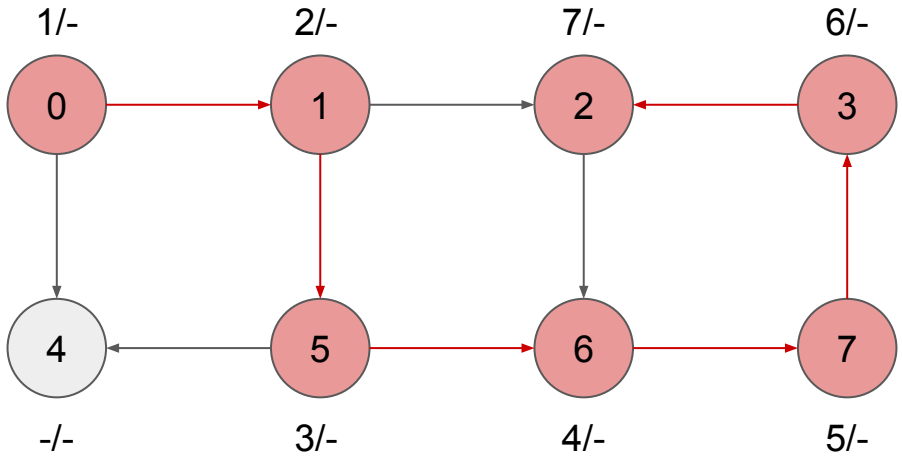
深度優先搜索 DFS

選出節點7連出去的邊中還未走訪的點 (節點3)



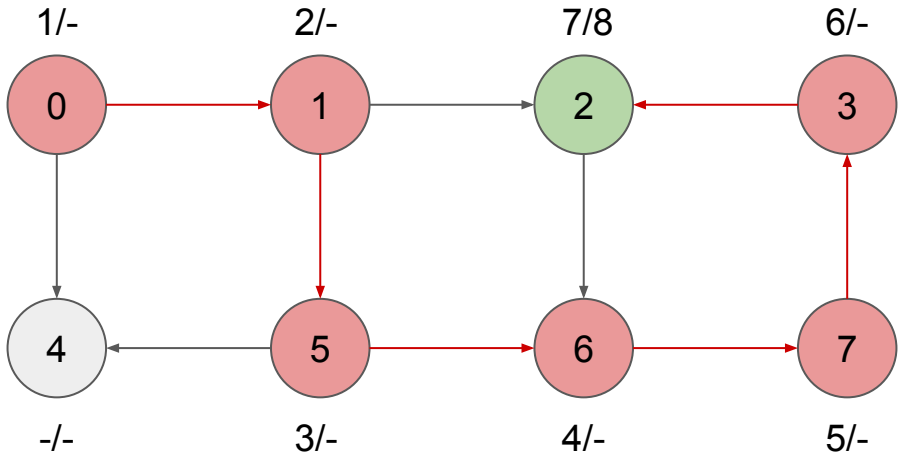
深度優先搜索 DFS

選出節點3連出去的邊中還未走訪的點 (節點2)



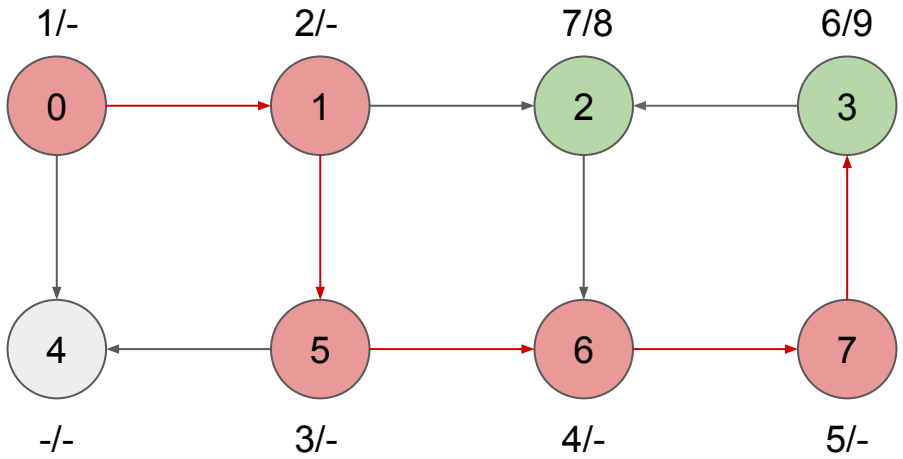
深度優先搜索 DFS

節點2連出去的邊都已經被走訪過了，所以節點 2走訪完了。



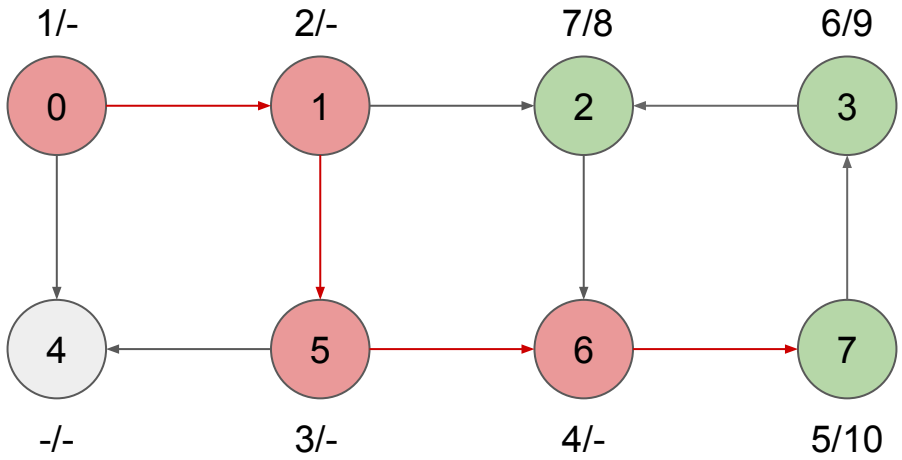
深度優先搜索 DFS

節點3連出去的邊都已經被走訪過了，所以節點 3走訪完了。



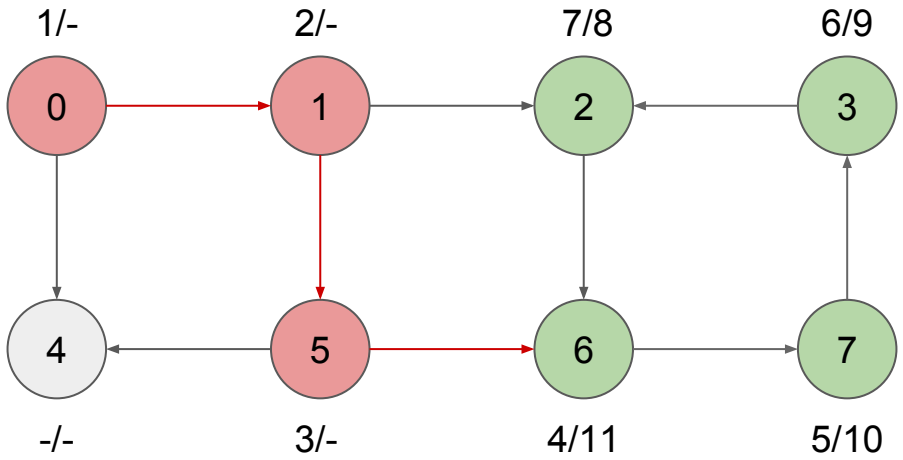
深度優先搜索 DFS

節點7連出去的邊都已經被走訪過了，所以節點 7走訪完了。



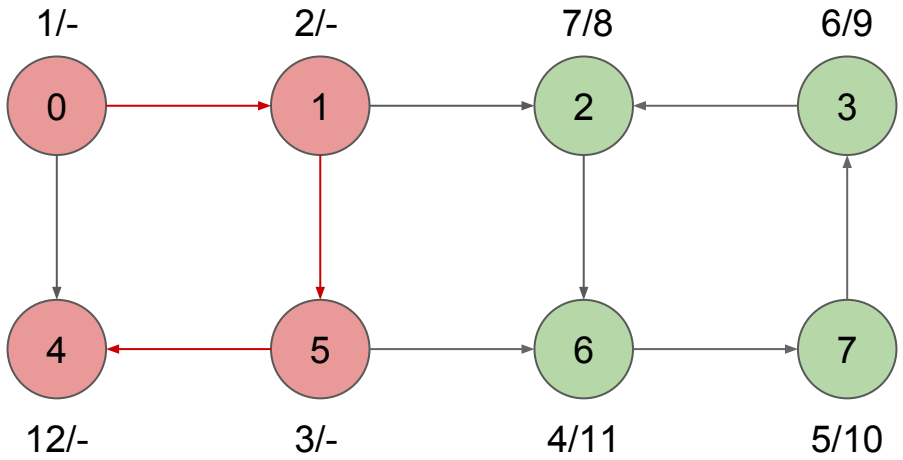
深度優先搜索 DFS

節點6連出去的邊都已經被走訪過了，所以節點 6走訪完了。



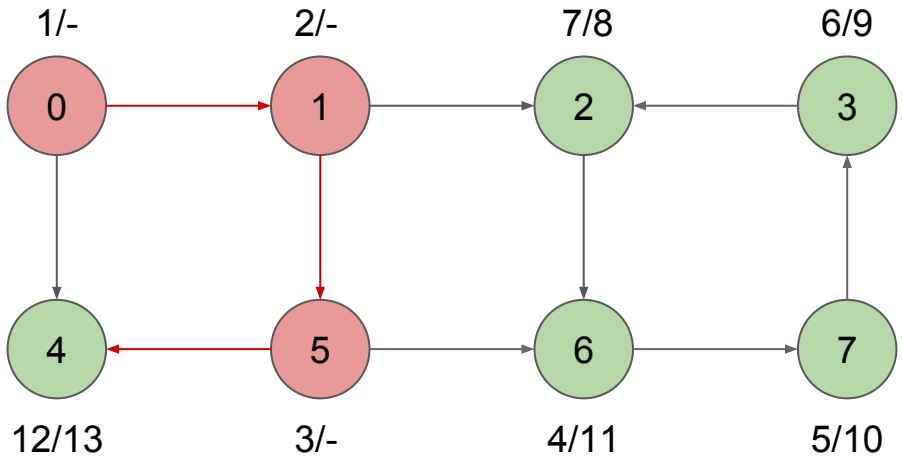
深度優先搜索 DFS

選出節點5連出去的邊中還未走訪的點 (節點4)



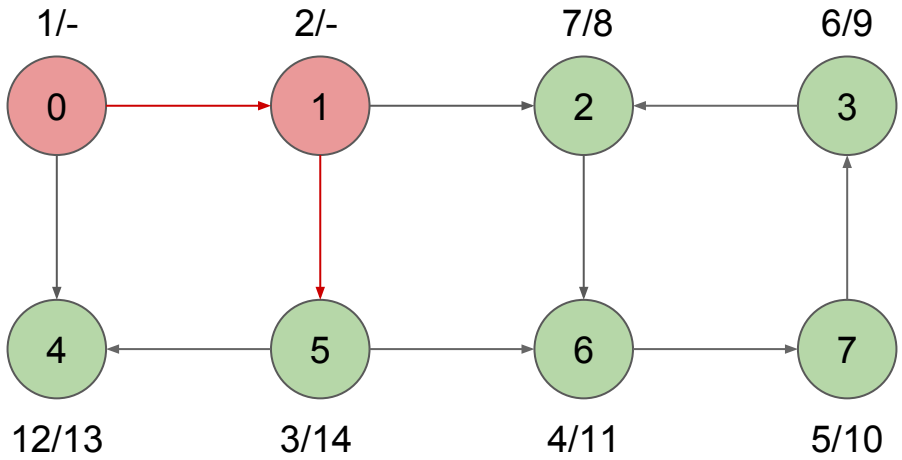
深度優先搜索 DFS

節點4連出去的邊都已經被走訪過了，所以節點 4走訪完了。



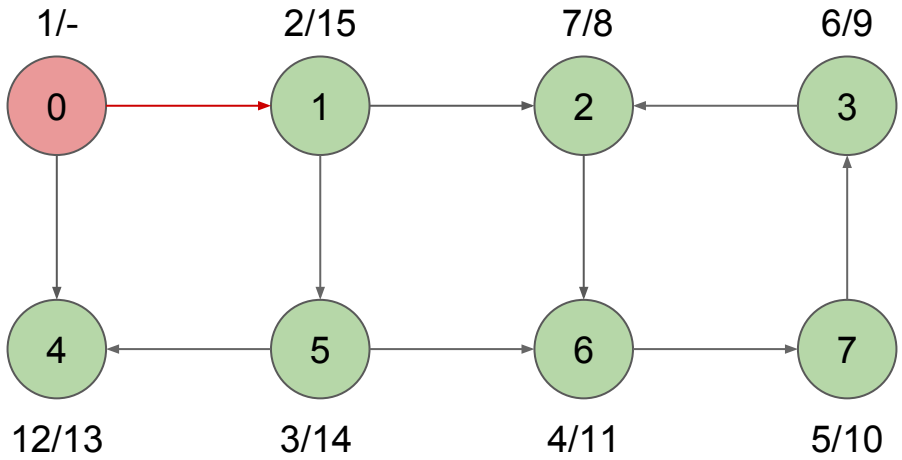
深度優先搜索 DFS

節點5連出去的邊都已經被走訪過了，所以節點 5走訪完了。



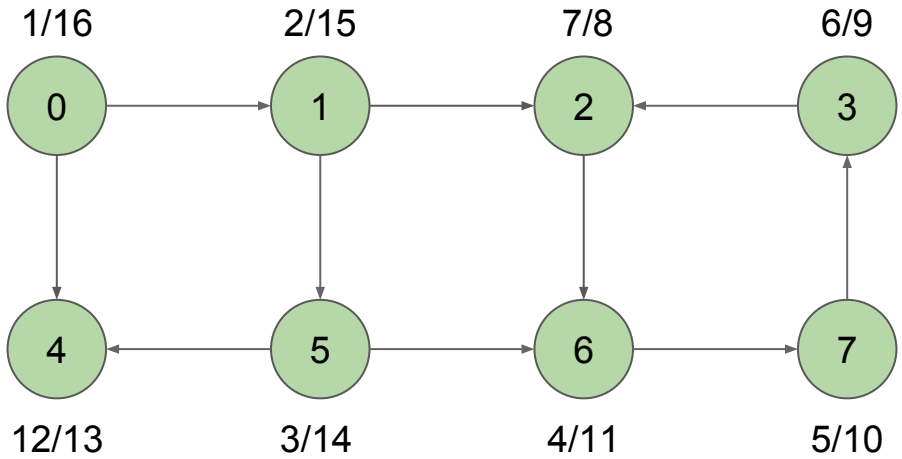
深度優先搜索 DFS

節點1連出去的邊都已經被走訪過了，所以節點 1走訪完了。



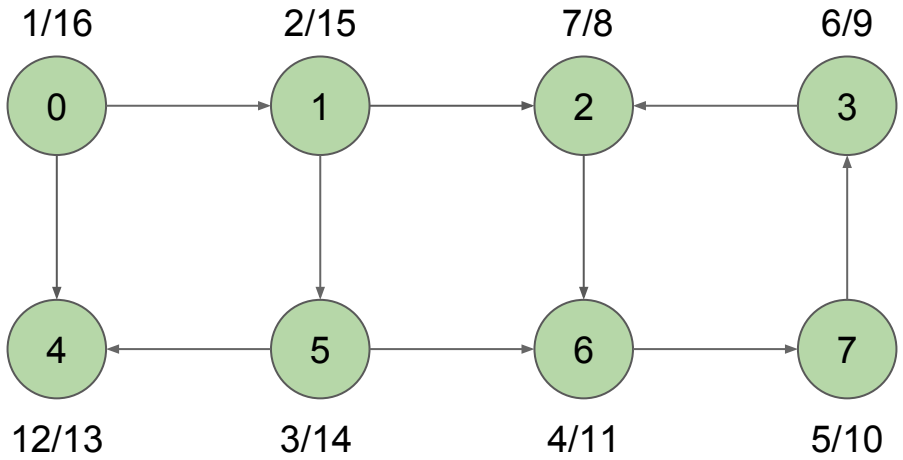
深度優先搜索 DFS

節點0連出去的邊都已經被走訪過了，所以節點 0走訪完了。



深度優先搜索 DFS

DFS結束，取得所有點的 (In stamp / Out stamp)



深度優先搜索 分析

- 如果使用鄰接串列
  - 每個邊跟點只會被掃過一次
  - 時間複雜度O(V+E)
- 如果使用鄰接矩陣
  - 每次拜訪每個點都要在花 O(V)的時間掃過跟每個點是否有邊
  - 時間複雜度O(V^2)

深度優先搜索 Code

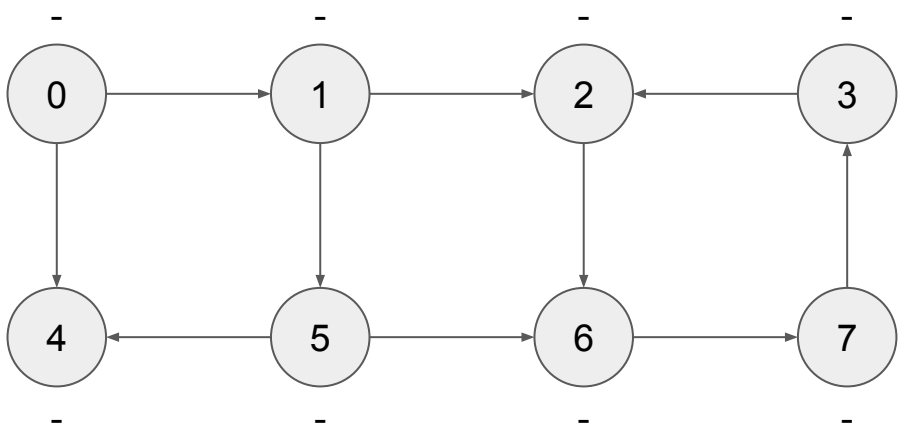
```cpp
const int MAXN = 1e3 + 5;
int inStamp[MAXN], outStamp[MAXN], stamp;
vector<int> Graph[MAXN];
void init() {
    memset(inStamp, 0, sizeof(inStamp));
    memset(outStamp, 0, sizeof(outStamp));
    stamp = 0;
}
void dfs(int u) {
    inStamp[u] = ++stamp;          // 拜訪前蓋inStamp戳章
    for (auto &v : Graph[u])       // 列舉所有邊
        if (inStamp[v] == 0)       // 還未拜訪過的點才dfs
            dfs(v);                // 拜訪下一個點
    outStamp[u] = ++stamp;         // 拜訪完蓋outStamp戳章
}
```

廣度優先搜索 BFS

- 又名 **B**readth **F**irst **S**earch
- 概念和淹水相同
  - 從原點擴散出去
- 重要的資訊
  - level值
  - 無權重圖中，為該點距離起點的最短路徑。
- 搭配Queue實作

廣度優先搜索 BFS

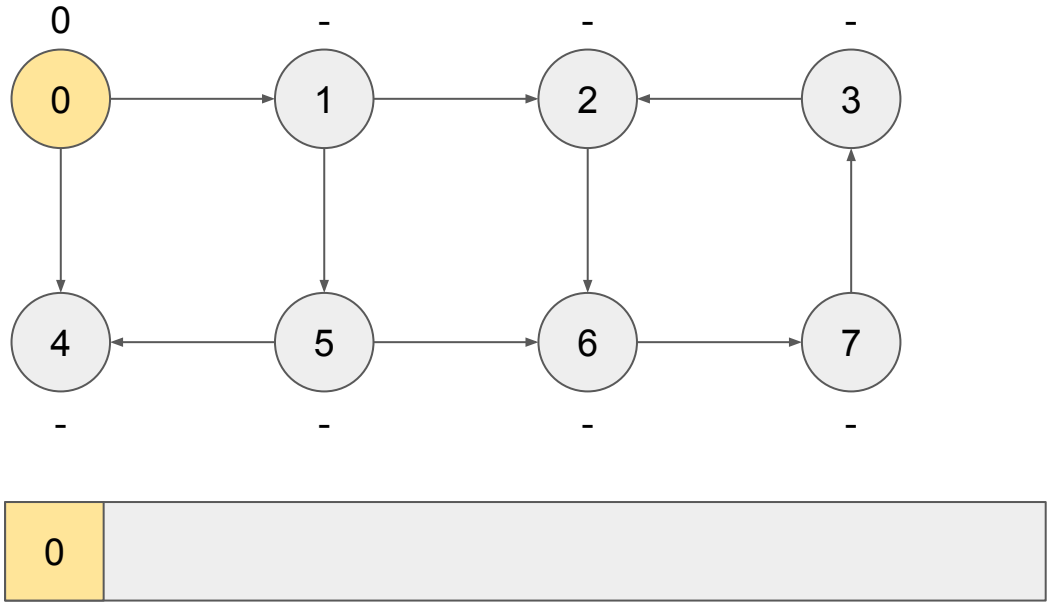
對這張圖做BFS



queue

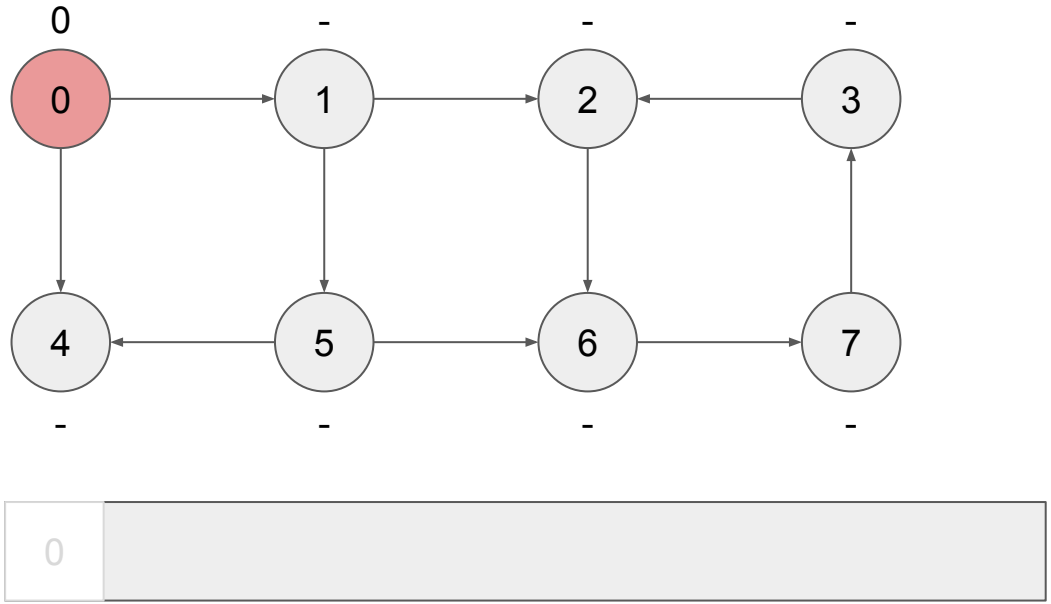
廣度優先搜索 BFS

節點0為起點，level值設為0，並丟入queue



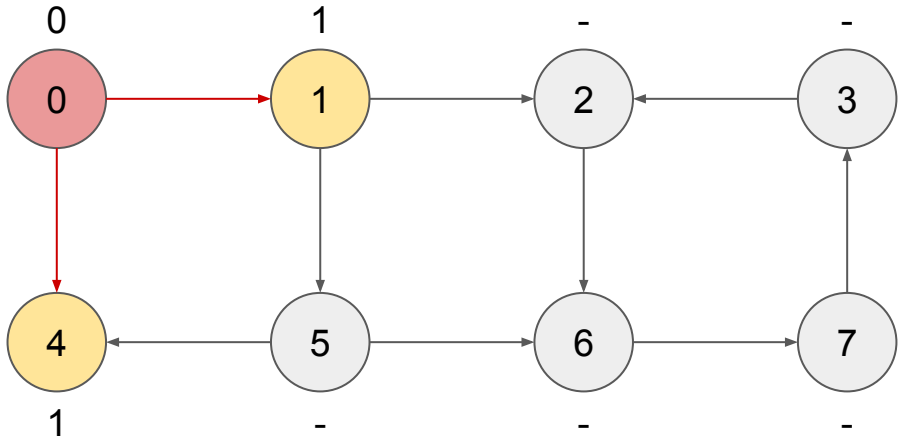
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



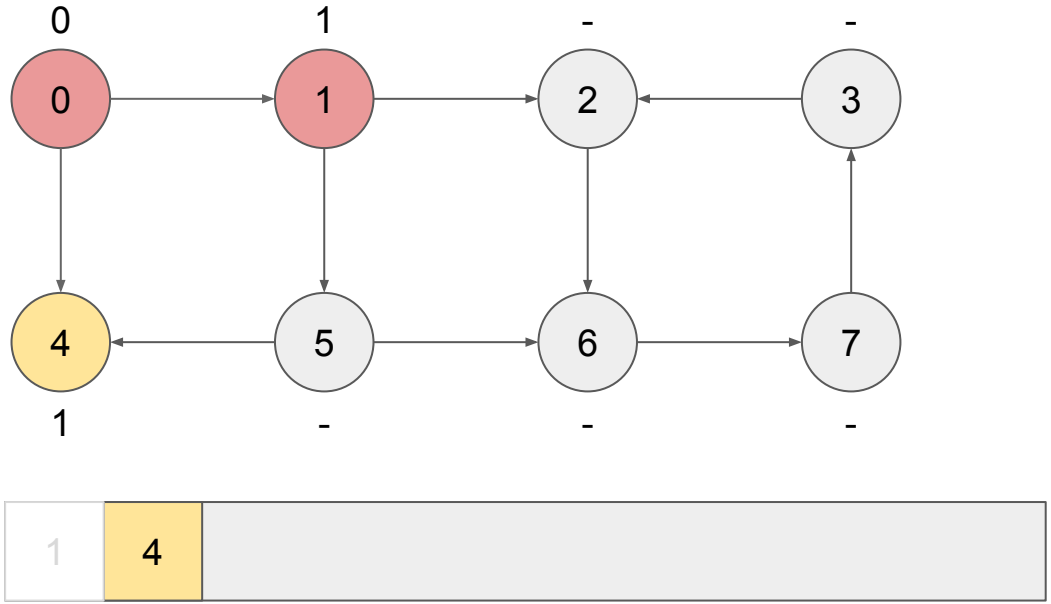
廣度優先搜索 BFS

將節點0相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



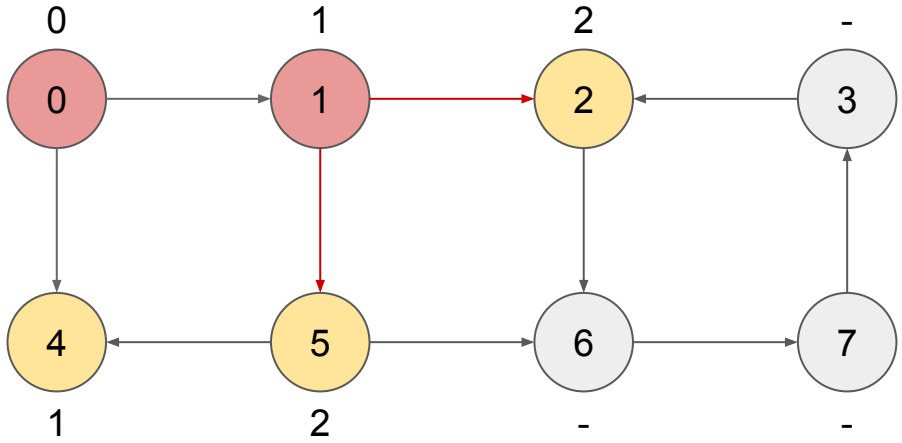
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



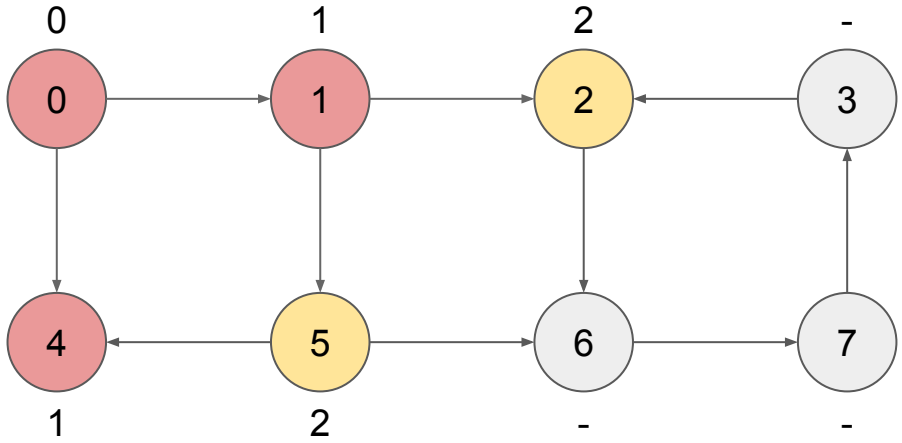
廣度優先搜索 BFS

將節點1相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



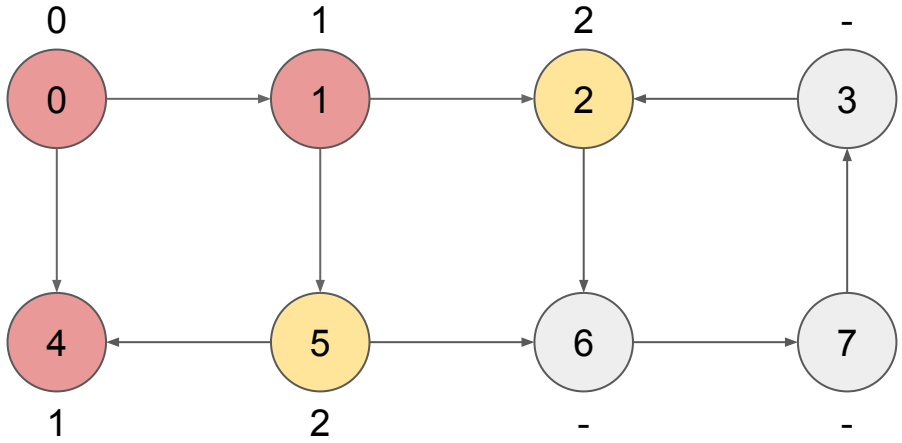
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



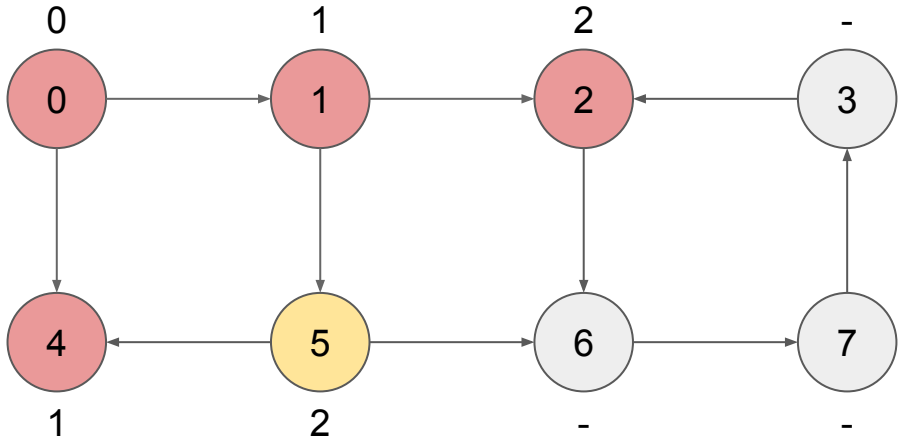
廣度優先搜索 BFS

節點4沒有相鄰且沒有被拜訪過的點, 不動作



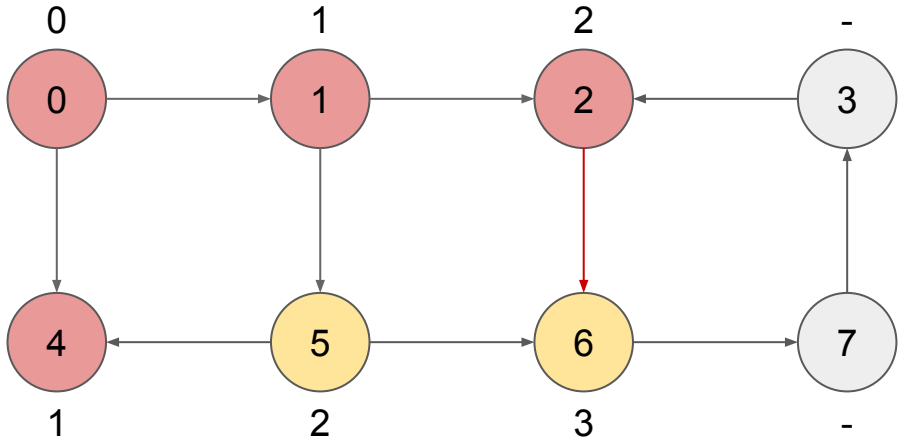
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



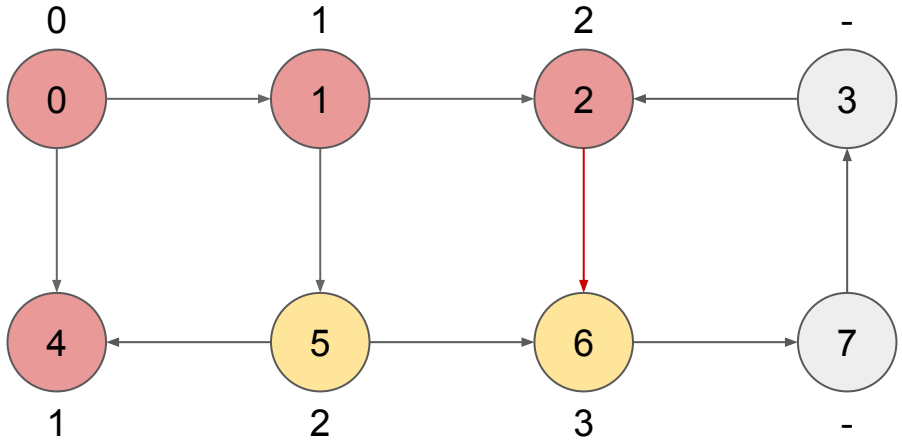
廣度優先搜索 BFS

將節點2相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



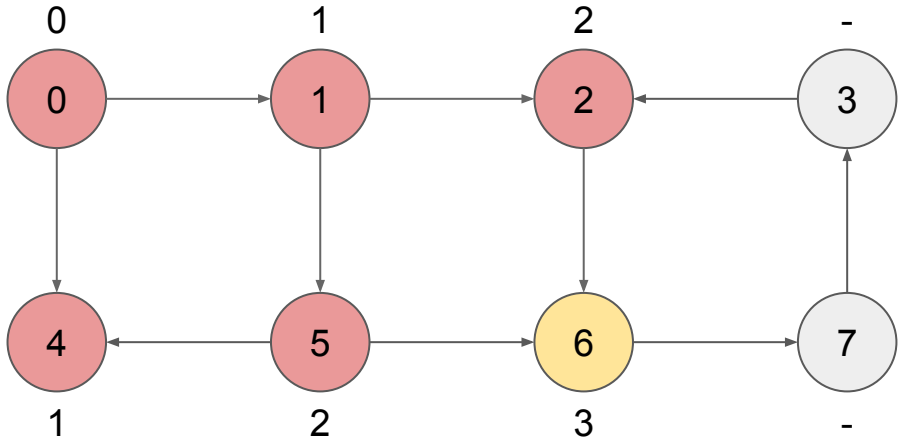
廣度優先搜索 BFS

將節點2相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



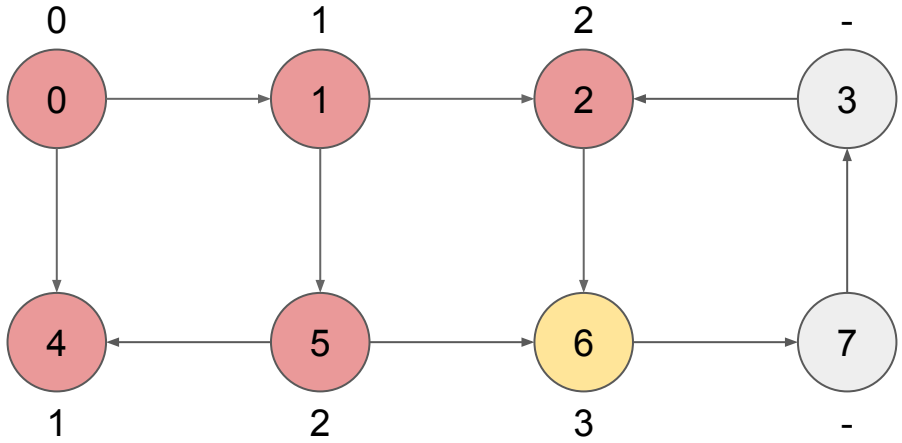
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



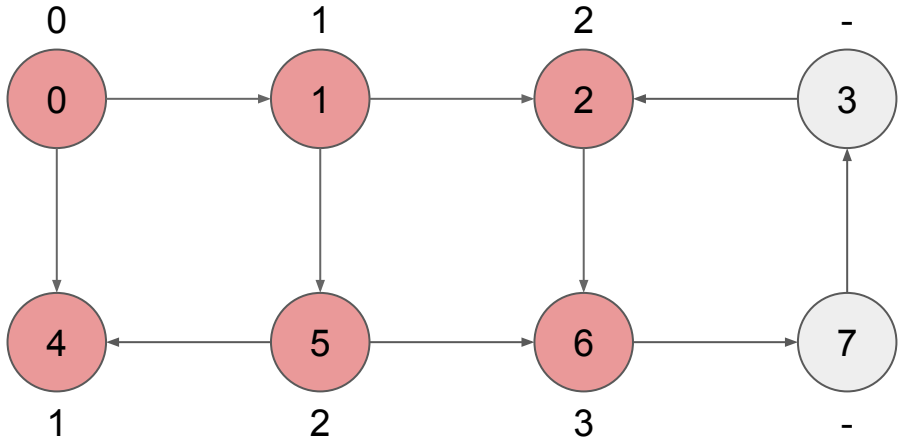
廣度優先搜索 BFS

節點5沒有相鄰且沒有被拜訪過的點, 不動作



廣度優先搜索 BFS

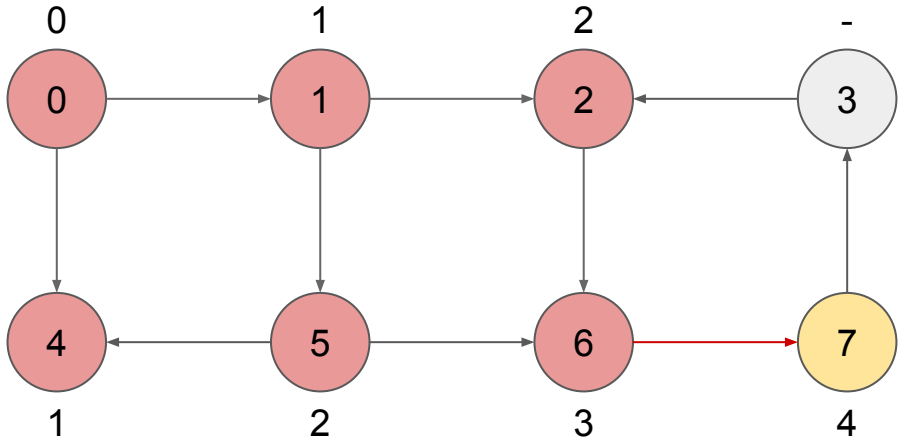
從queue中取最前面的節點，並 pop掉該節點



queue 6

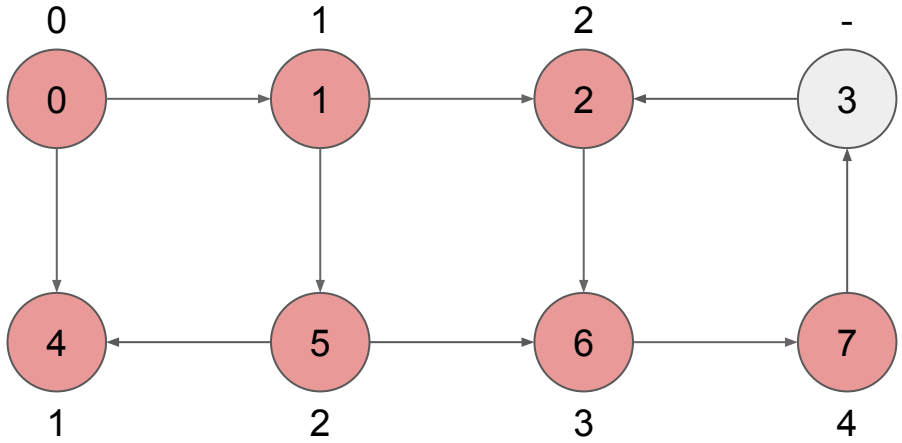
廣度優先搜索 BFS

將節點6相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



廣度優先搜索 BFS

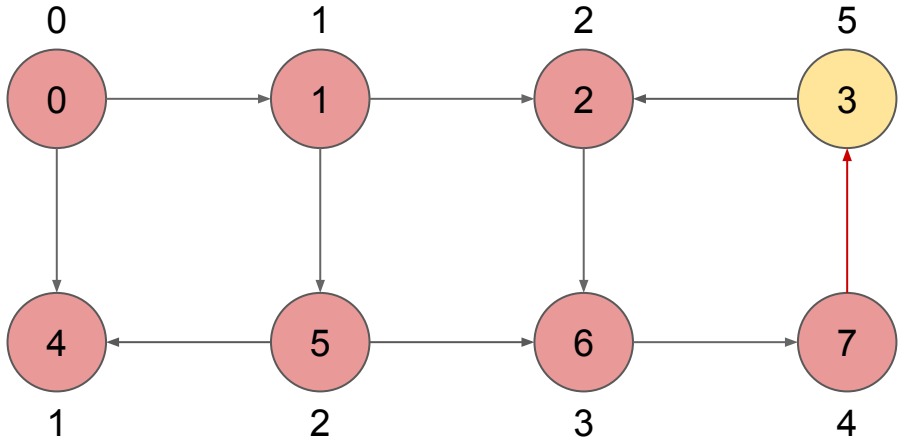
從queue中取最前面的節點，並 pop掉該節點



queue | 7 |

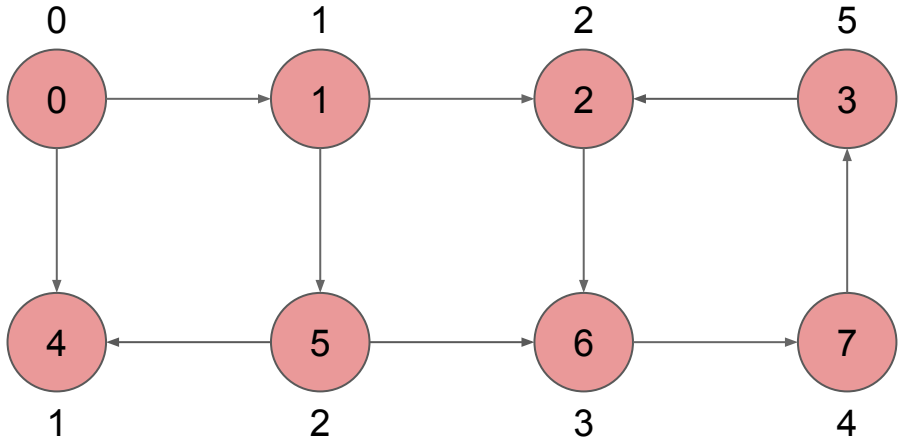
廣度優先搜索 BFS

將節點7相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



廣度優先搜索 BFS

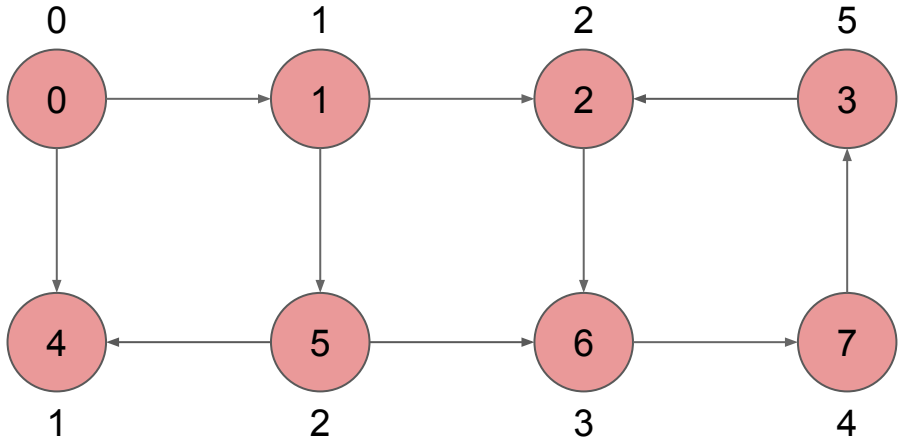
從queue中取最前面的節點，並 pop掉該節點



queue 3

廣度優先搜索 BFS

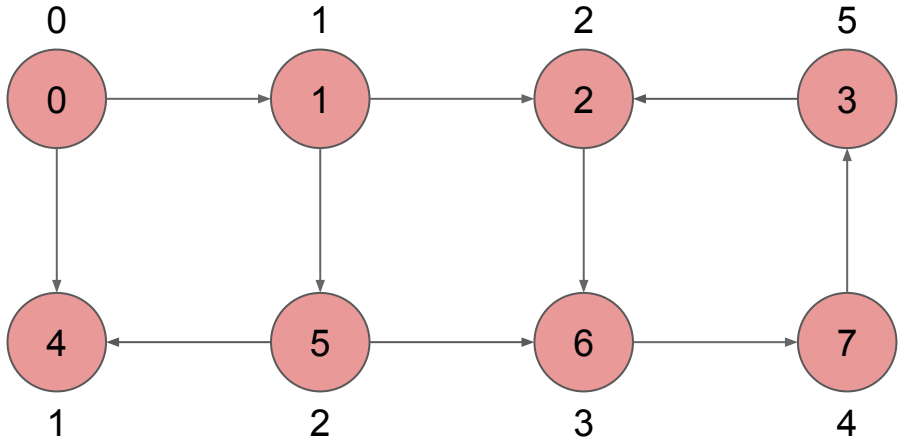
節點3沒有相鄰且沒有被拜訪過的點, 不動作



queue

廣度優先搜索 BFS

queue為空且每個點都走訪完畢, 完成 BFS



queue

廣度優先搜索 分析

- 如果使用鄰接串列
  - 每個邊跟點只會被掃過一次
  - 時間複雜度O(V+E)
- 如果使用鄰接矩陣
  - 每次拜訪每個點都要在花O(V)的時間掃過跟每個點是否有邊
  - 時間複雜度O(V^2)

廣度優先搜索 Code

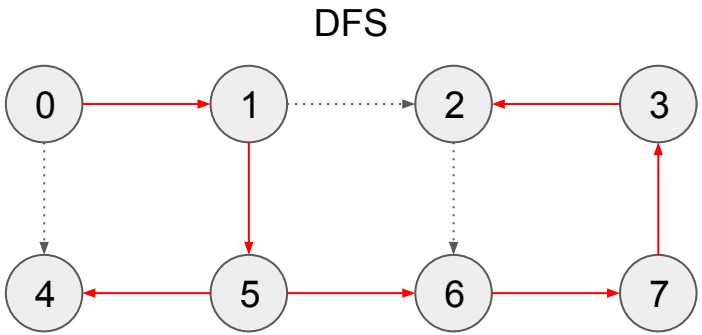
```cpp
const int MAXN = 1e3 + 5;
int level[MAXN];
vector<int> Graph[MAXN];
void init() {
    memset(level, -1, sizeof(level));
}
void bfs(int s) {
    queue<int> q;
    q.push(s); level[s] = 0;            // 起點丟進去queue裡面
    while (q.size()) {                  // 直到queue空為止才停
        int u = q.front(); q.pop();     // 拿出queue的最前端
        for (auto &v : Graph[u]) {      // 列舉該點的所有邊
            if (level[v] != -1)         // 遇到已經被拜訪的點
                continue;               // continue不動作
            level[v] = level[u] + 1;    // 否則，更新level[v]
            q.push(v);                  // 並丟入queue裡面
        }
    }
}
```

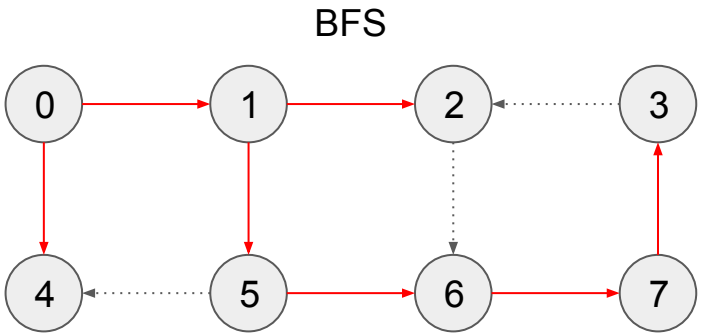
DFS & BFS

- 對圖最基礎的兩個操作
  - 大部分的演算法都為這兩種操作的組合
- 代價便宜
  - 如果用Adjaceny list, 代價均為線性時間

DFS & BFS 樹

DFS和BFS做完之後，會把一張圖變成一棵樹





樹

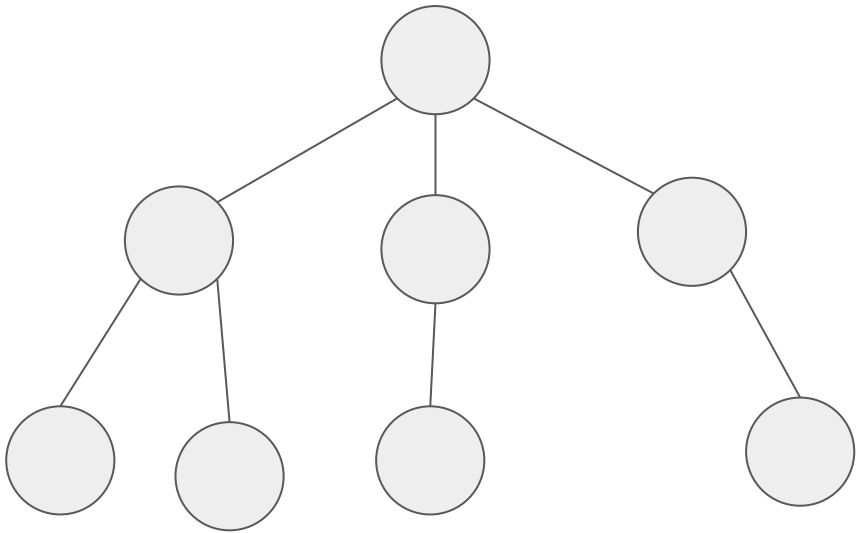
- 這是一棵仿真實世界的樹

樹

- 這是一棵仿真實世界的樹
- 但資工的樹通常是反過來的

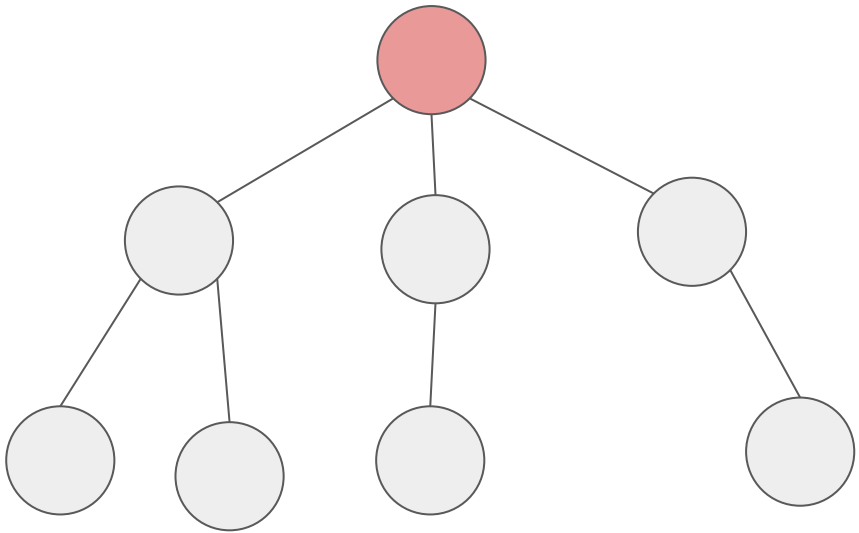
樹

- 樹的一些重要名詞



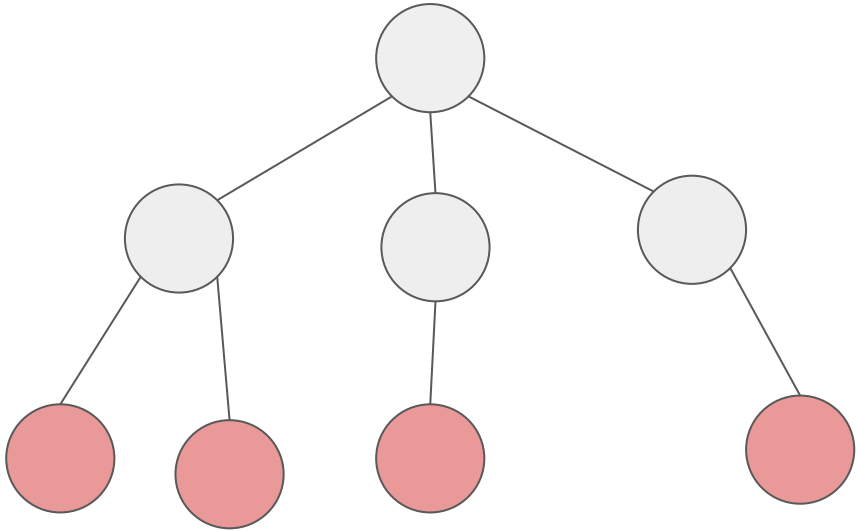
樹

- 根
  - 位於樹的最頂端



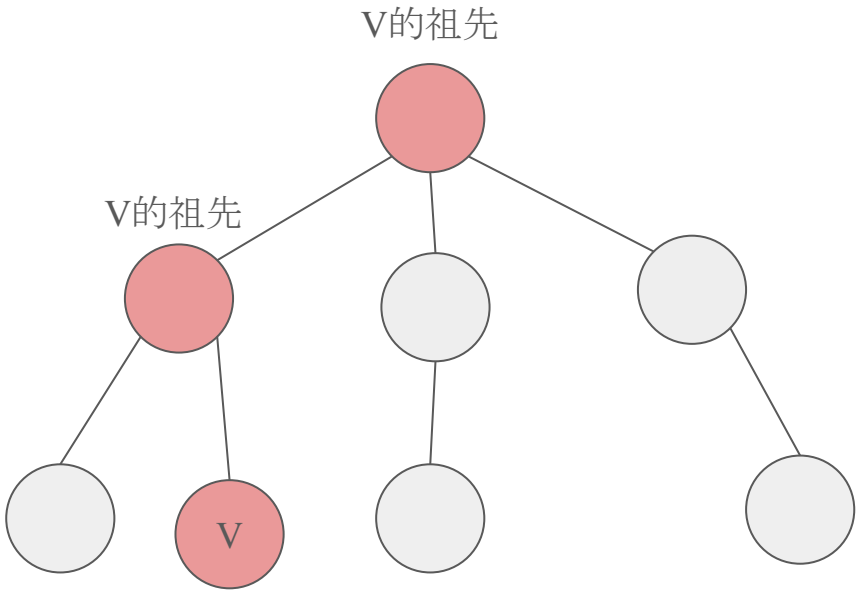
樹

- 葉子
  - 位於樹的最底端



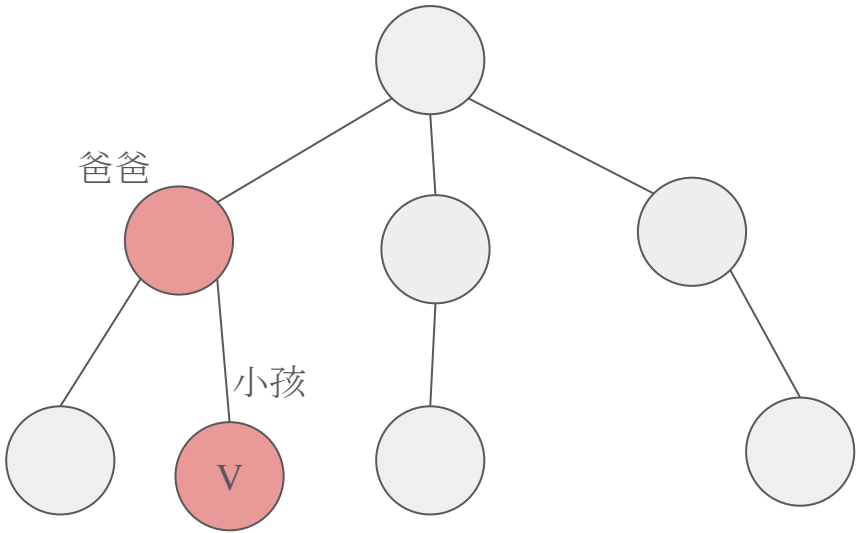
樹

- 點和點關係
  - 祖先
    - 該點到根的路徑經過的點



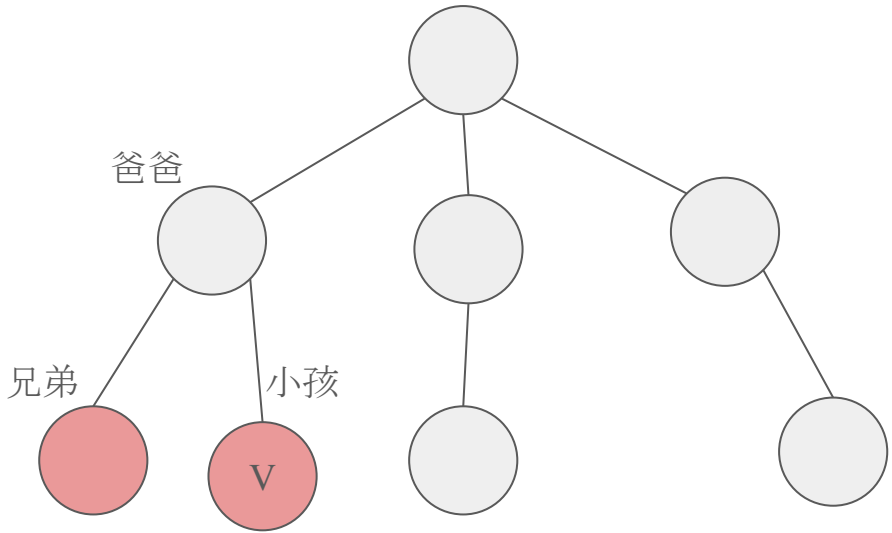
樹

- 點和點關係
  - 小孩和爸爸
    - 該點的上一層



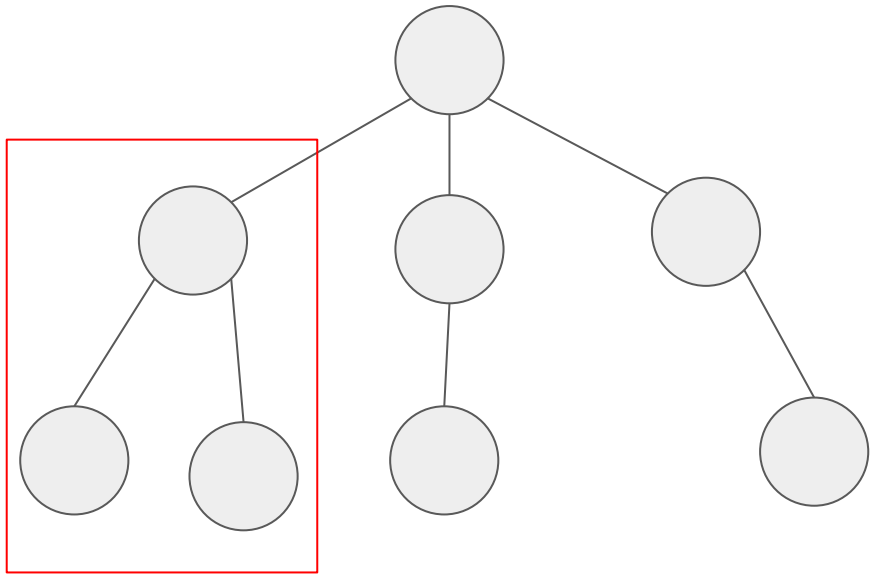
樹

- 點和點關係
  - 兄弟姊妹
    - 同一個爸爸



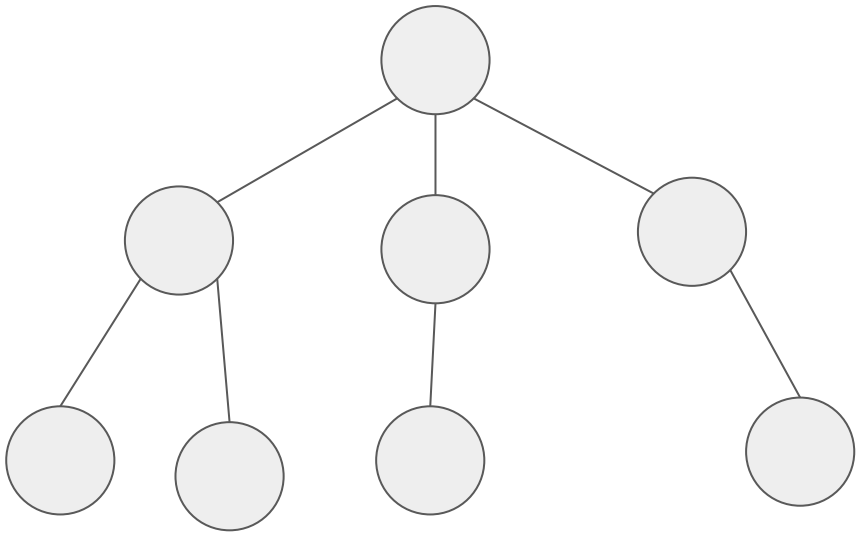
樹

- 子樹
  - 樹的某一部份切下來也會是一棵樹



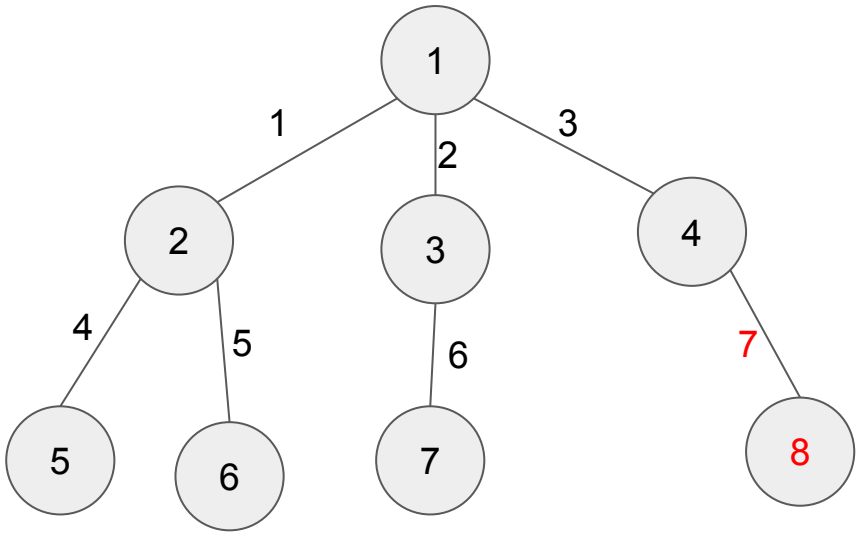
樹

- 樹的一些重要性質



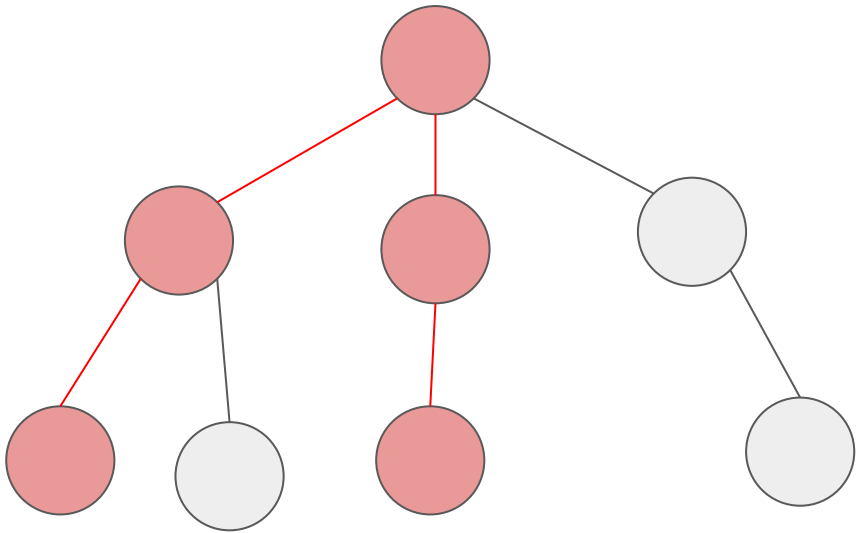
樹

- 樹的一些重要性質
  - n個點的樹僅有n-1條邊



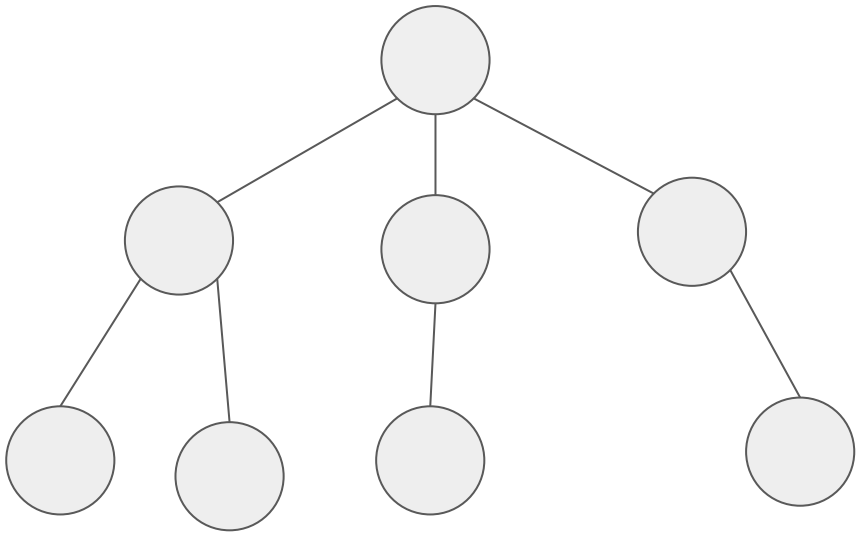
樹

- 樹的一些重要性質
  - n個點的樹僅有n-1條邊
  - 任兩點僅有唯一一條路徑



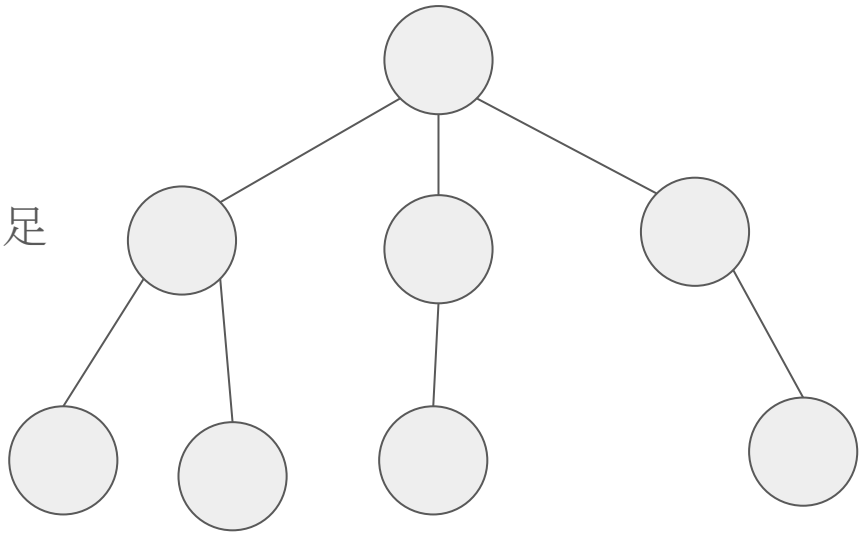
樹

- 樹的一些重要性質
  - n個點的樹僅有n-1條邊
  - 任兩點僅有唯一一條路徑
  - 找不到環



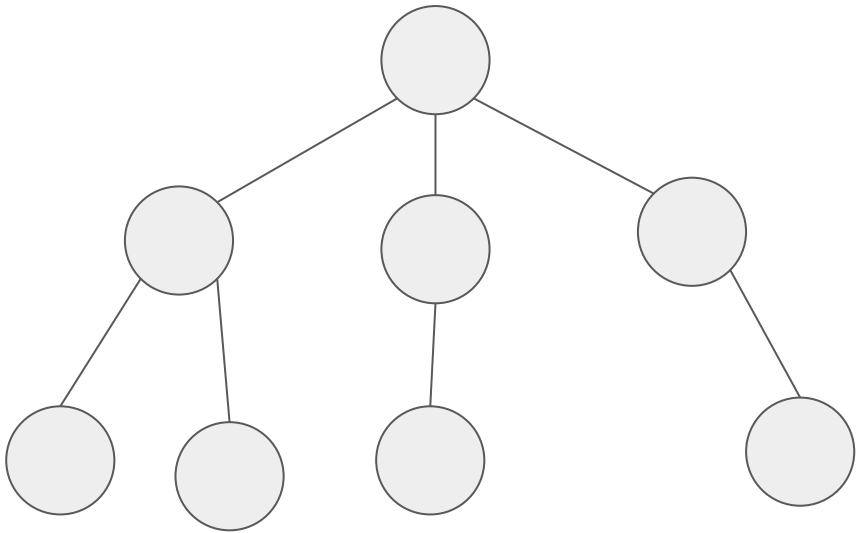
樹

- 樹的一些重要性質
  - n個點的樹僅有n-1條邊
  - 任兩點僅有唯一一條路徑
  - 找不到環
- 滿足任兩個性質，第三個就會自動滿足



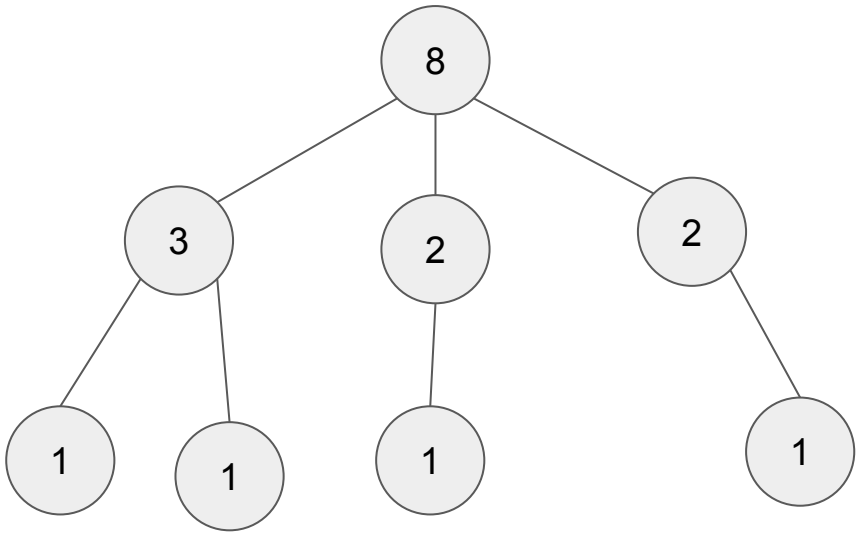
樹的問題 (子樹的size)

- 建立出每一個子樹他的size有多大



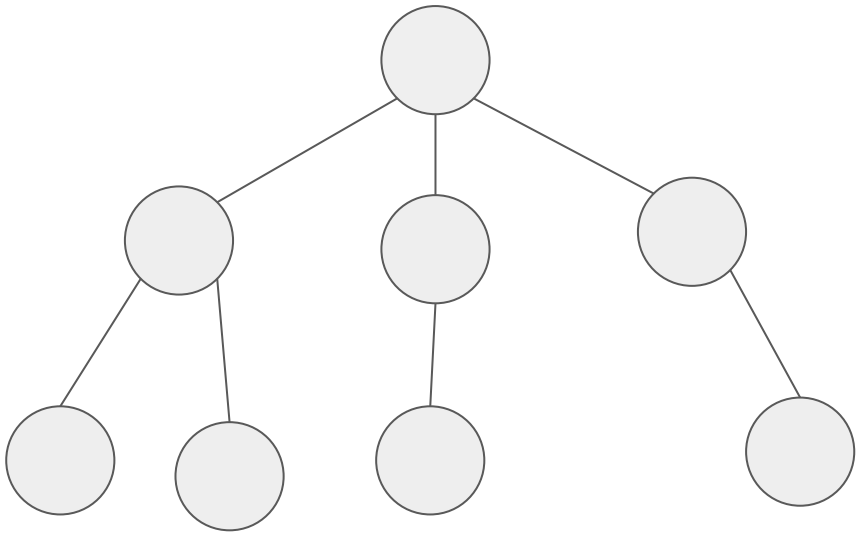
樹的問題 (子樹的size)

- 建立出每一個子樹他的size有多大
- 以右圖來說是這樣



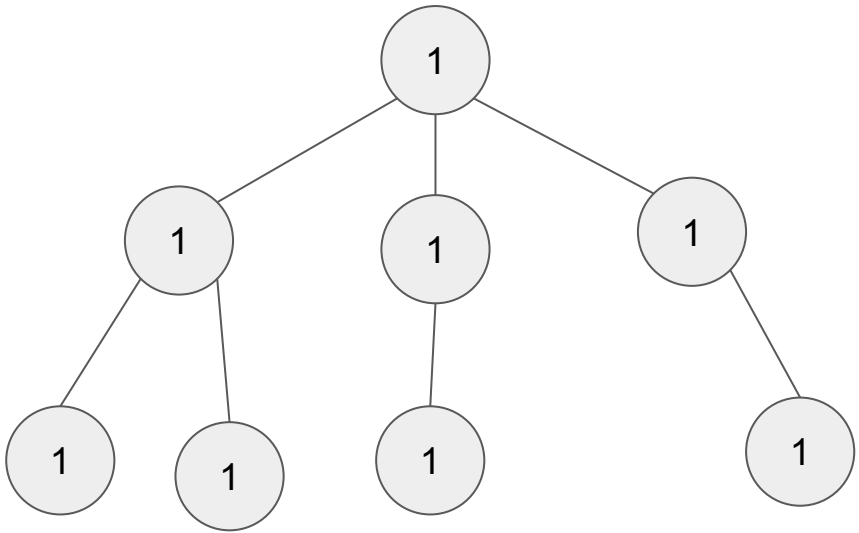
樹的問題 (子樹的size)

- 方法:一次DFS



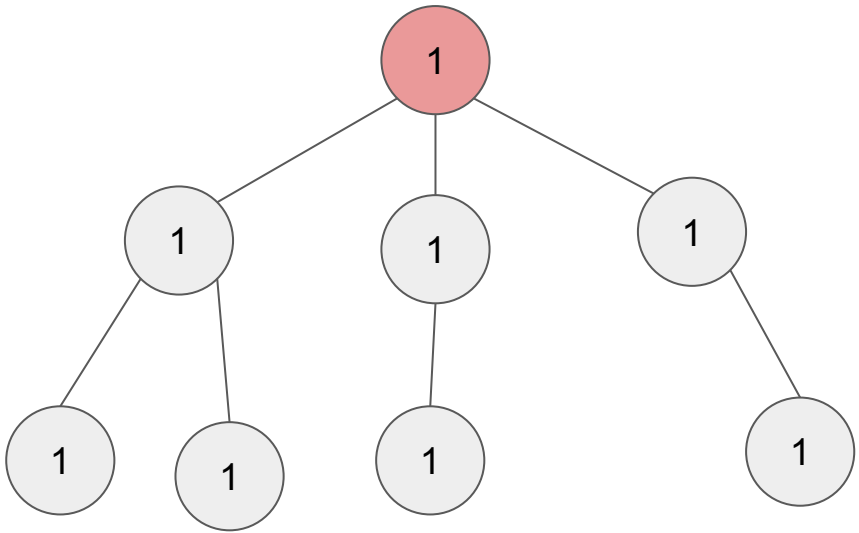
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1



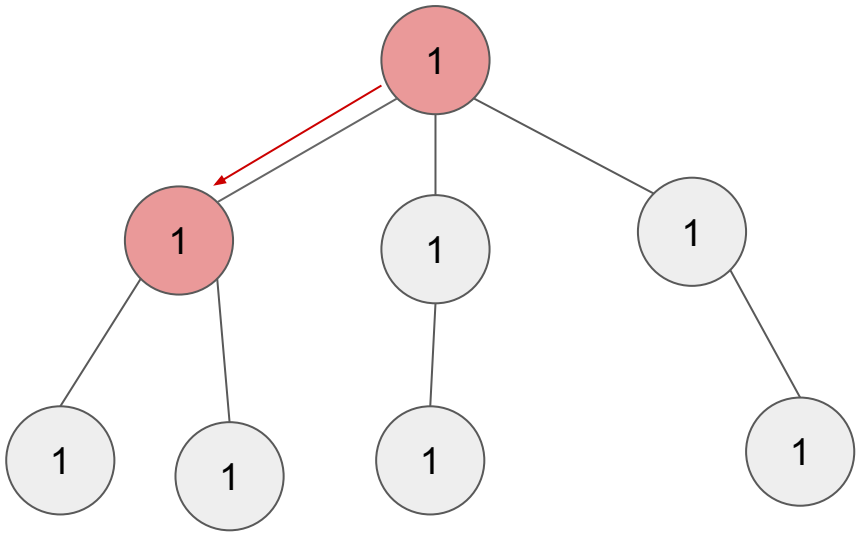
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS



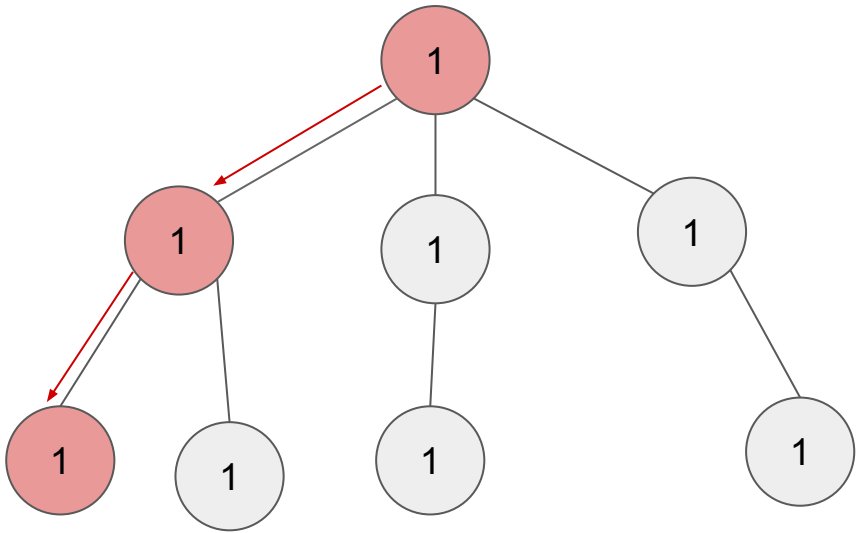
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們



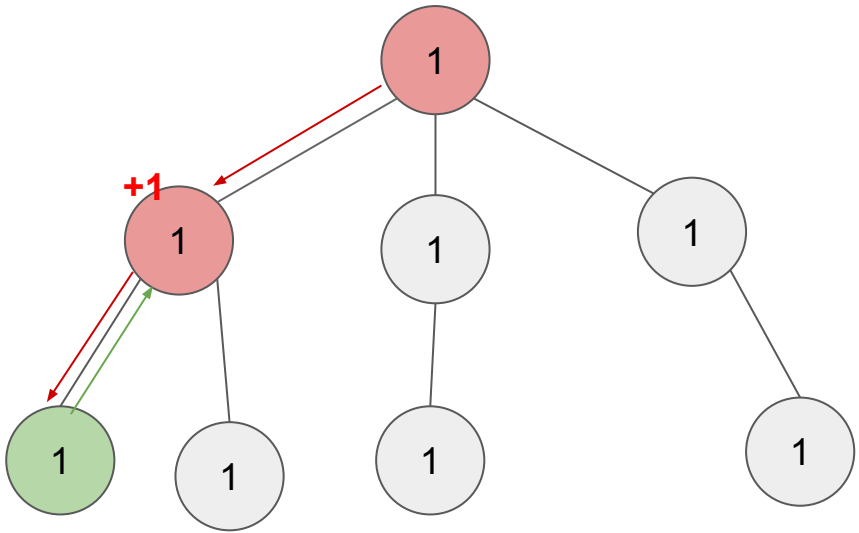
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們



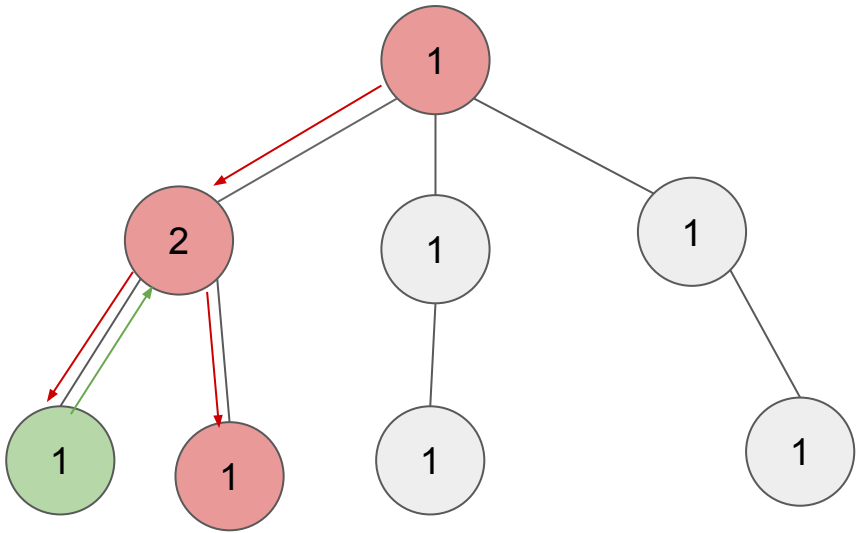
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸



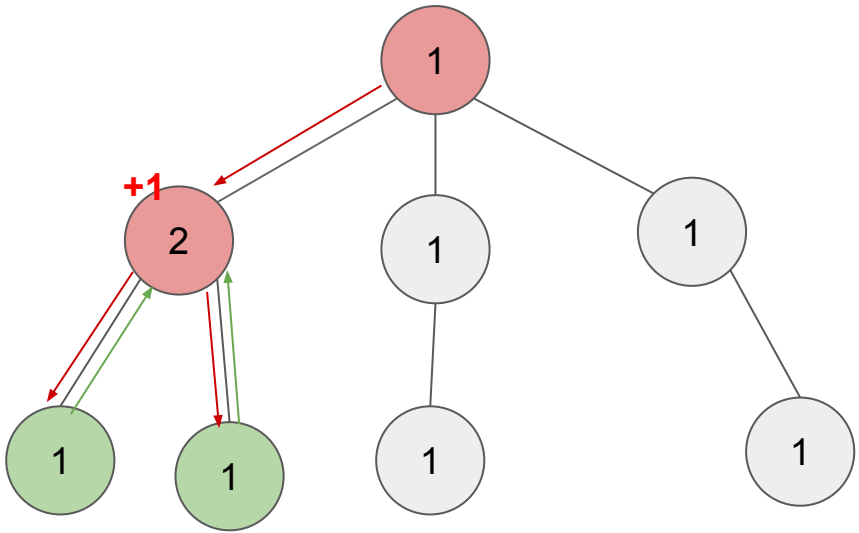
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸



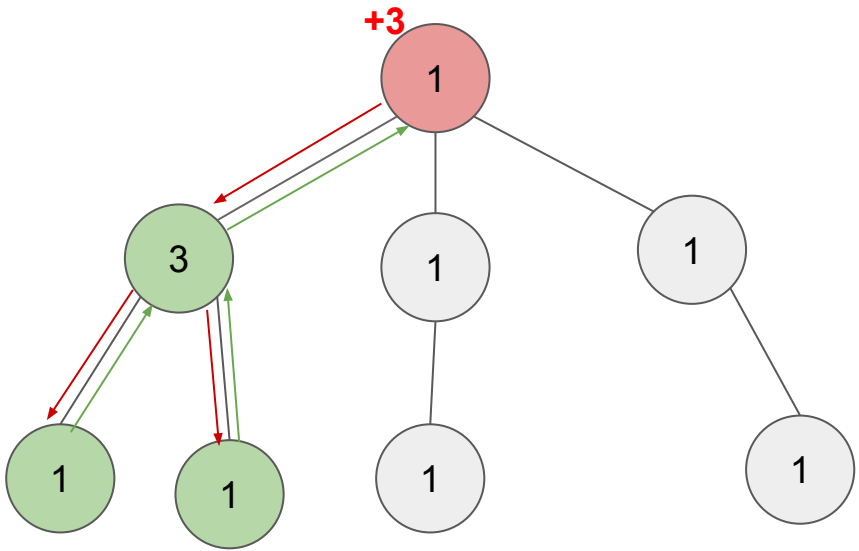
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸



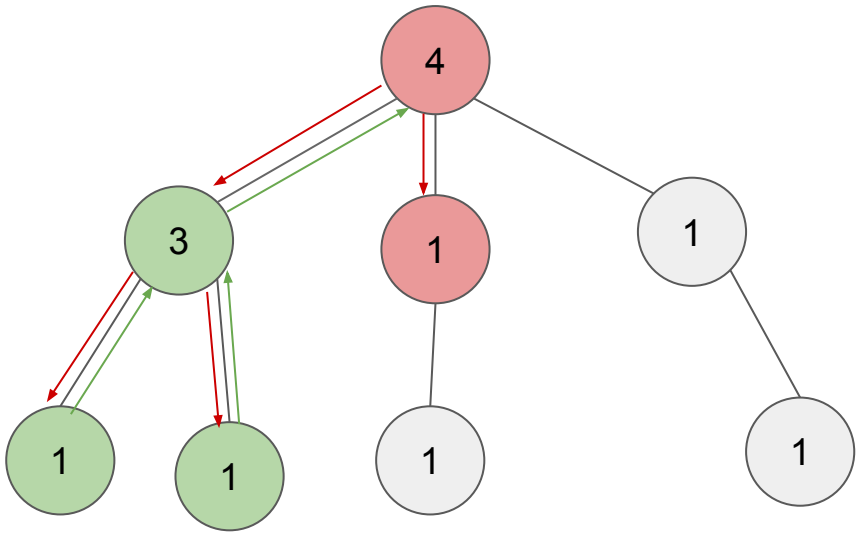
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸



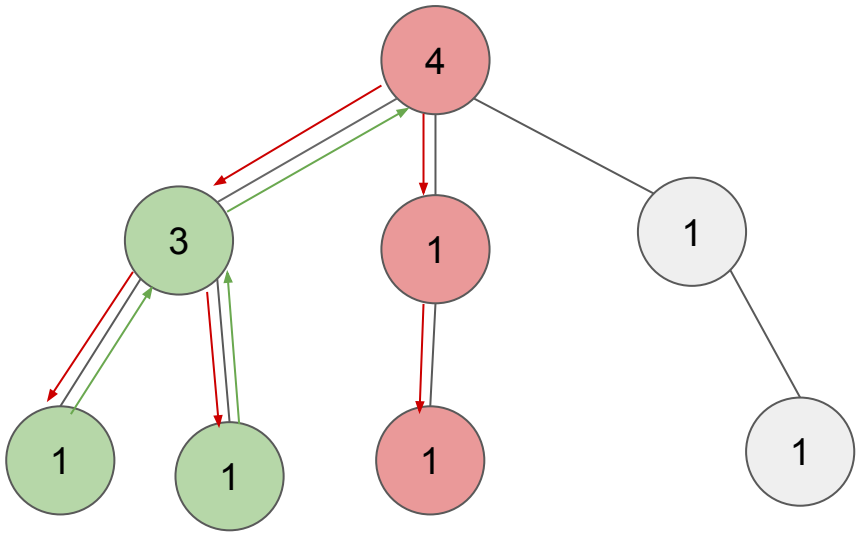
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸
  - 做到DFS結束就會是答案了



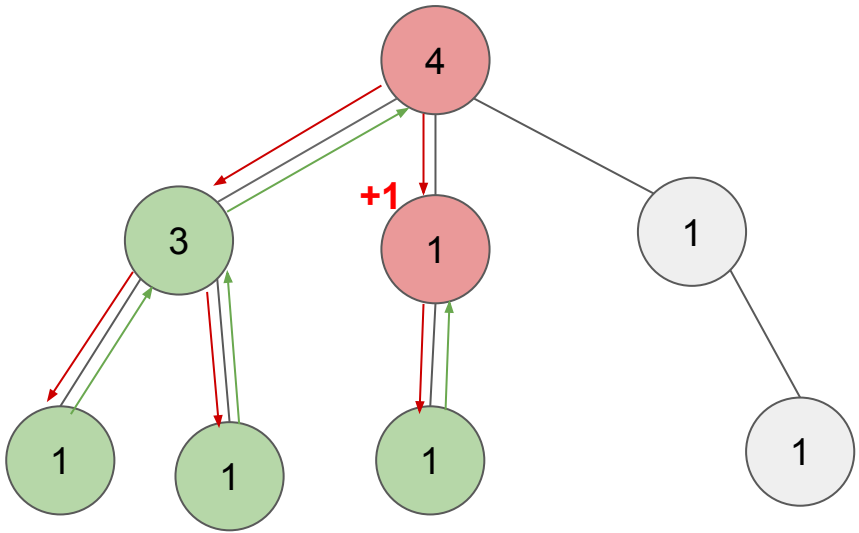
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸
  - 做到DFS結束就會是答案了



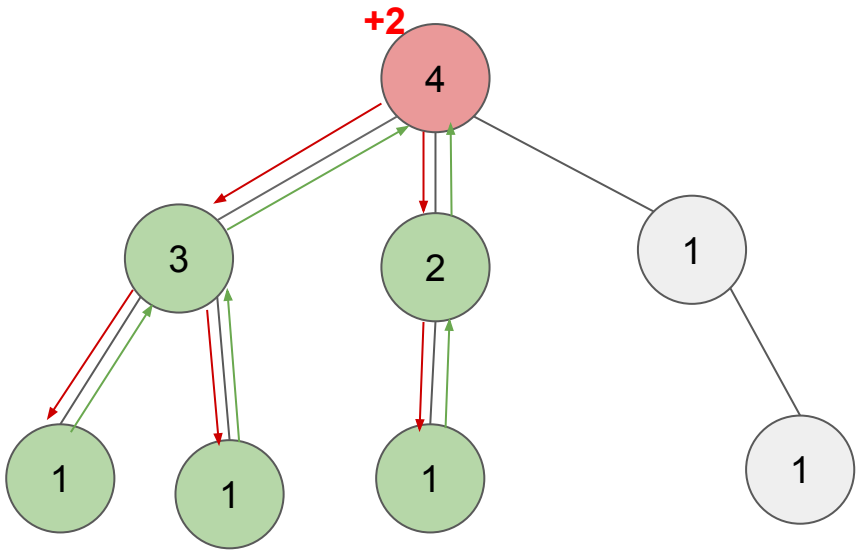
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸
  - 做到DFS結束就會是答案了



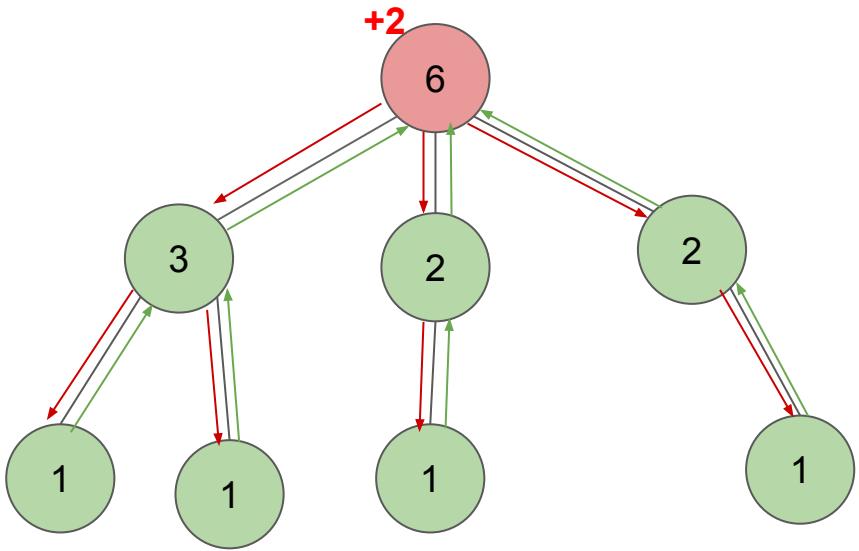
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸
  - 做到DFS結束就會是答案了



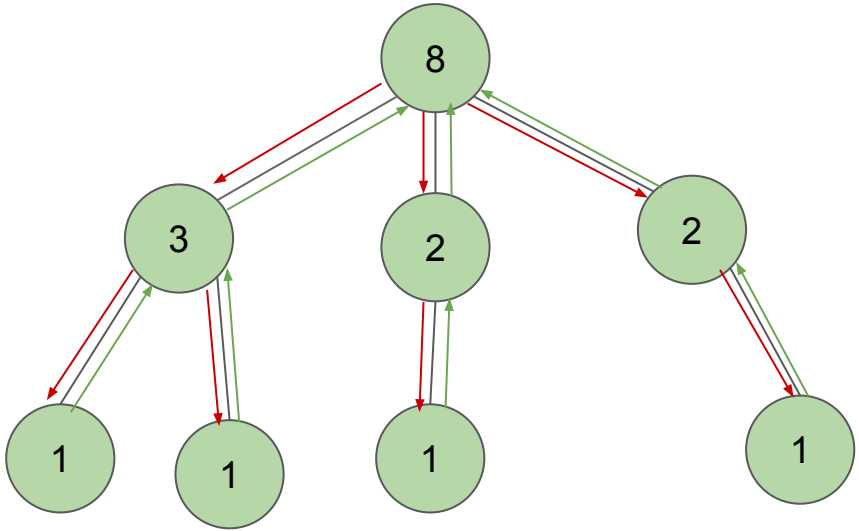
樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸
  - 做到DFS結束就會是答案了



樹的問題 (子樹的size)

- 方法:一次DFS
- 先將每一個節點size值設成1
- 從根開始DFS
  - 遍歷他的小孩們
  - 往上的時候把小孩的 值加給爸爸
  - 做到DFS結束就會是答案了
- 過程只有一次DFS
  - 時間複雜度O(V+E)

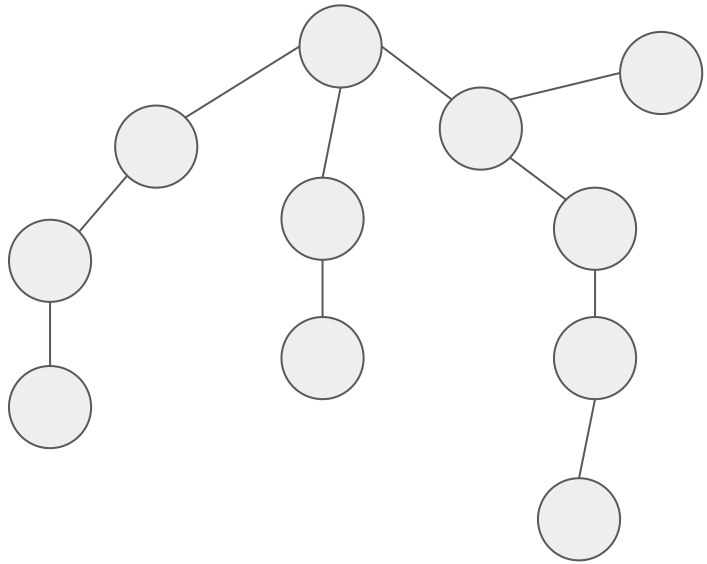


樹的問題 (子樹的size) Code

```
 5 const int MAXN = 1e3 + 5;
 6 vector<int> Graph[MAXN];
 7 int sz[MAXN];
 8 int dfs(int u, int p) {
 9     sz[u] = 1;                          // 初始化每一個節點sz為1
10     for (auto &v : Graph[u])            // 列舉每一條邊
11         if (v != p)                     // v不為爸爸才列舉
12             sz[u] += dfs(v, u);         // 把小孩的sz加來爸爸
13     return sz[u];                       // 回傳給u的爸爸
14 }
```

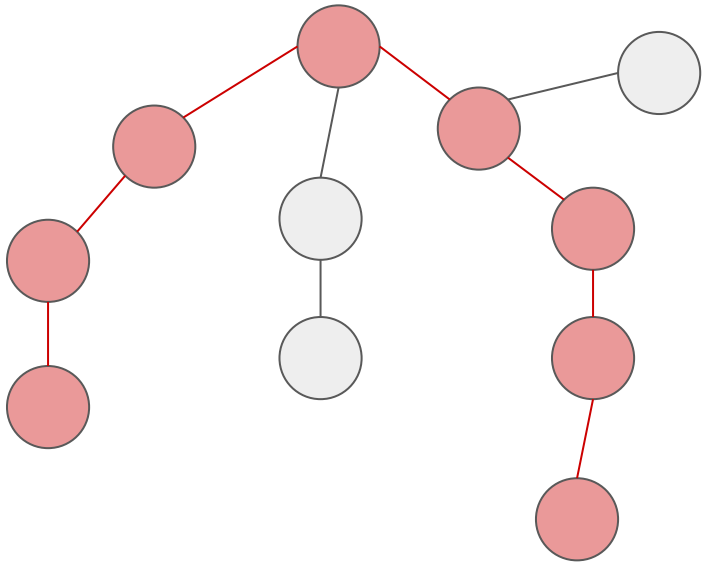
樹的問題 (樹的直徑)

- 找出這棵樹中最長的那條路徑



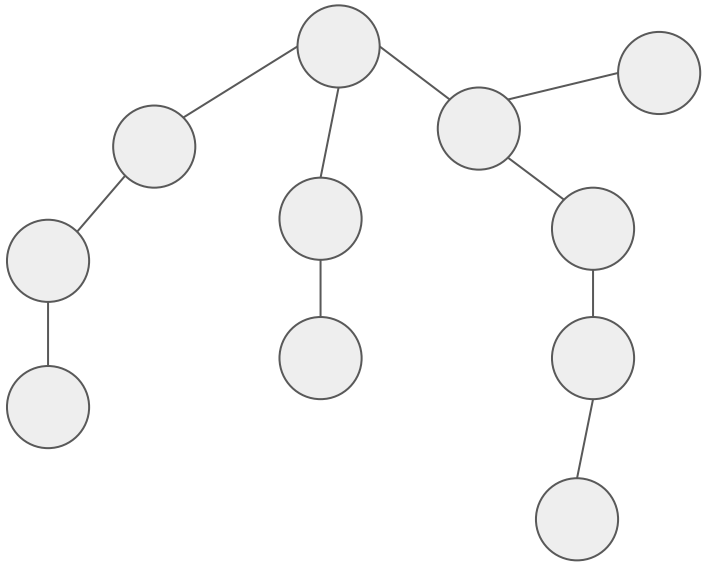
樹的問題 (樹的直徑)

- 找出這棵樹中最長的那條路徑
- 以右圖來說是這樣



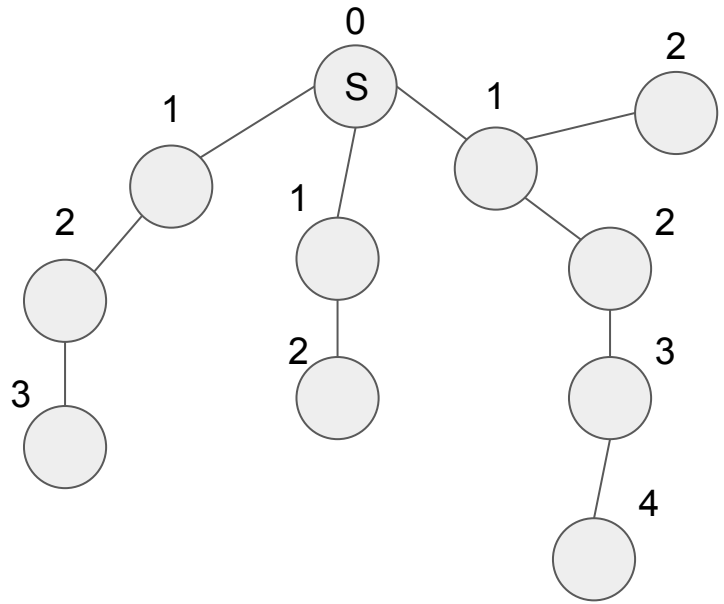
樹的問題 (樹的直徑)

- 方法:兩次BFS



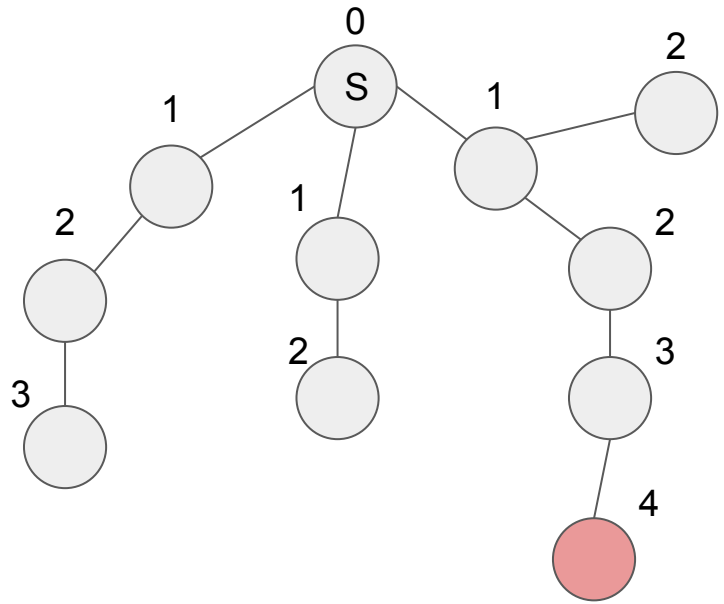
樹的問題 (樹的直徑)

- 方法:兩次BFS
- 隨便對一個點為起點做BFS



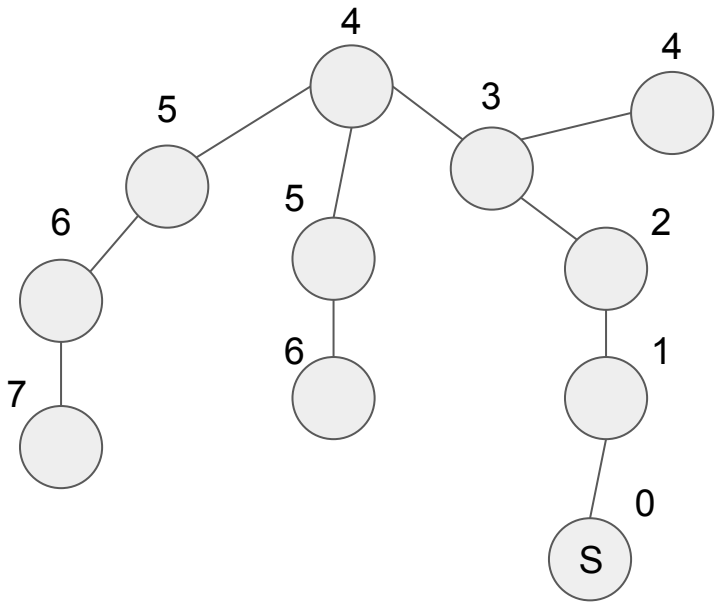
樹的問題 (樹的直徑)

- 方法:兩次BFS
- 隨便對一個點為起點做BFS
- 找出離S最遠的點



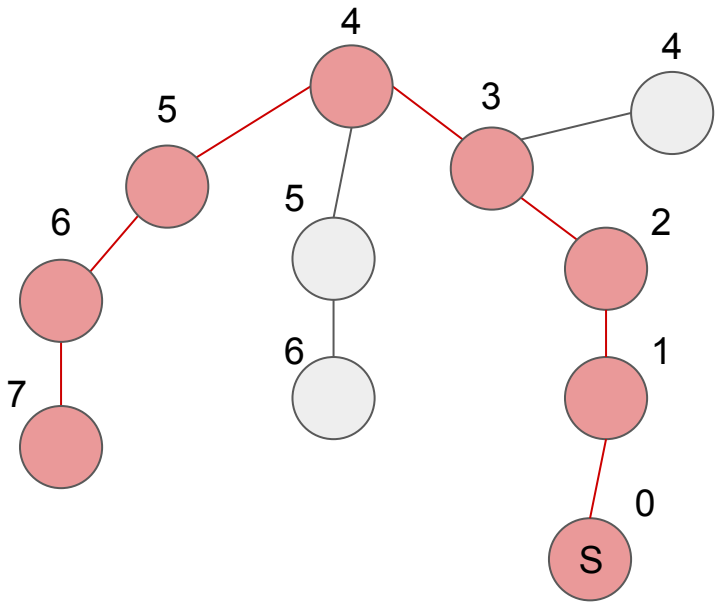
樹的問題 (樹的直徑)

- 方法:兩次BFS
- 隨便對一個點為起點做BFS
- 找出離S最遠的點
- 再以該點做一次BFS



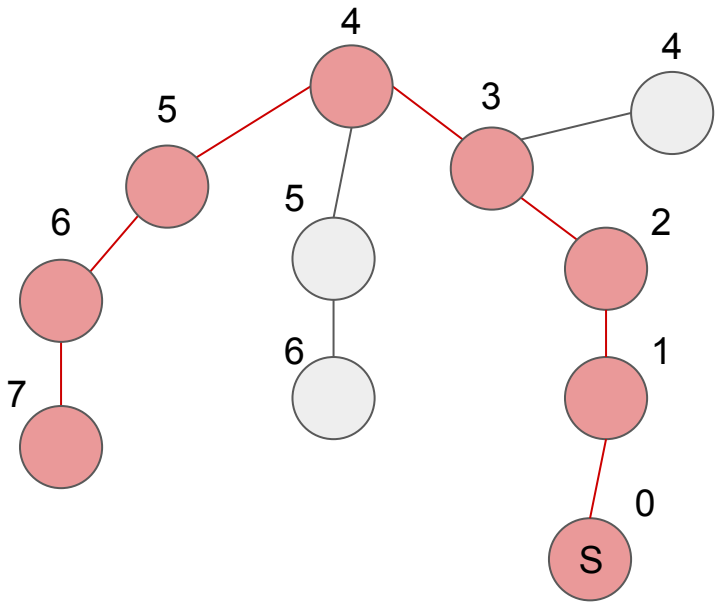
樹的問題 (樹的直徑)

- 方法:兩次BFS
- 隨便對一個點為起點做BFS
- 找出離S最遠的點
- 再以該點做一次BFS
- S和最遠的點的路徑就是樹直徑



樹的問題 (樹的直徑)

- 方法:兩次BFS
- 隨便對一個點為起點做BFS
- 找出離S最遠的點
- 再以該點做一次BFS
- S和最遠的點的路徑就是樹直徑
- 做兩次BFS, 時間複雜度O(V+E)

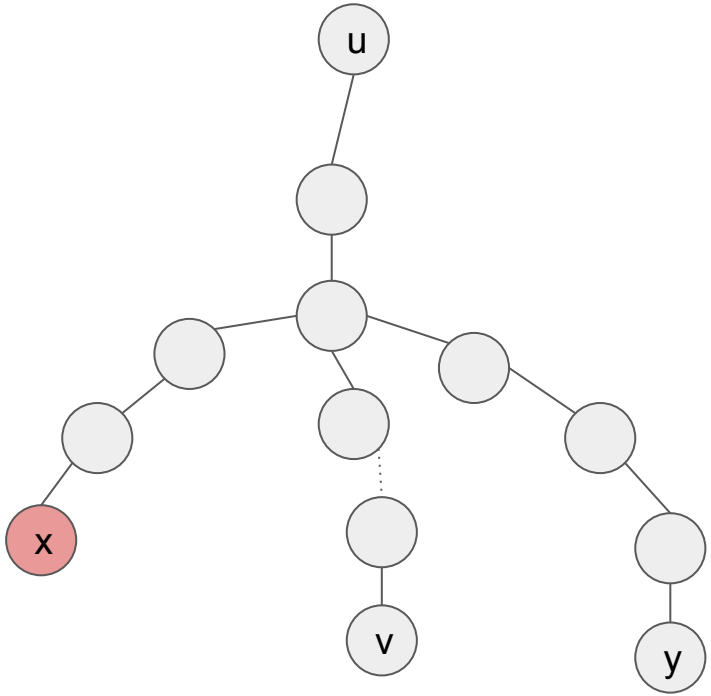


樹的問題 (樹的直徑) Code

```
 5 const int MAXN = 1e3 + 5;
 6 vector<int> Graph[MAXN];
 7 int level[MAXN];
 8 void init();                              // 銜接bfs的code
 9 void bfs(int s);                          // 銜接bfs的code
10 int getDiameter() {
11     init(); bfs(0);                       // 初始化並bfs隨便一個點
12     int maxLevel = -1, maxIndex = -1;     // 找到最大的level值和他的index
13     for (int i = 0 ; i < MAXN ; i++) {
14         if (maxLevel < level[i]) {
15             maxLevel = level[i];
16             maxIndex = i;
17         }
18     }
19     init(); bfs(maxIndex);                // 初始化並從maxIndex開始bfs
20     maxLevel = -1, maxIndex = -1;         // 找到最大的level值 即為直徑
21     for (int i = 0 ; i < MAXN ; i++) {
22         if (maxLevel < level[i]) {
23             maxLevel = level[i];
24             maxIndex = i;
25         }
26     }
27     return maxLevel;
28 }
```

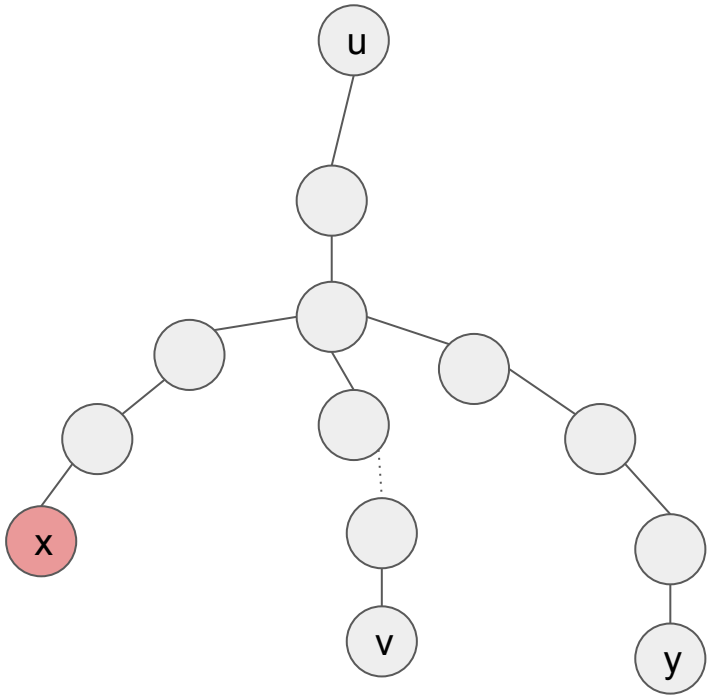
樹的問題 (樹的直徑) 正確性

- 假設樹的直徑為 x - y, 且BFS的起點為x



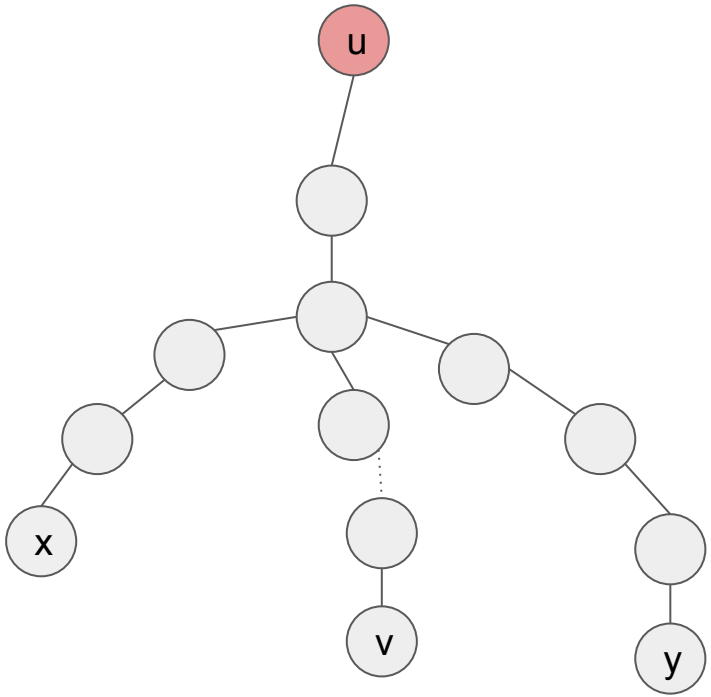
樹的問題 (樹的直徑) 正確性

- 假設樹的直徑為 x - y, 且BFS的起點為x
  - y 一定是 bfs最遠的那個點, 因為是樹直徑



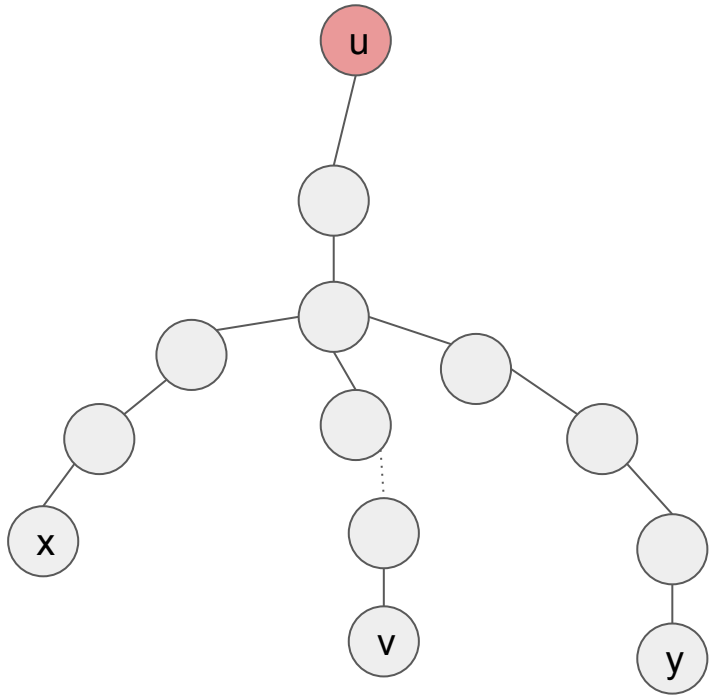
樹的問題 (樹的直徑) 正確性

- 假設樹的直徑為 x - y, 且BFS的起點為x
  - y 一定是 bfs最遠的那個點, 因為是樹直徑
- 假設BFS的起點不是樹直徑的任兩端而是 u



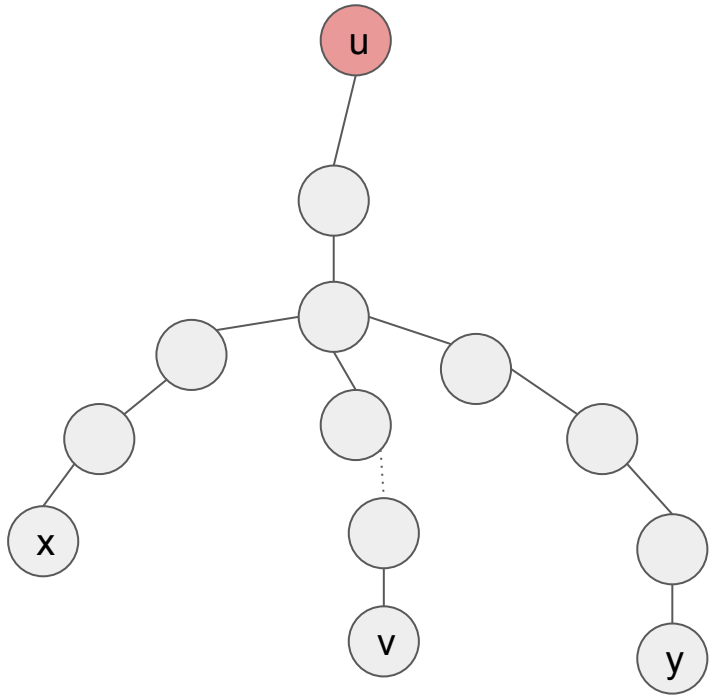
樹的問題 (樹的直徑) 正確性

- 假設樹的直徑為 x - y, 且BFS的起點為x
  - y 一定是 bfs最遠的那個點, 因為是樹直徑
- 假設BFS的起點不是樹直徑的任兩端而是 u
  - 最遠的點不是 x 或 y, 而是 v



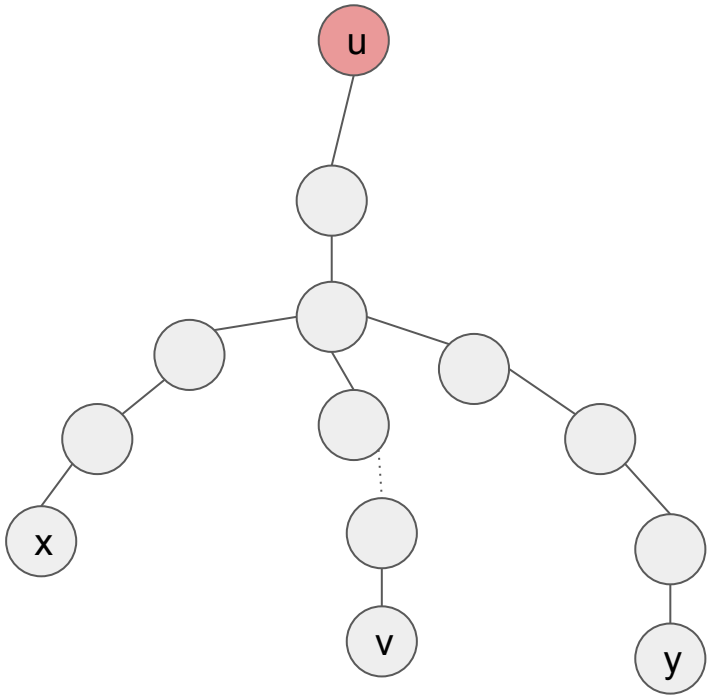
樹的問題 (樹的直徑) 正確性

- 假設樹的直徑為 x - y, 且BFS的起點為x
  - y 一定是 bfs最遠的那個點, 因為是樹直徑
- 假設BFS的起點不是樹直徑的任兩端而是 u
  - 最遠的點不是 x 或 y, 而是 v
  - 那 x - y 一定不是樹直徑, v - y或v - x一定比x - y長



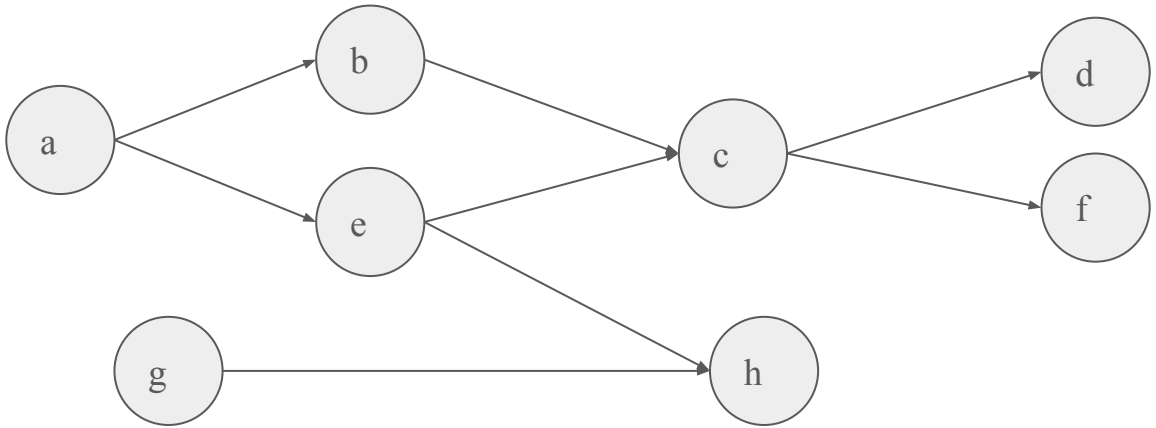
樹的問題 (樹的直徑) 正確性

- 假設**樹的直徑為 x - y**, 且BFS的起點為x
  - y 一定是 bfs最遠的那個點, 因為是樹直徑
- 假設BFS的起點不是樹直徑的任兩端而是 u
  - 最遠的點不是 x 或 y, 而是 v
  - 那 **x - y 一定不是樹直徑**, v - y或v - x一定比x - y長
  - 矛盾, 所以最遠的點一定是 x 或 y。



有向無環圖 DAG

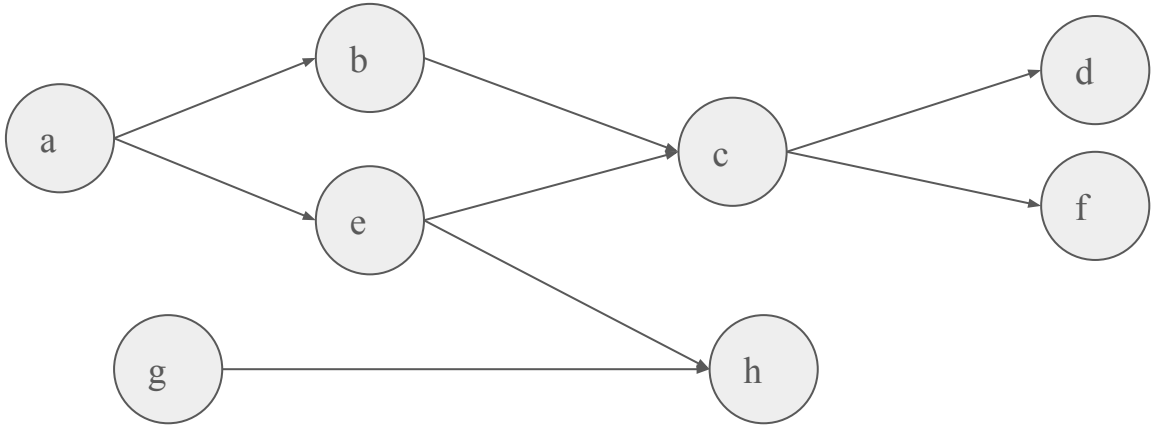
- Directed Acyclic Graph



拓樸排序 Topological Sort

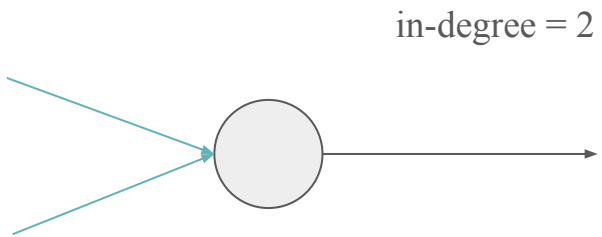
- 找出一種在 DAG 上合理的排列順序
- 方法
  - BFS 拔拔樂
  - DFS 離開點的順序

a b e g h c d f



BFS 拔拔樂

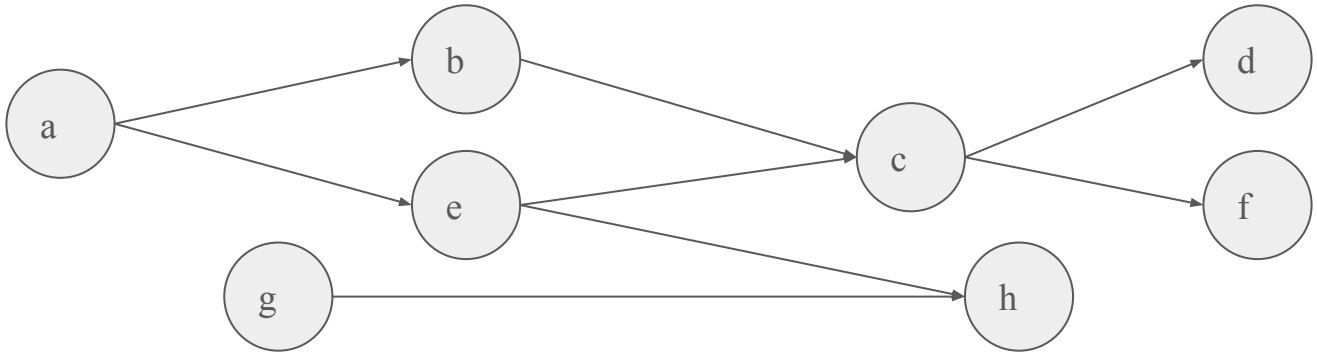
- 每次摘掉 1 個 in-degree 為 0 的起點
- 將它鄰居的 in-degree 都減 1
- 如果遇到減完後 in-degree 變成 0 的就丟進 queue 裡面



BFS 拔拔樂

topo-sort	

node	a	b	c	d	e	f	g	h
in-degree	0	1	2	1	1	1	0	2

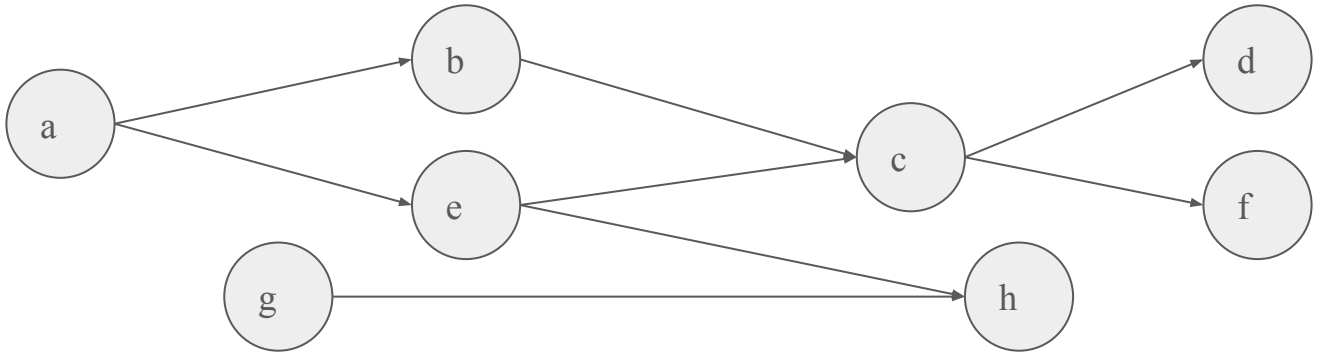


queue

BFS 拔拔樂

topo-sort	

node	a	b	c	d	e	f	g	h
in-degree	0	1	2	1	1	1	0	2

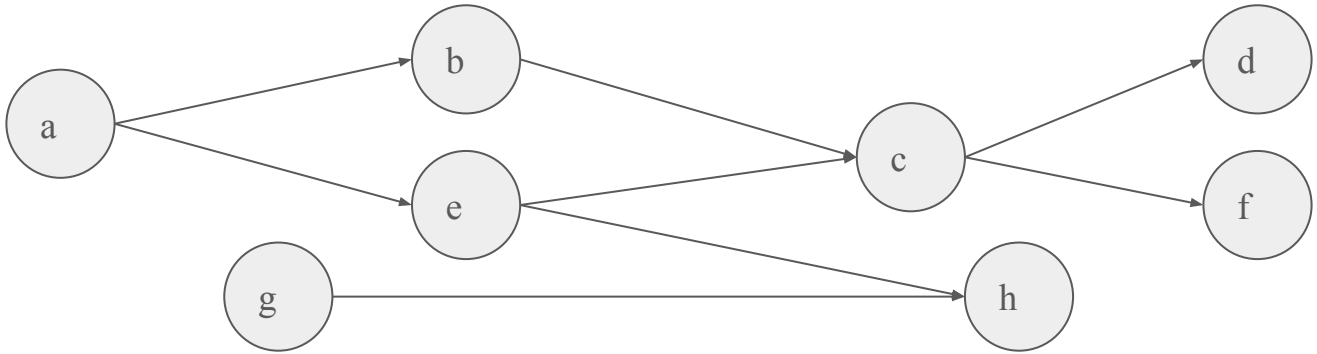


queue	a	

BFS 拔拔樂

topo-sort	

node	a	b	c	d	e	f	g	h
in-degree	0	1	2	1	1	1	0	2

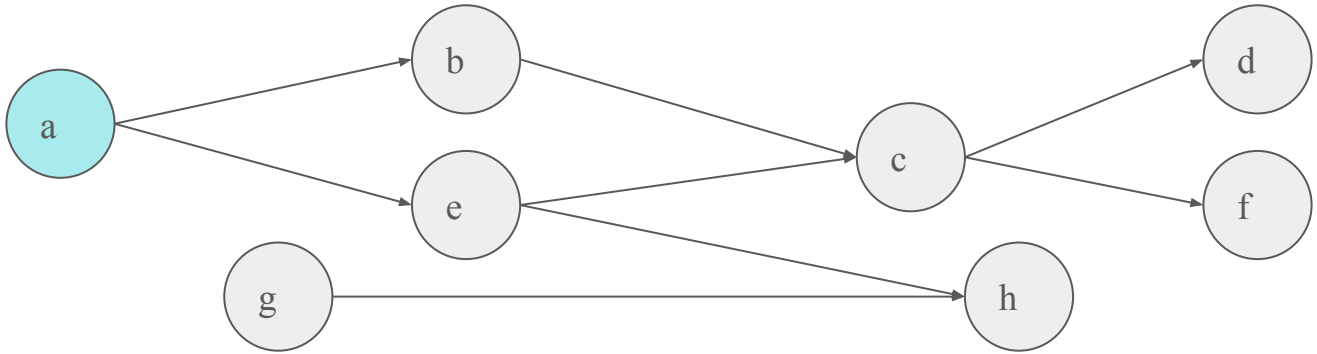


queue	a	g	

BFS 拔拔樂

topo-sort	a

node	a	b	c	d	e	f	g	h
in-degree	0	1	2	1	1	1	0	2

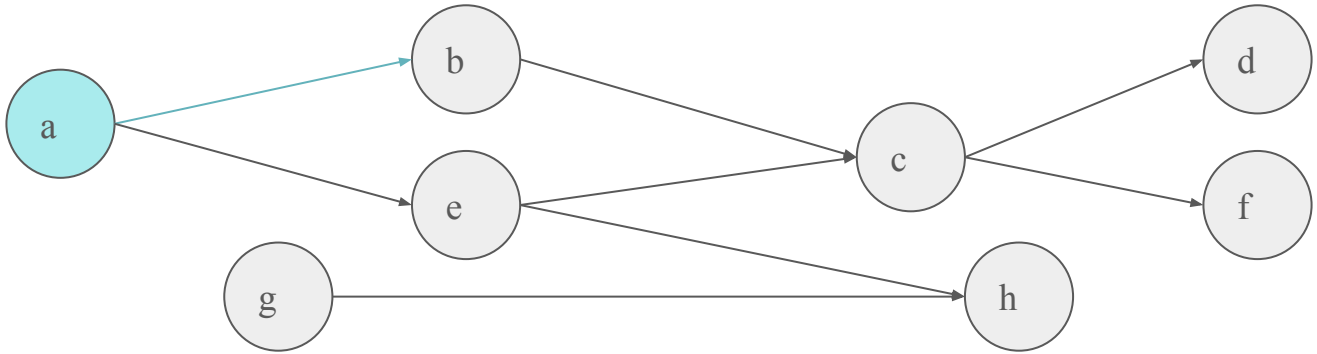


queue	a	g

BFS 拔拔樂

topo-sort	a

node	a	b	c	d	e	f	g	h
in-degree	0	1	2	1	1	1	0	2



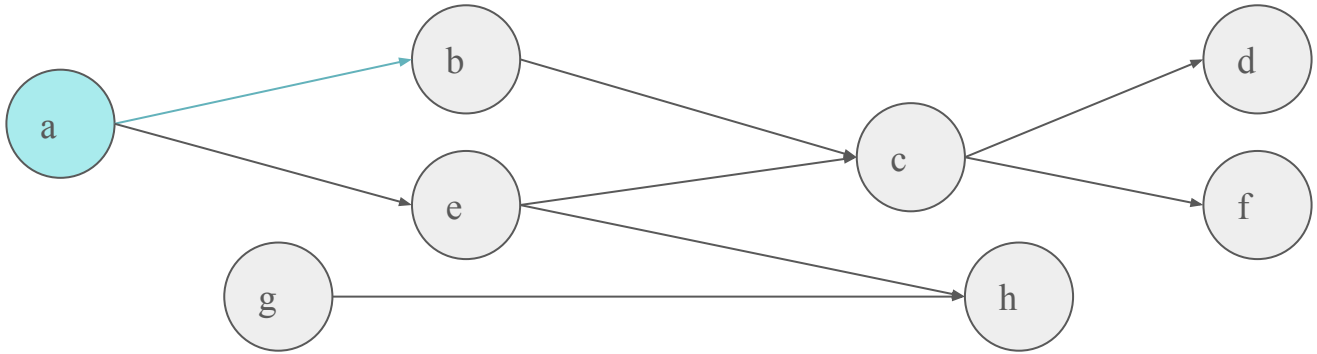
queue

g	

BFS 拔拔樂

topo-sort	a

node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	1	1	0	2



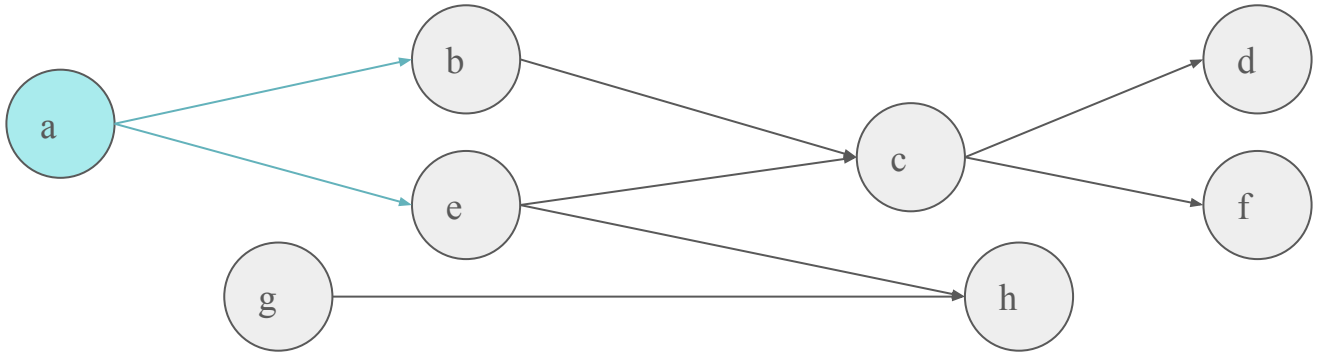
queue

g	b	

BFS 拔拔樂

topo-sort	a

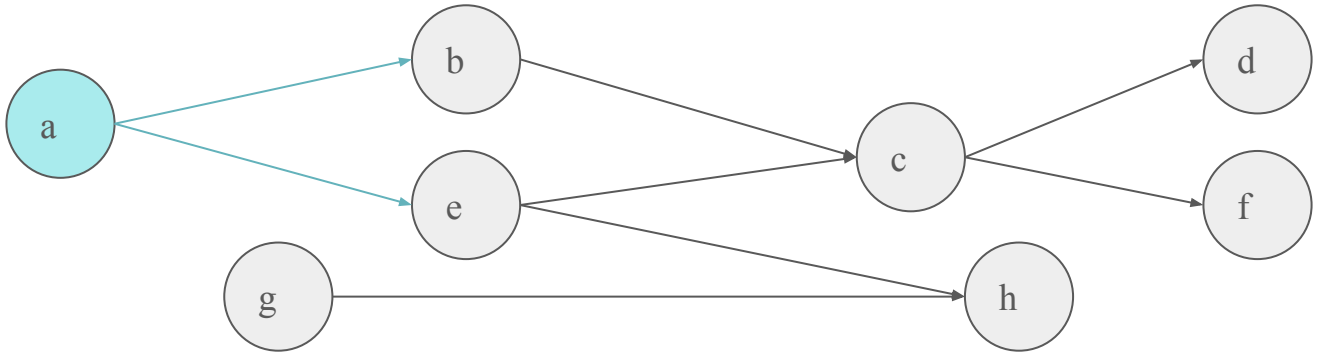
node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	1	1	0	2



BFS 拔拔樂

topo-sort	a

node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	0	1	0	2



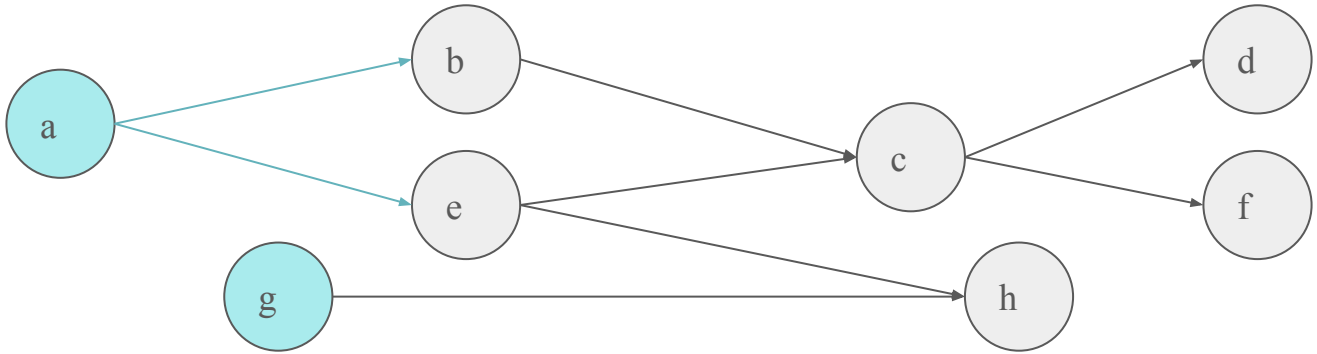
queue

g	b	e	

BFS 拔拔樂

topo-sort	a	g						

node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	0	1	0	2

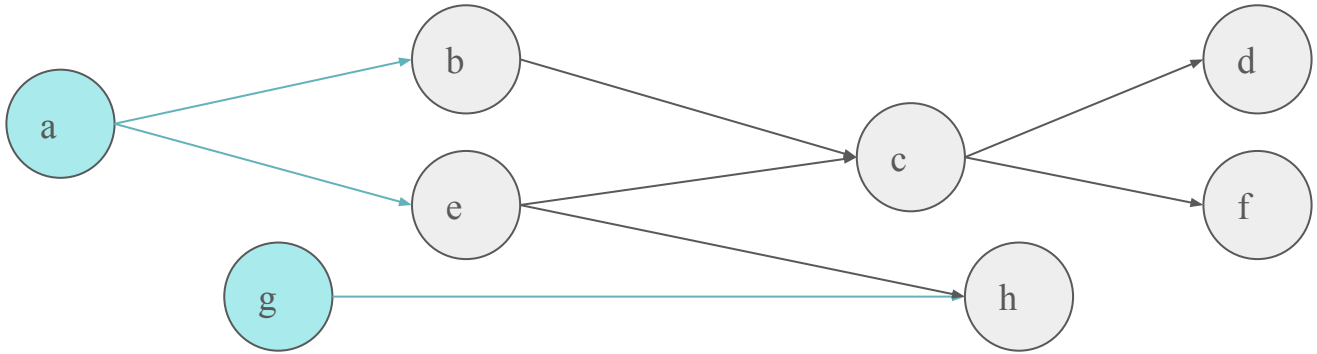


queue	g	b	e					

BFS 拔拔樂

topo-sort	a g

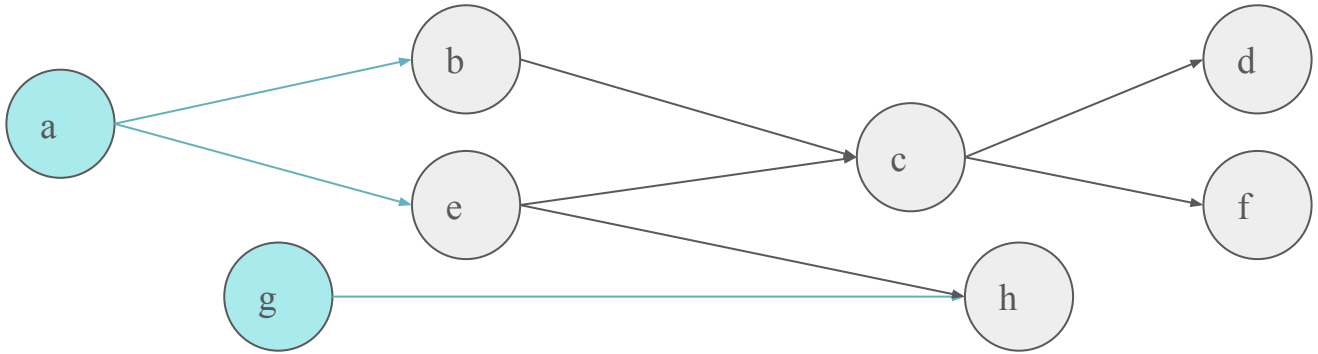
node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	0	1	0	2



BFS 拔拔樂

topo-sort	a g

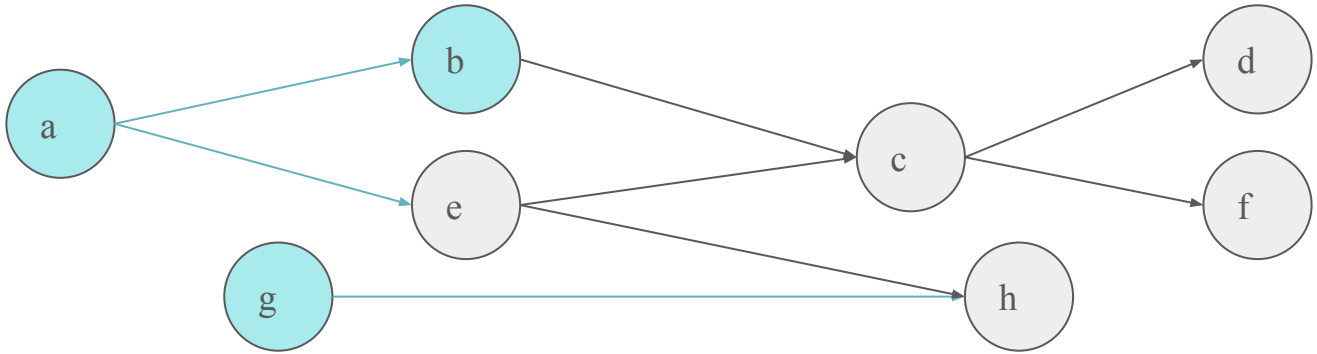
node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	0	1	0	1



BFS 拔拔樂

topo-sort	a g b

node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	0	1	0	1



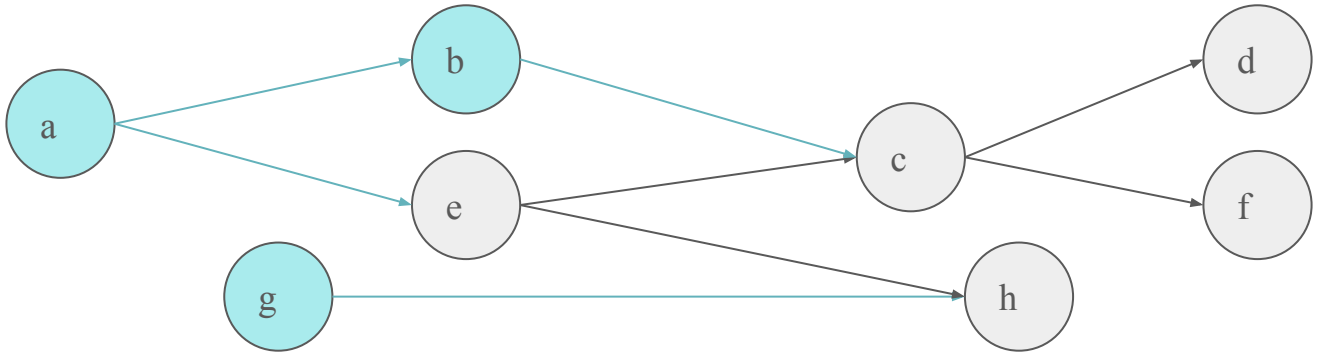
queue

b	e

BFS 拔拔樂

topo-sort	a g b

node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	0	1	0	1

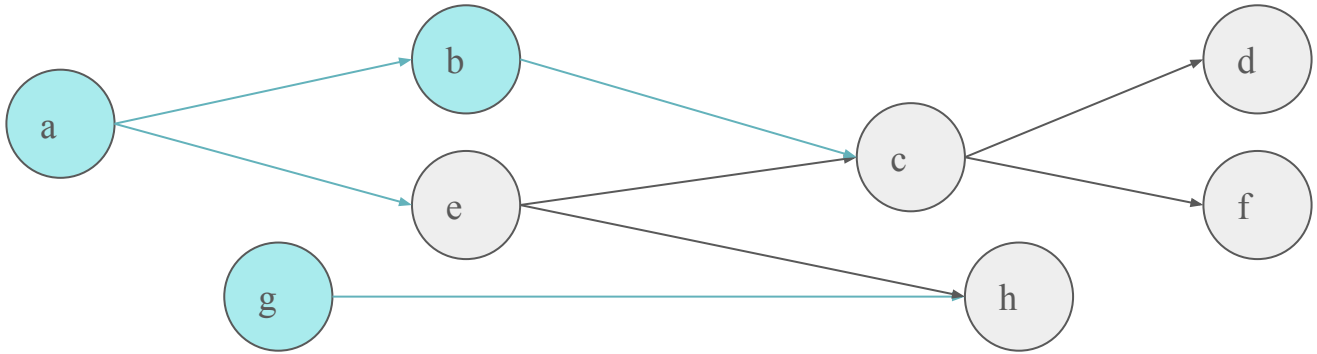


queue

BFS 拔拔樂

topo-sort	a g b

node	a	b	c	d	e	f	g	h
in-degree	0	0	1	1	0	1	0	1

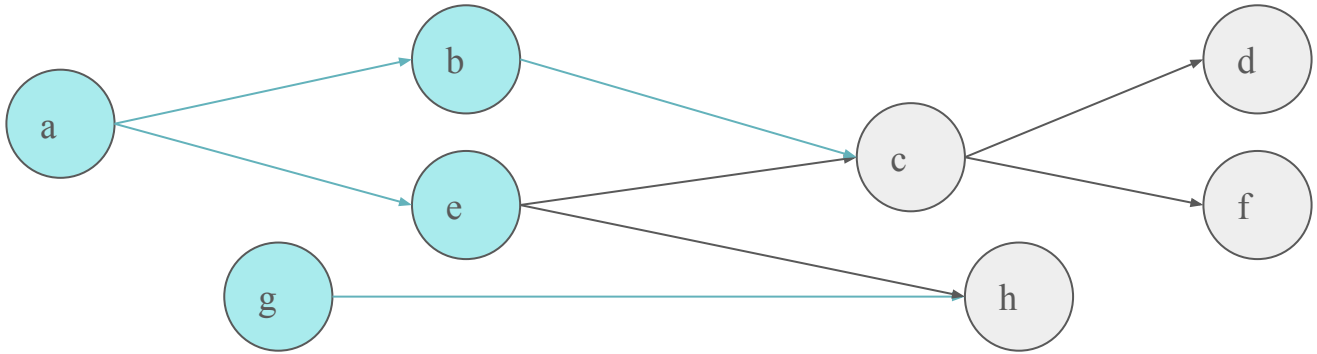


queue

BFS 拔拔樂

topo-sort	a g b e

node	a	b	c	d	e	f	g	h
in-degree	0	0	1	1	0	1	0	1

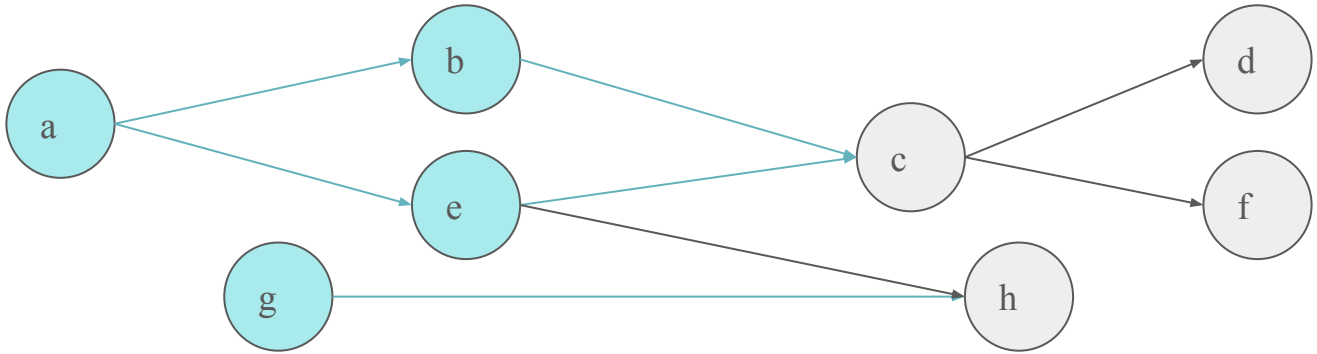


queue	e

BFS 拔拔樂

topo-sort	a g b e

node	a	b	c	d	e	f	g	h
in-degree	0	0	1	1	0	1	0	1

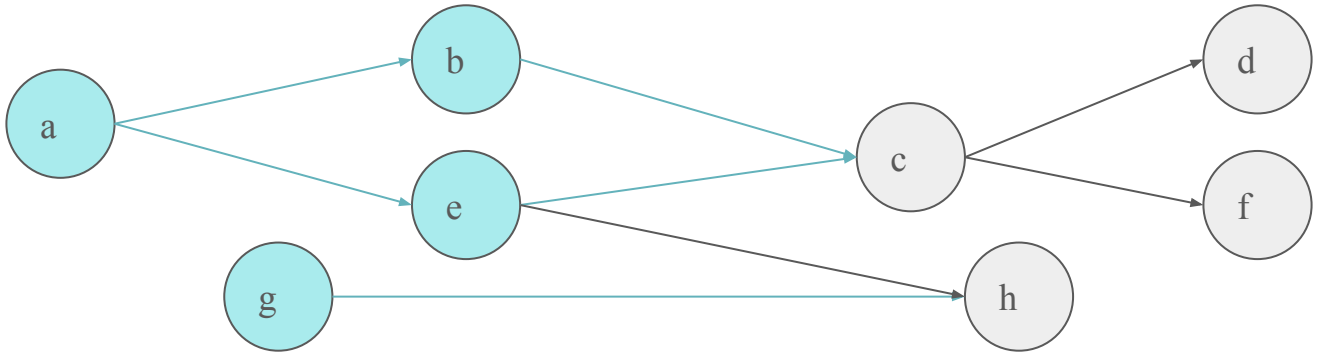


queue

BFS 拔拔樂

topo-sort	a g b e

node	a	b	c	d	e	f	g	h
in-degree	0	0	0	1	0	1	0	1

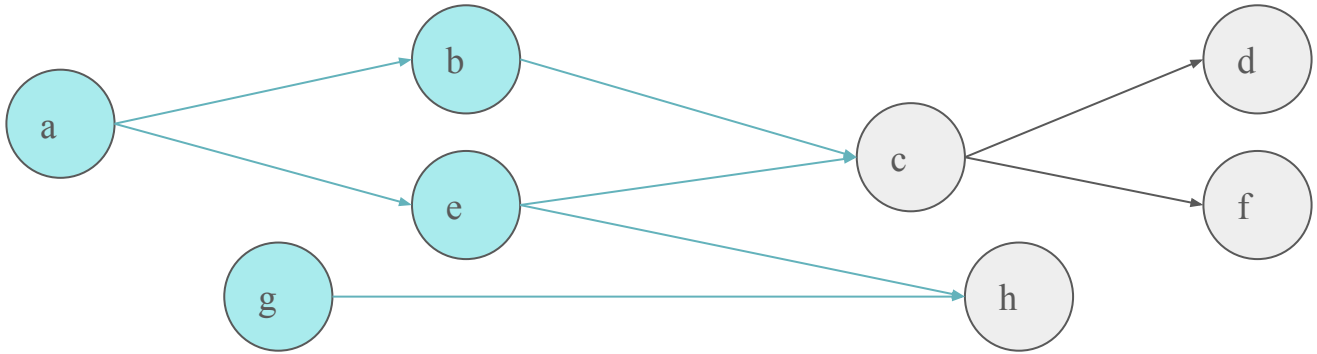


queue

BFS 拔拔樂

topo-sort	a g b e

node	a	b	c	d	e	f	g	h
in-degree	0	0	0	1	0	1	0	1

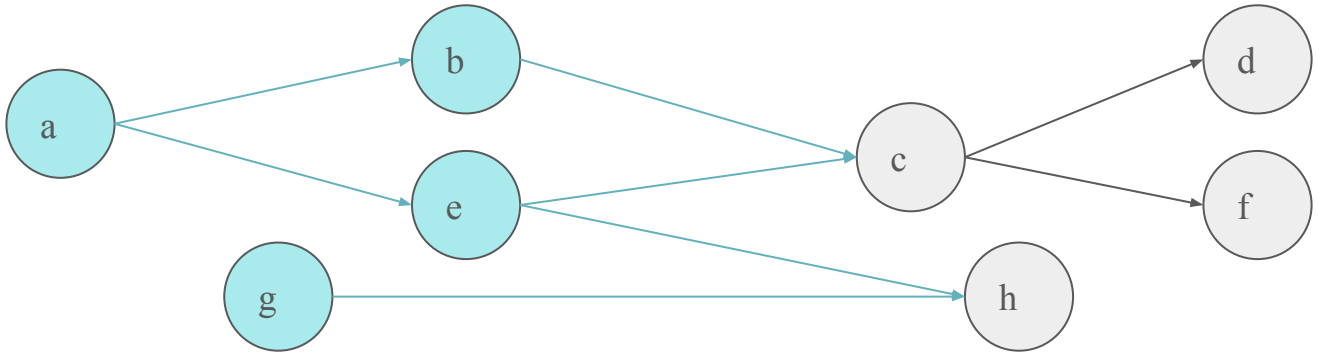


queue

BFS 拔拔樂

topo-sort	a g b e

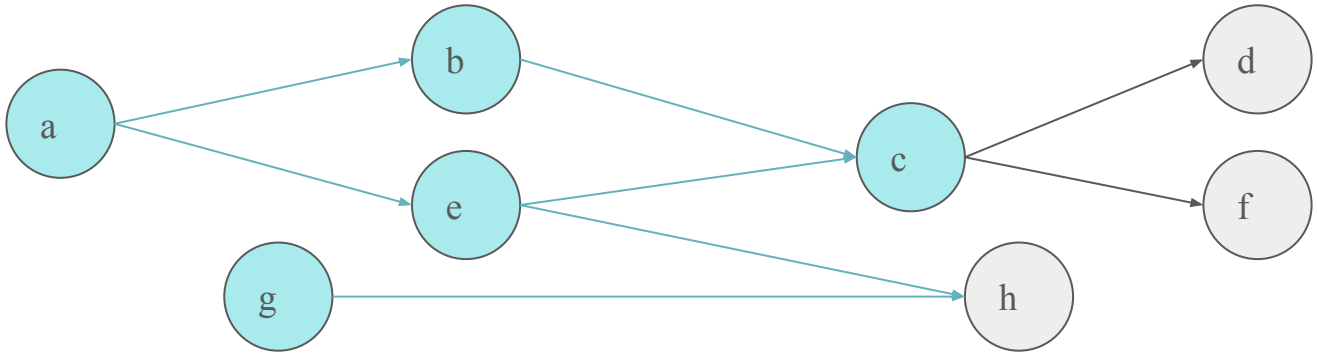
node	a	b	c	d	e	f	g	h
in-degree	0	0	0	1	0	1	0	0



BFS 拔拔樂

topo-sort	a g b e c

node	a	b	c	d	e	f	g	h
in-degree	0	0	0	1	0	1	0	0

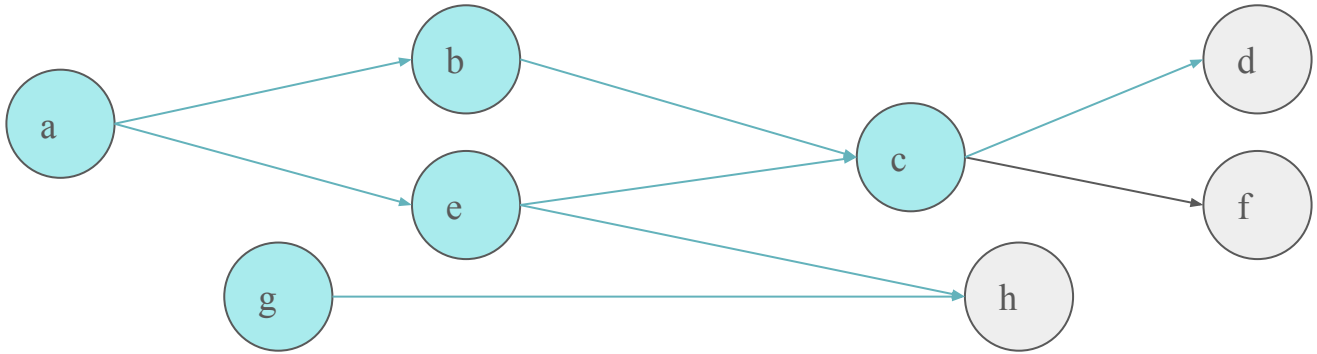


queue	c	h

BFS 拔拔樂

topo-sort	a g b e c

node	a	b	c	d	e	f	g	h
in-degree	0	0	0	0	0	1	0	0

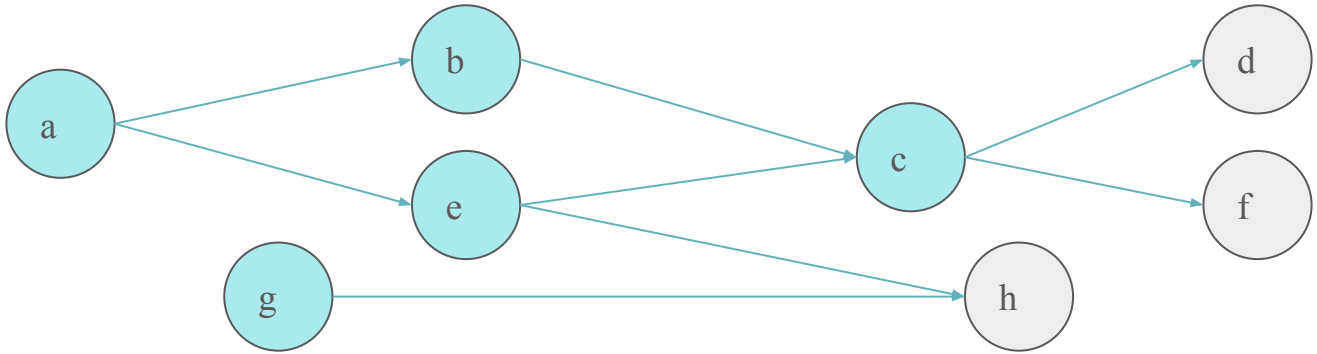


queue

BFS 拔拔樂

topo-sort	a g b e c

node	a	b	c	d	e	f	g	h
in-degree	0	0	0	0	0	0	0	0



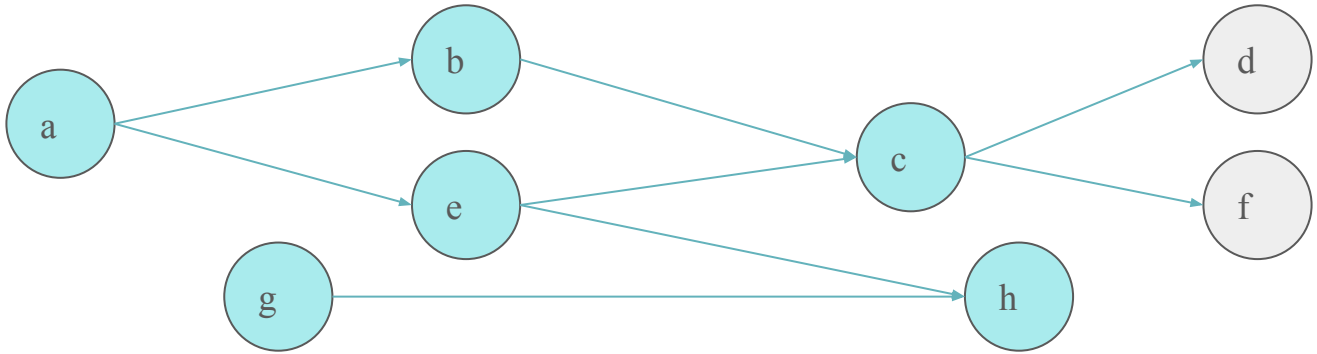
queue

h	d	f	

BFS 拔拔樂

topo-sort	a g b e c h

node	a	b	c	d	e	f	g	h
in-degree	0	0	0	0	0	0	0	0

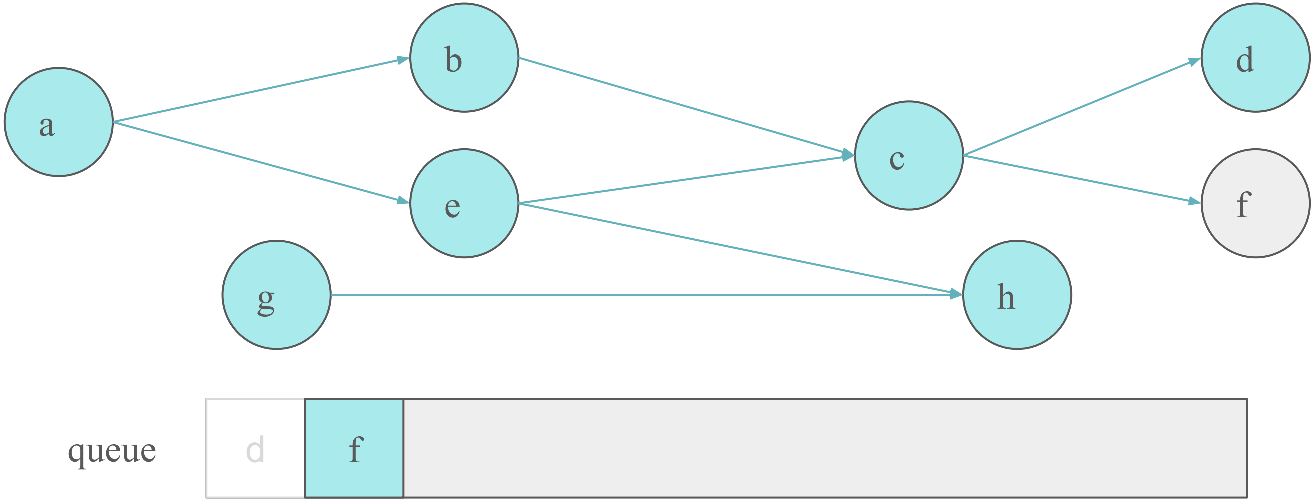


queue | h | d | f |

BFS 拔拔樂

topo-sort	a g b e c h d

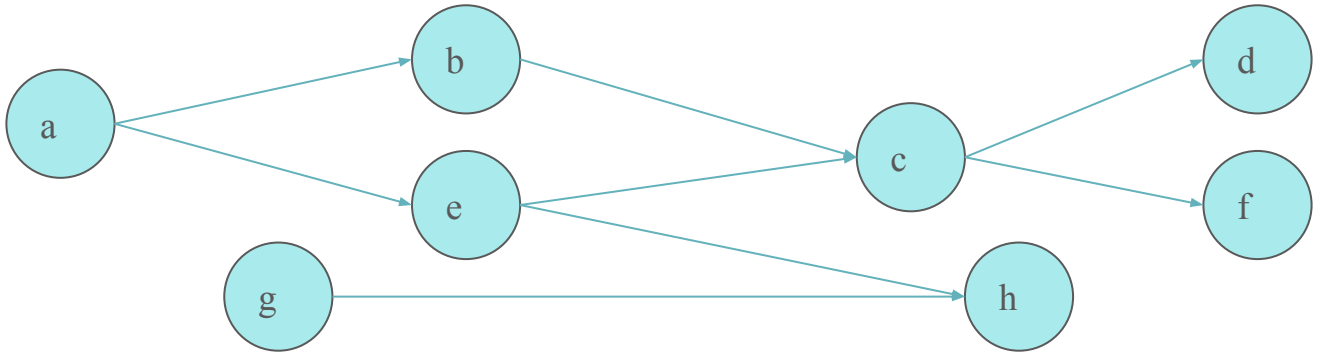
node	a	b	c	d	e	f	g	h
in-degree	0	0	0	0	0	0	0	0



BFS 拔拔樂

topo-sort	a g b e c h d f

node	a	b	c	d	e	f	g	h
in-degree	0	0	0	0	0	0	0	0



queue | f |

BFS 拔拔樂 Code

```
 5 const int MAXN = 1e3 + 5;
 6 int n;                                  // 有幾個節點
 7 vector<int> G[MAXN];
 8 int indegree[MAXN];                     // 存放indegree
 9 void init() {
10     memset(indegree, 0, sizeof(indegree));
11 }
12 void addEdge(int u, int v) {
13     G[u].push_back(v);
14     indegree[v]++;                     // 加邊記得維護indegree
15 }
```

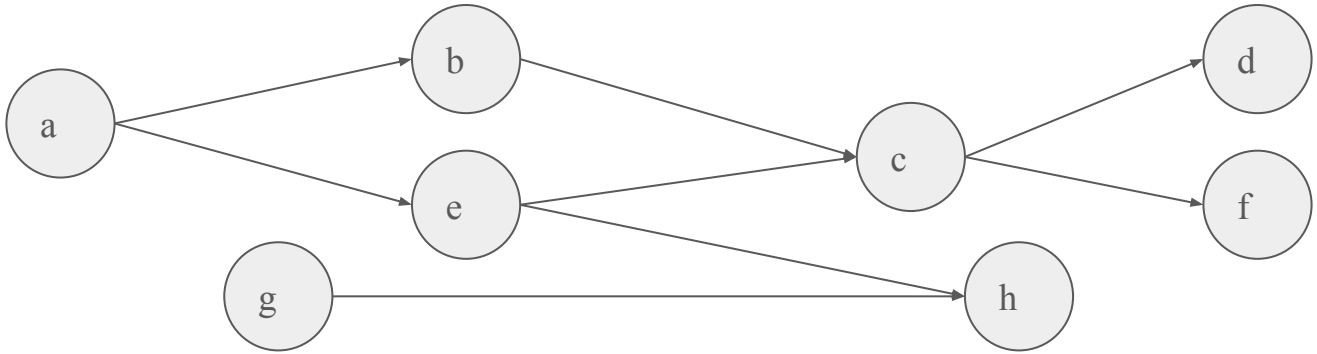
BFS 拔拔樂 Code

```
16 void BFS_topology() {
17     queue<int> q;
18     for (int i = 0 ; i < n ; i++)
19         if (indegree[i] == 0)          // 若node[i] indegree為0
20             q.push(i);                 // 放入queue中
21     vector<int> topology;              // 存放拓樸排序
22     while (q.size()) {                 // 當queue還沒空就繼續做
23         int u = q.front(); q.pop();
24         topology.push_back(u);         // 拔拔樂的點放進去topology裡面
25         for (auto &v : G[u]) {
26             indegree[v]--;             // 維護好相連節點的indegree
27             if (indegree[v] == 0)      // 如果indegree變成0了
28                 q.push(v);             // 丟進去queue裡面變成被拔拔樂的點
29         }
30     }
31 }
```

DFS 點離開的順序

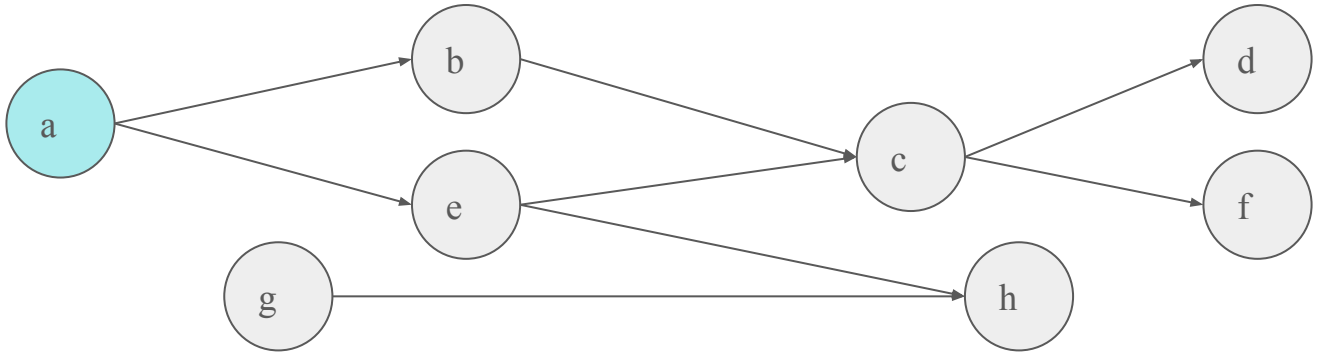
- finish-time
  - 點離開的順序
- 如何判斷:
  - 到了某個點之後沒有路可以再走下去了
  - 可以直接走到的 node 都 visit 過了或沒有邊接出去
- Topo-sort 即為 finish-time 順序的 reverse

DFS 離開順序



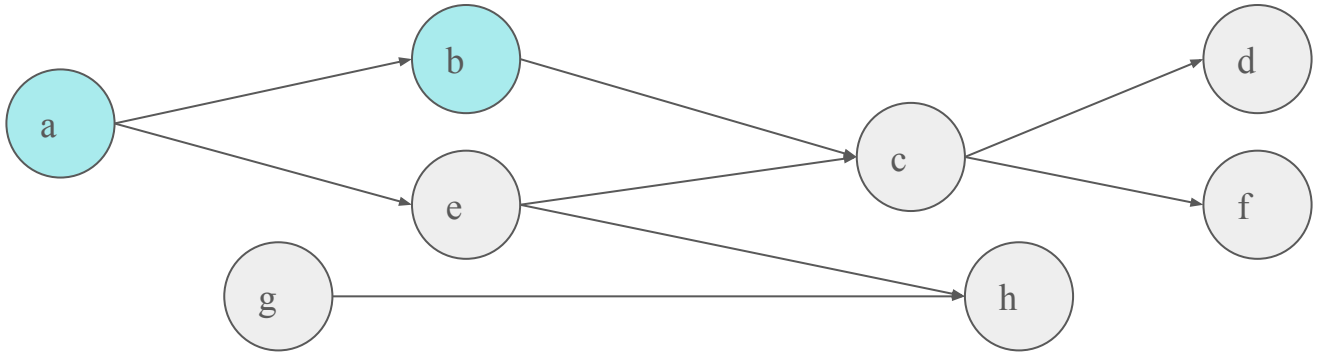
finish-time	

DFS 離開順序



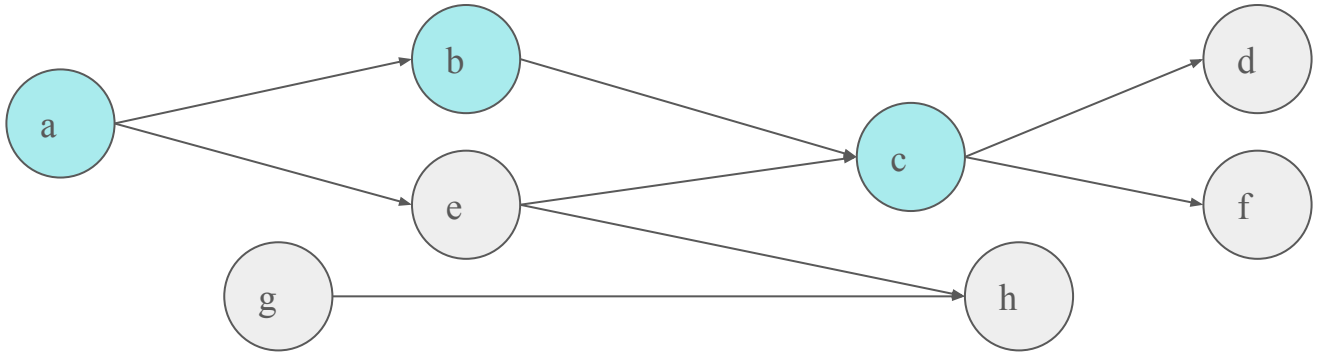
finish-time	

DFS 離開順序



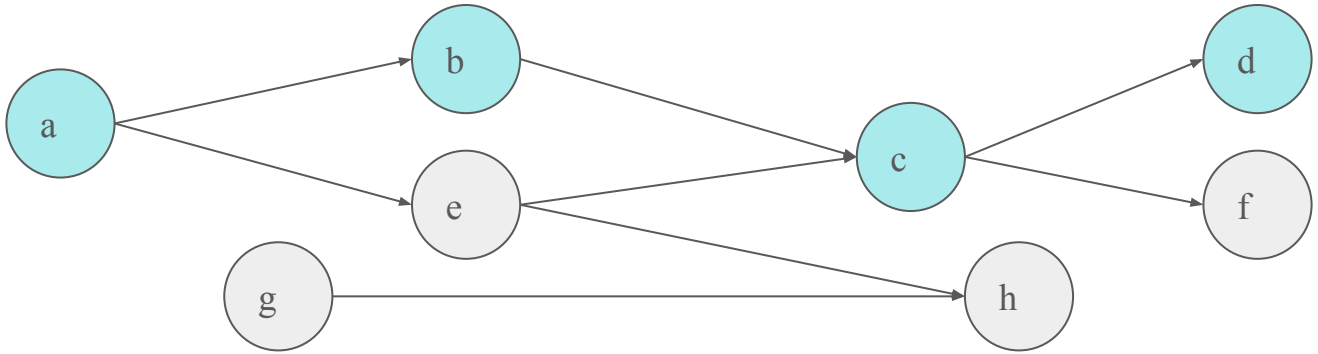
finish-time	

DFS 離開順序



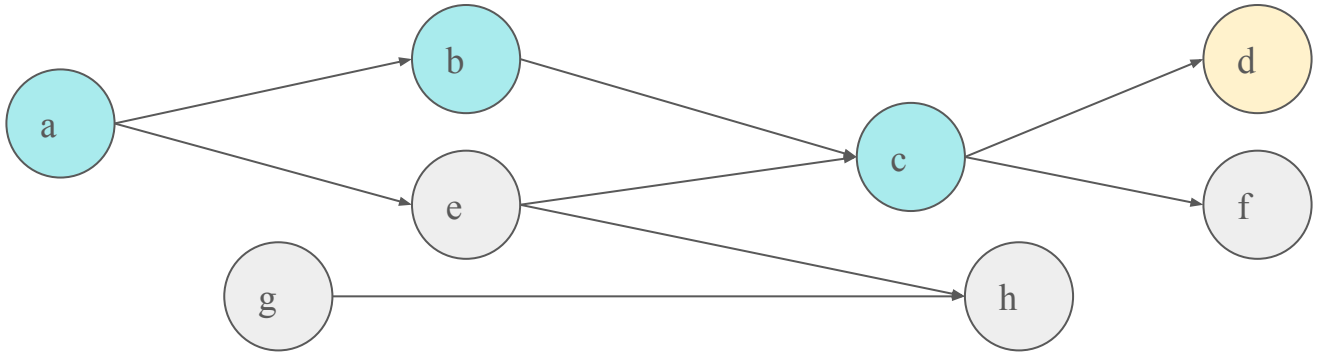
finish-time	

DFS 離開順序



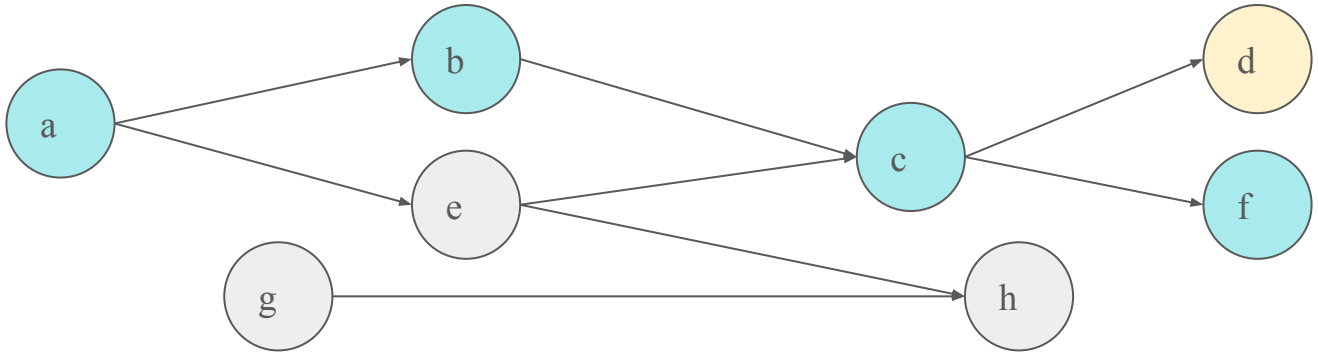
finish-time	

DFS 離開順序



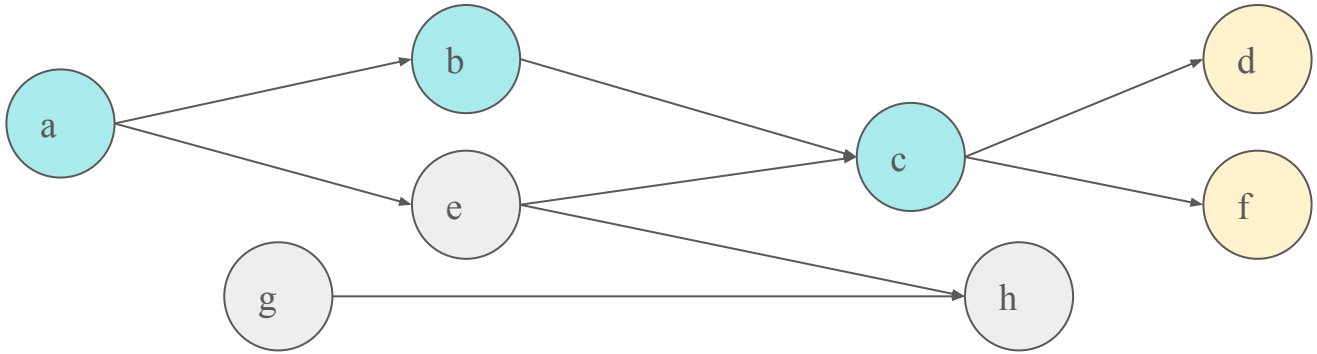
finish-time	d

DFS 離開順序



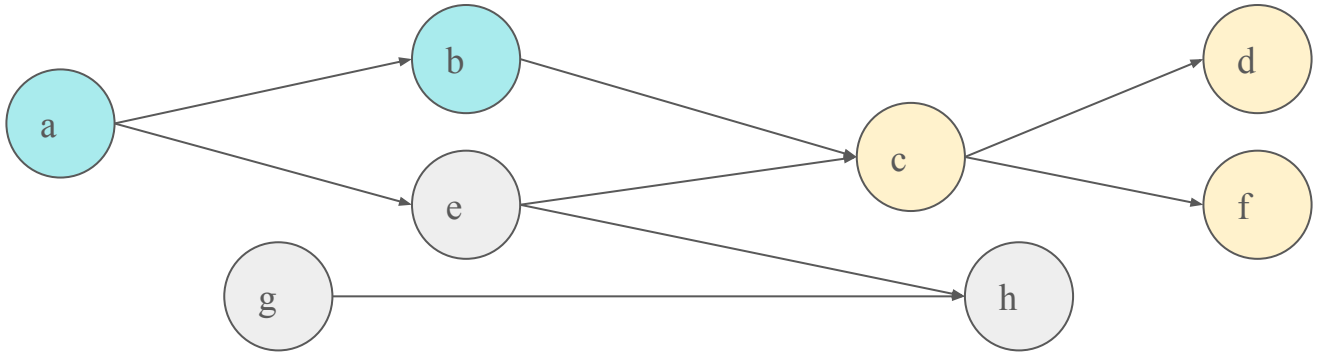
finish-time	d

DFS 離開順序



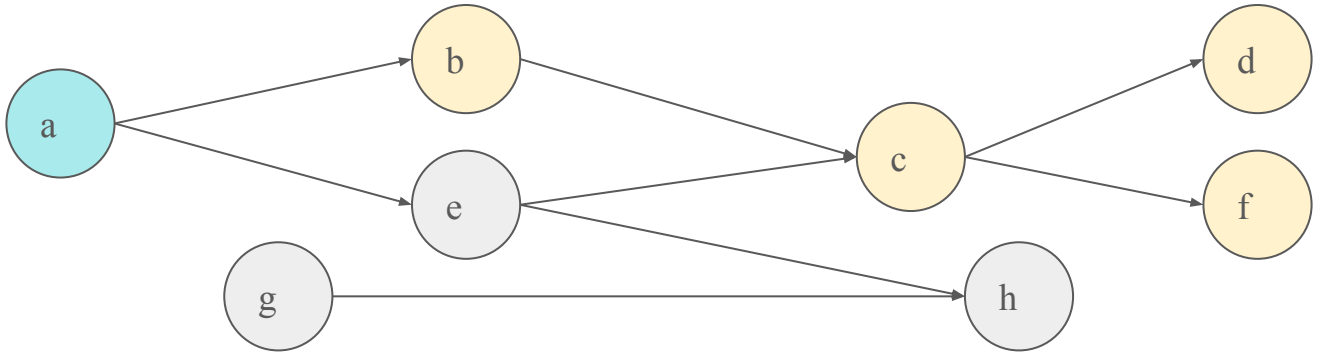
finish-time	d f

DFS 離開順序



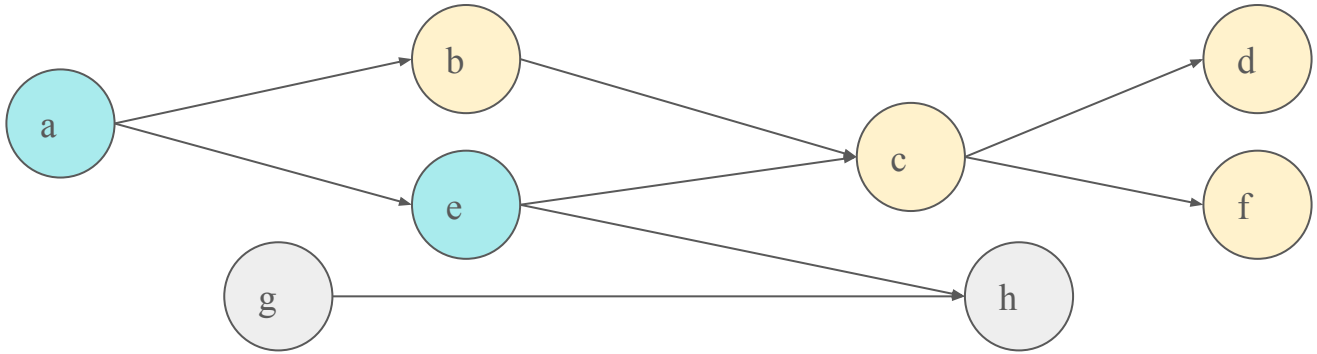
finish-time	d f c

DFS 離開順序



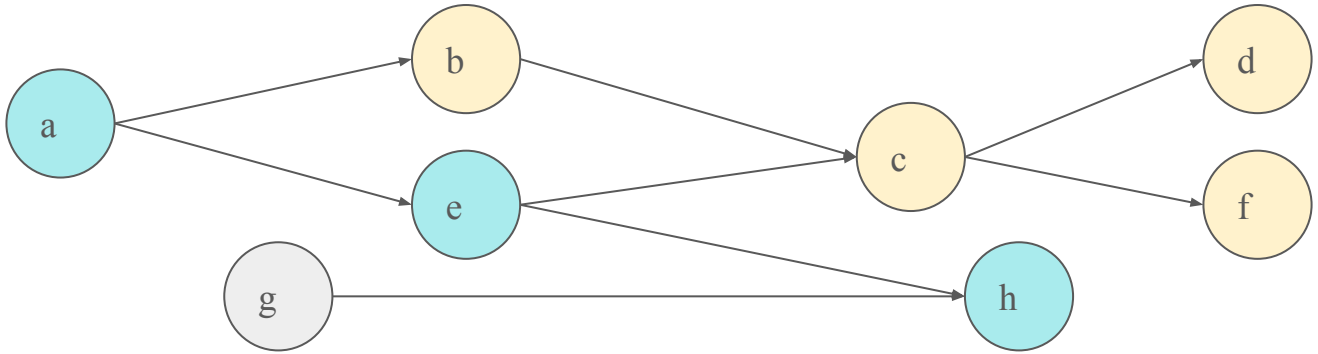
finish-time	d f c b

DFS 離開順序



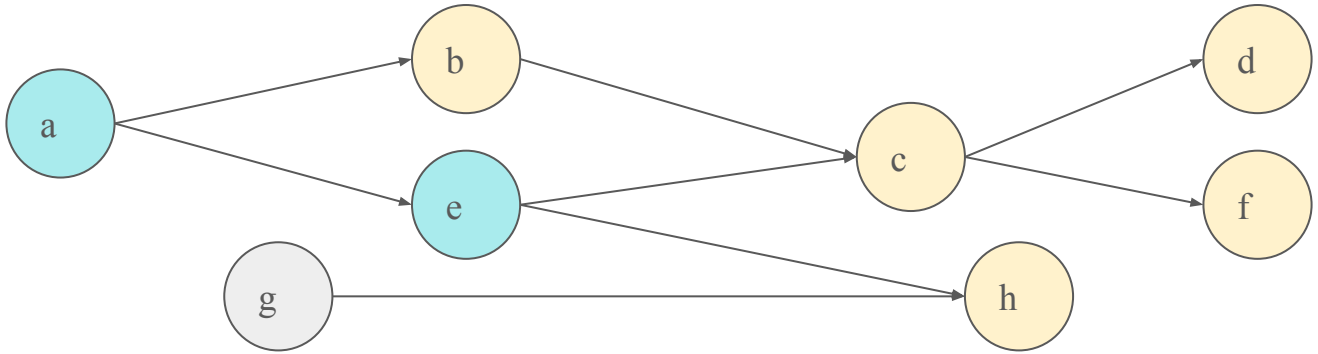
finish-time	d f c b

DFS 離開順序



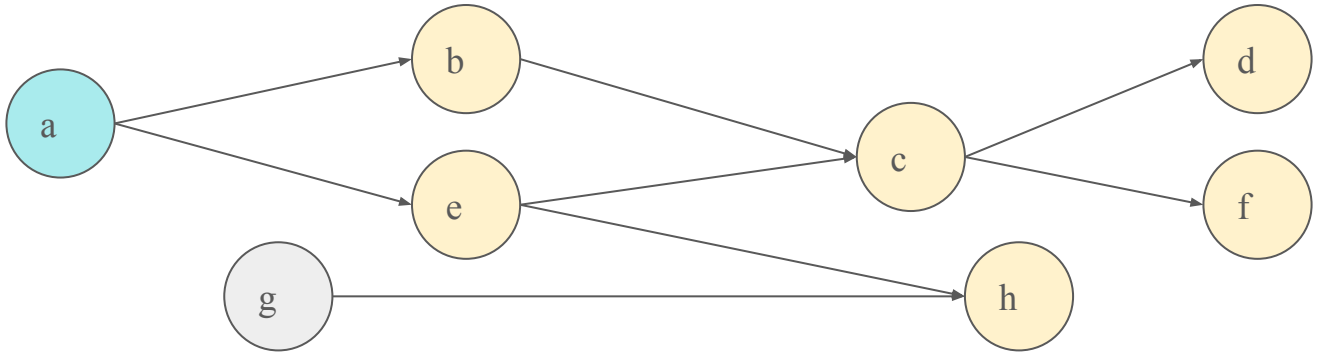
finish-time	d f c b

DFS 離開順序



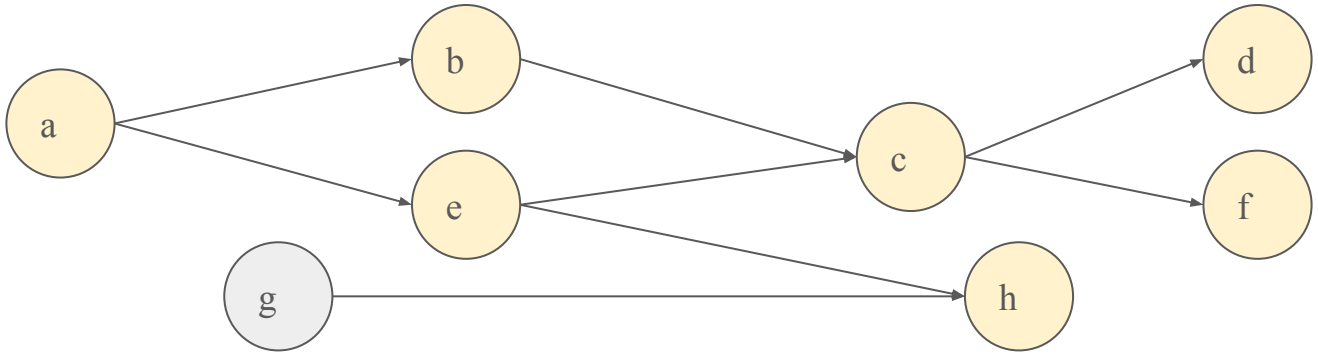
finish-time	d f c b h

DFS 離開順序



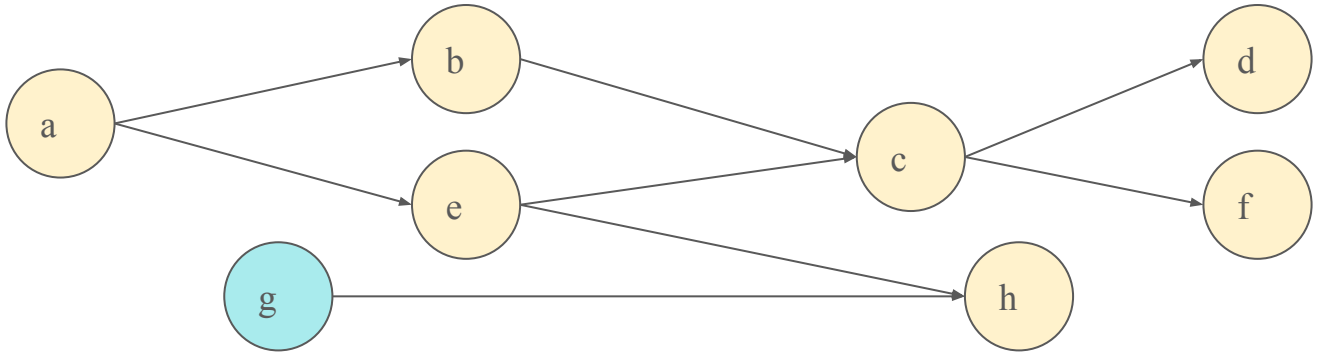
finish-time	d f c b h e

DFS 離開順序



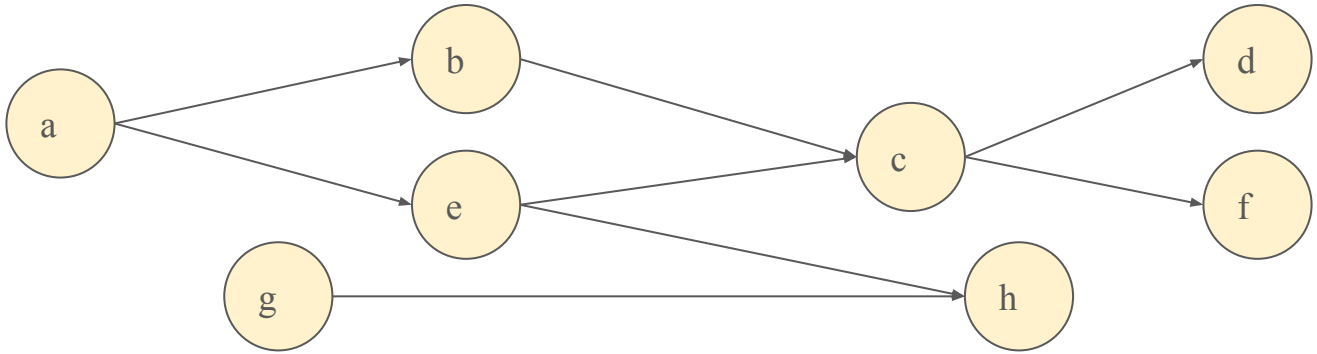
finish-time	d f c b h e a

DFS 離開順序



finish-time	d f c b h e a

DFS 離開順序



finish-time	d f c b h e a g

DFS 離開順序

finish-time	d f c b h e a g

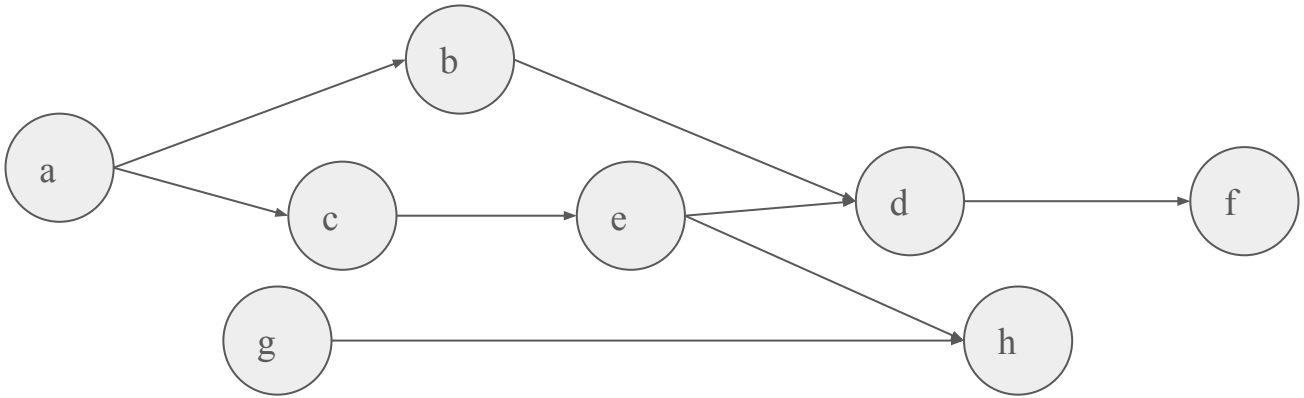
topo-sort	g a e h b c f d

DFS 離開順序 Code

```cpp
const int MAXN = 1e3 + 5;
vector<int> G[MAXN];
vector<int> topology;                                   // 拓樸排序
bool visited[MAXN];                                     // 有沒有被拜訪的陣列
int n;                                                  // 有幾個節點
void init() {
    memset(visited, false, sizeof(visited));
}
void dfs(int u) {
    visited[u] = true;                                  // 設定為已經拜訪
    for (auto &v : G[u])                                // 列舉每個相鄰的節點
        dfs(v);                                         // 用dfs走訪他
    topology.push_back(u);                              // 儲存 finish time (outStamp) 的順序
}
void dfs_topology() {
    for (int i = 0 ; i < n ; i++)                       // 列舉每個點
        if (!visited[i])                                // 如果遇到還沒有走訪的節點
            dfs(i);                                     // 用dfs走訪他
    reverse(topology.begin(), topology.end());          // finish time (outStamp) 的編號反過來
}
```

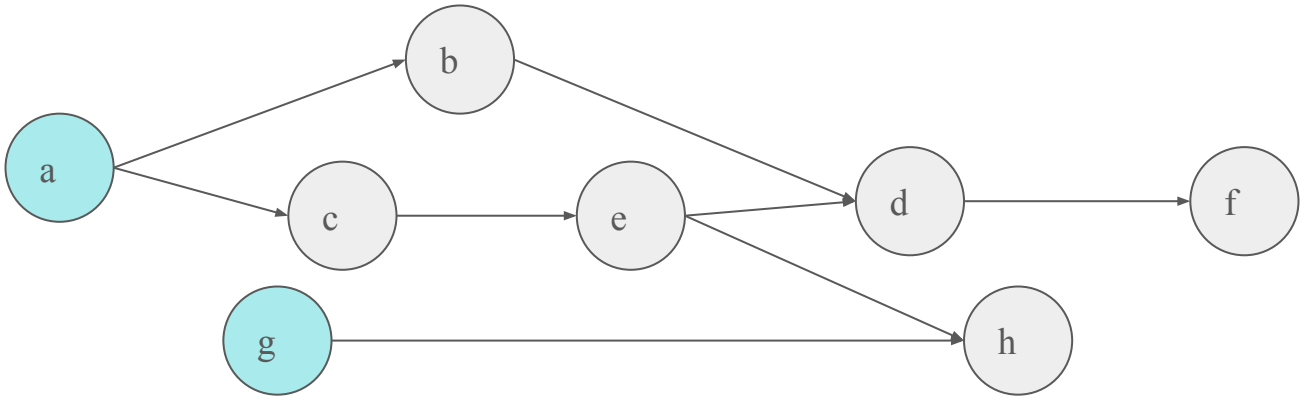
DAG 最長路

node	a	b	c	d	e	f	g	h
dist	-1	-1	-1	-1	-1	-1	-1	-1



DAG 最長路

node	a	b	c	d	e	f	g	h
dist	0	-1	-1	-1	-1	-1	0	-1



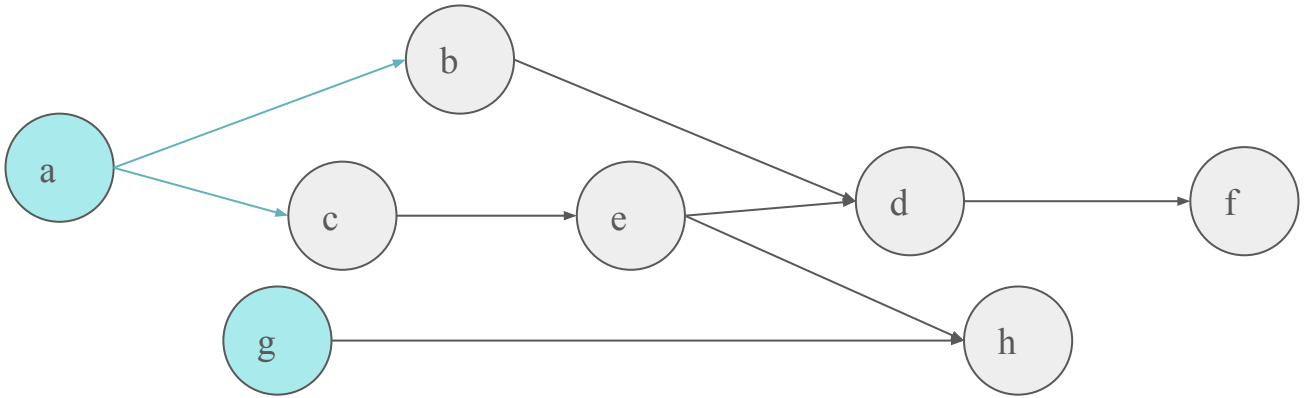
DAG 最長路

dist[x] = max(dist[x], dist[parent[x]] + 1)

dist[b] = max(dist[b], dist[a] + 1)

dist[c] = max(dist[c], dist[a] + 1)

node	a	b	c	d	e	f	g	h
dist	0	1	1	-1	-1	-1	0	-1

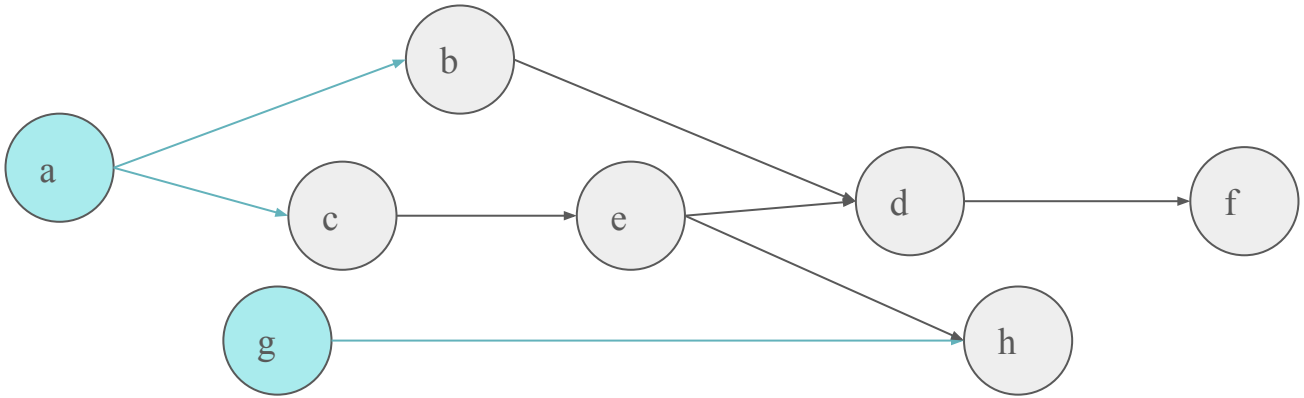


DAG 最長路

dist[x] = max(dist[x], dist[parent[x]] + 1)

dist[h] = max(dist[h], dist[g] + 1)

node	a	b	c	d	e	f	g	h
dist	0	1	1	-1	-1	-1	0	1

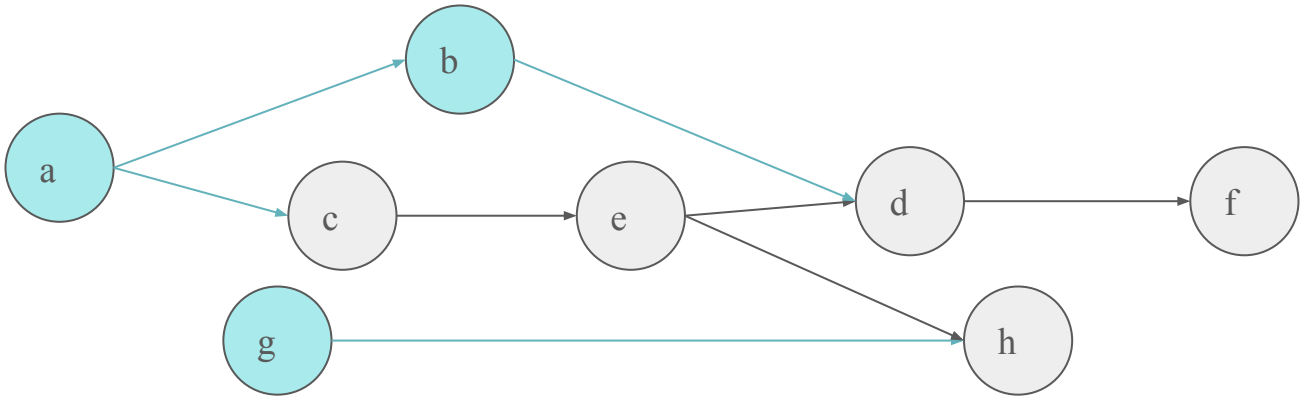


DAG 最長路

dist[x] = max(dist[x], dist[parent[x]] + 1)

dist[d] = max(dist[d], dist[b] + 1)

node	a	b	c	d	e	f	g	h
dist	0	1	1	2	-1	-1	0	1

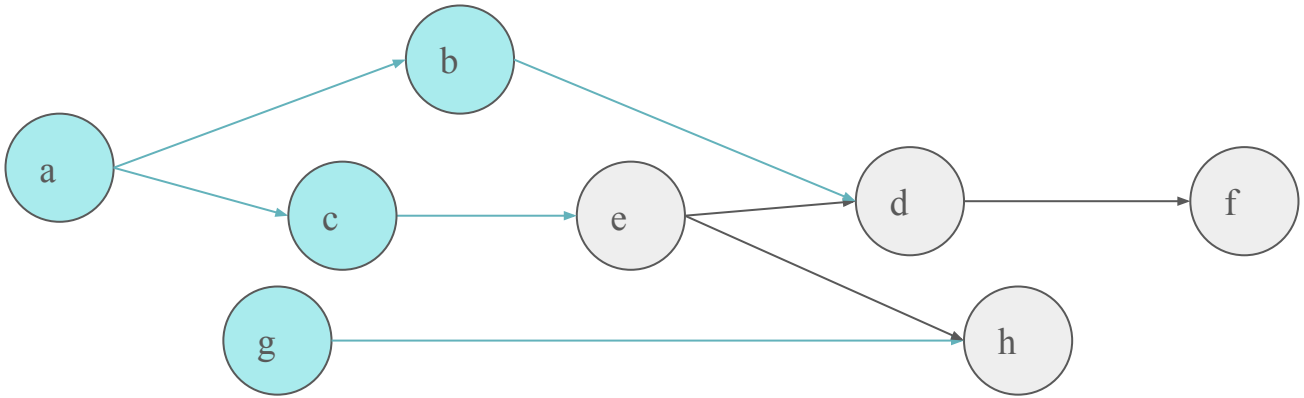


DAG 最長路

dist[x] = max(dist[x], dist[parent[x]] + 1)

dist[e] = max(dist[e], dist[c] + 1)

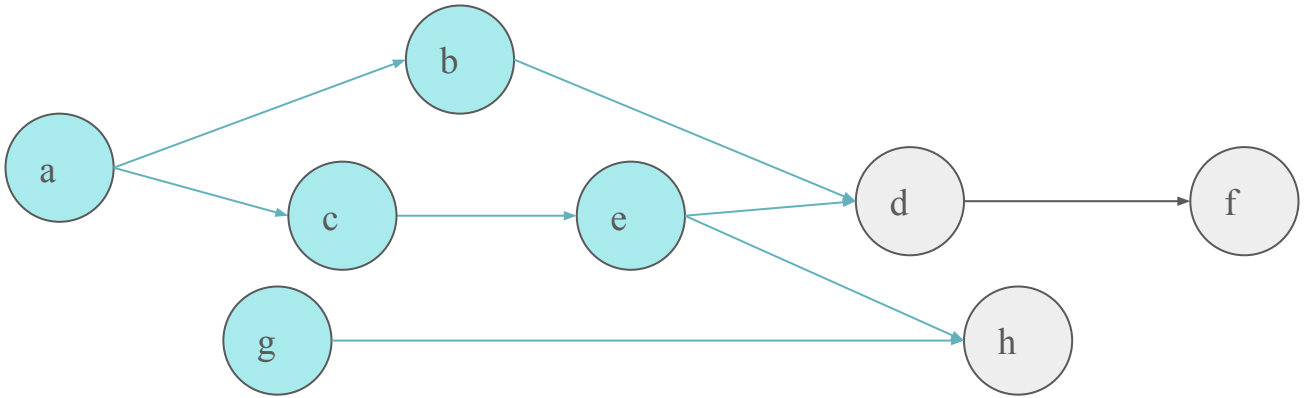
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	-1	0	1



DAG 最長路

dist[x] = max(dist[x], dist[parent[x]] + 1)
dist[d] = max(dist[d], dist[e] + 1)
dist[h] = max(dist[h], dist[e] + 1)

node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	-1	0	3

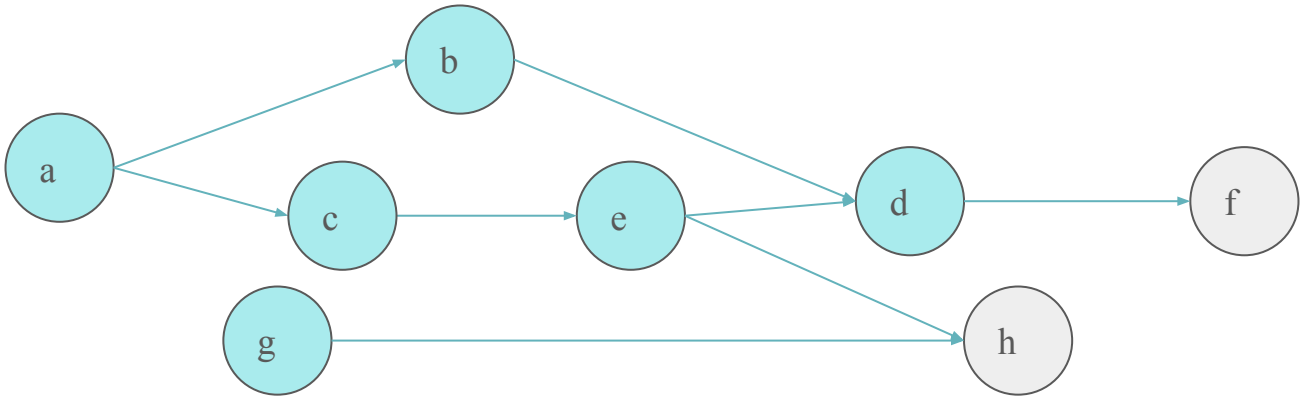


DAG 最長路

dist[x] = max(dist[x], dist[parent[x]] + 1)

dist[f] = max(dist[f], dist[d] + 1)

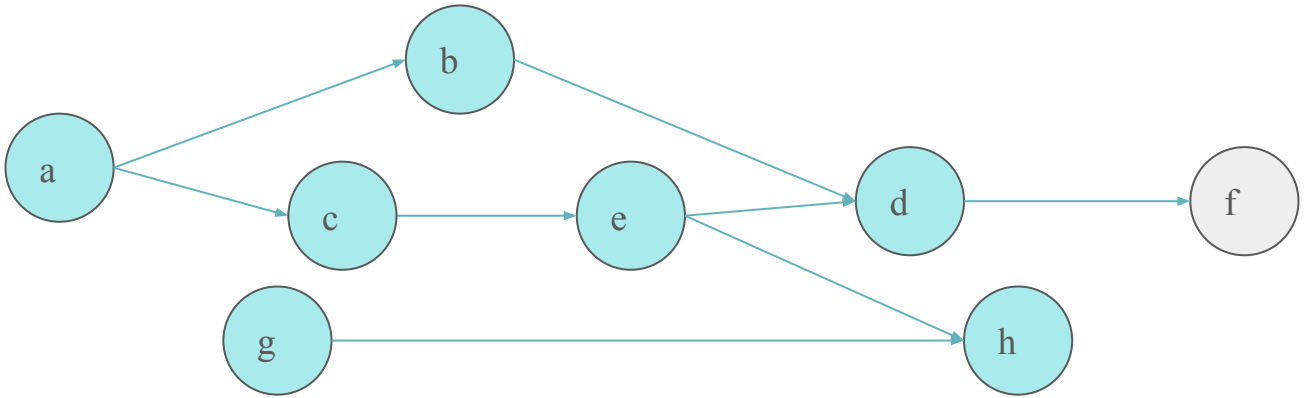
node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	4	0	3



DAG 最長路

$$dist[x] = \max(dist[x], dist[parent[x]] + 1)$$

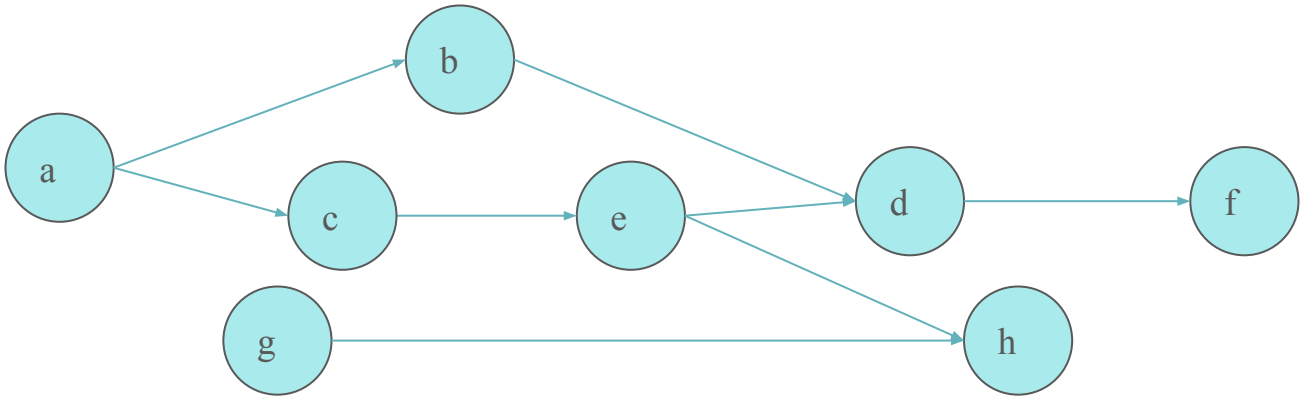
node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	4	0	3



DAG 最長路

$$dist[x] = \max(dist[x], dist[parent[x]] + 1)$$

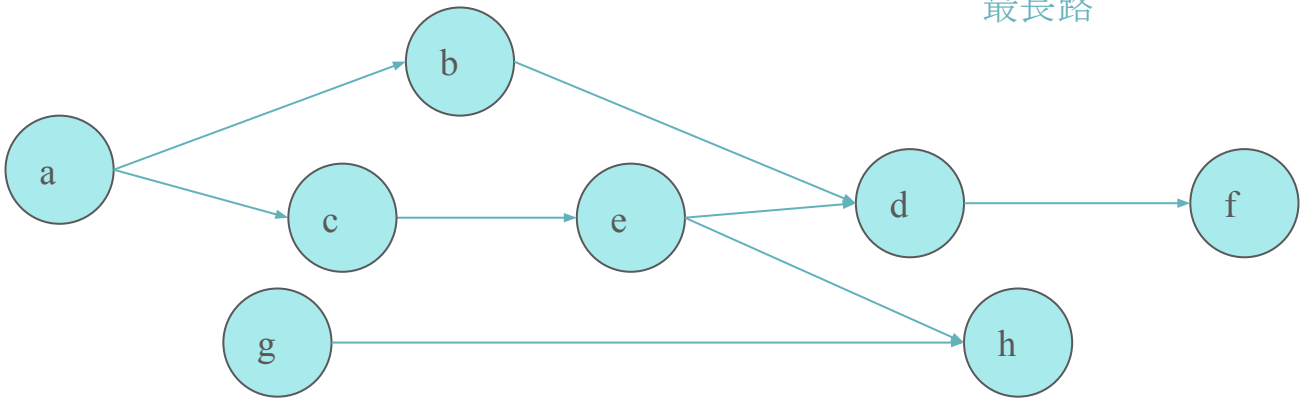
node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	4	0	3



DAG 最長路

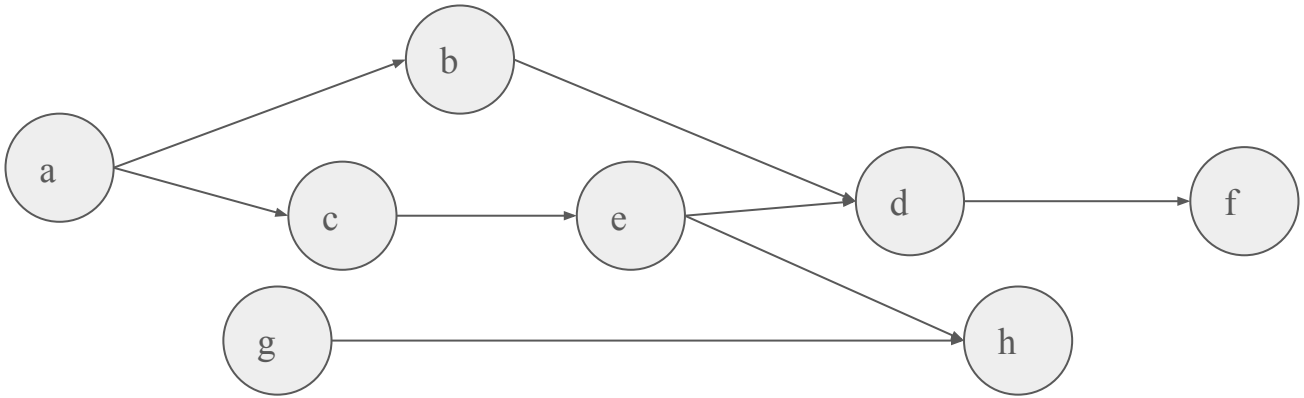
node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	4	0	3

最長路



DAG 最短路

node	a	b	c	d	e	f	g	h
dist	INF	INF	INF	INF	INF	INF	INF	INF



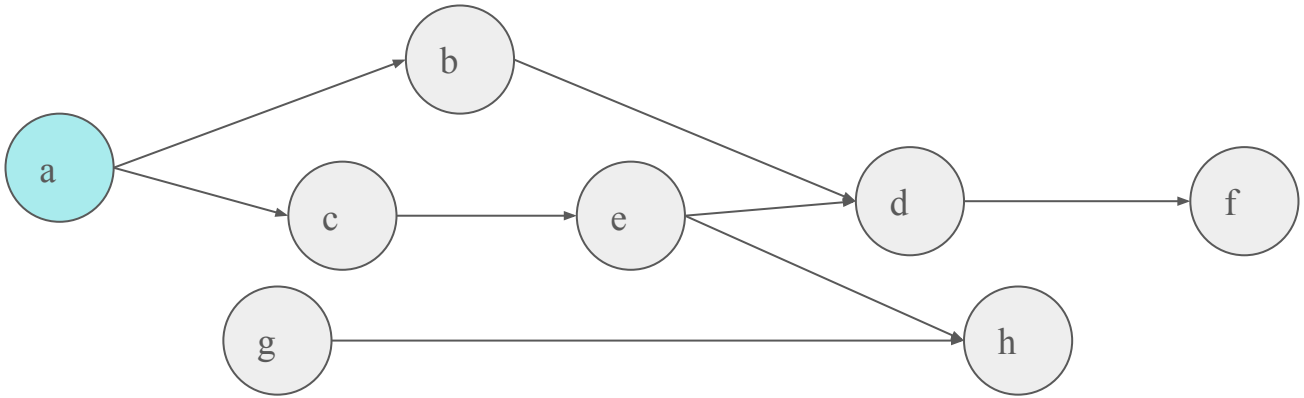
DAG 最短路

選定一個起點 s

dist[s] = 0

dist[a] = 0

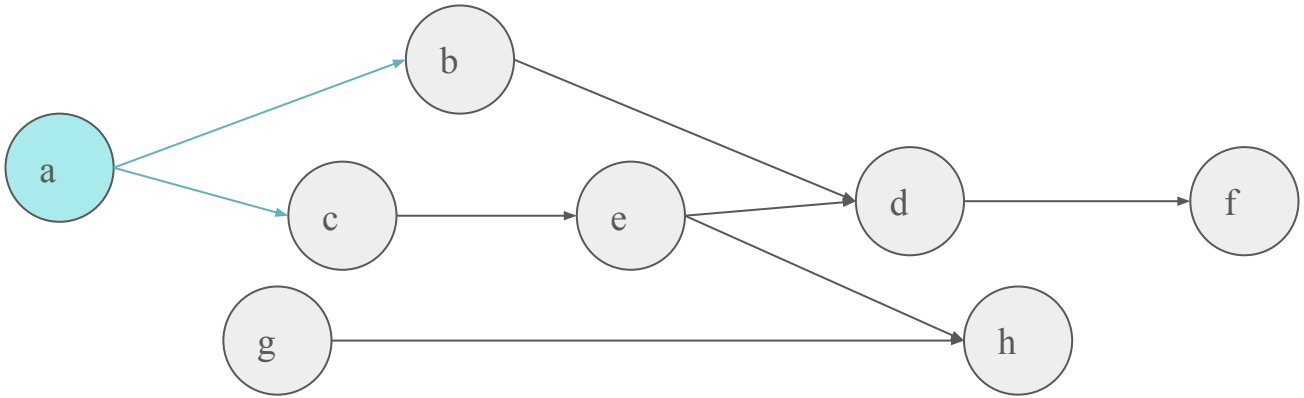
node	a	b	c	d	e	f	g	h
dist	0	INF	INF	INF	INF	INF	INF	INF



DAG 最短路

dist[x] = min(dist[x], dist[parent[x]] + 1)
dist[b] = min(dist[b], dist[a] + 1)
dist[c] = min(dist[c], dist[a] + 1)

node	a	b	c	d	e	f	g	h
dist	0	1	1	INF	INF	INF	INF	INF

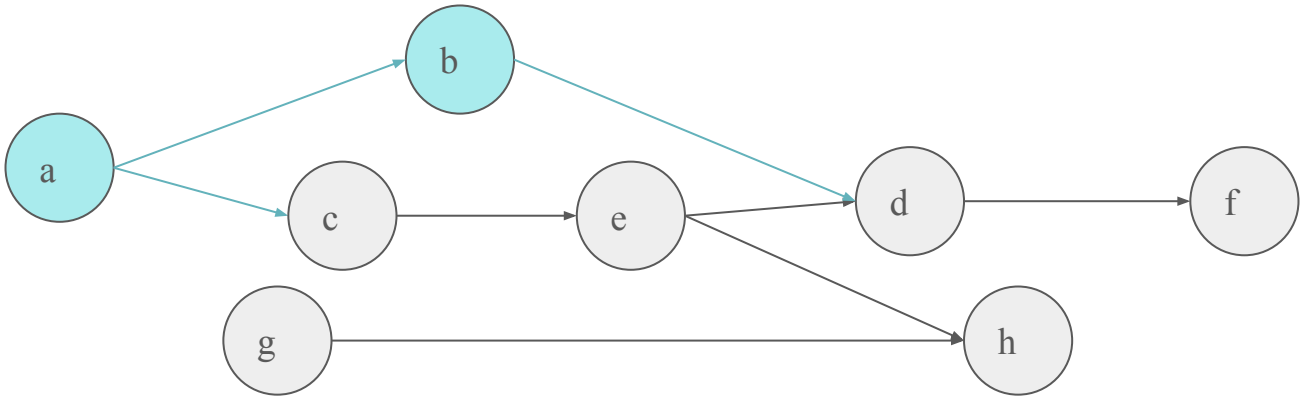


DAG 最短路

dist[x] = min(dist[x], dist[parent[x]] + 1)

dist[d] = min(dist[d], dist[b] + 1)

node	a	b	c	d	e	f	g	h
dist	0	1	1	2	INF	INF	INF	INF

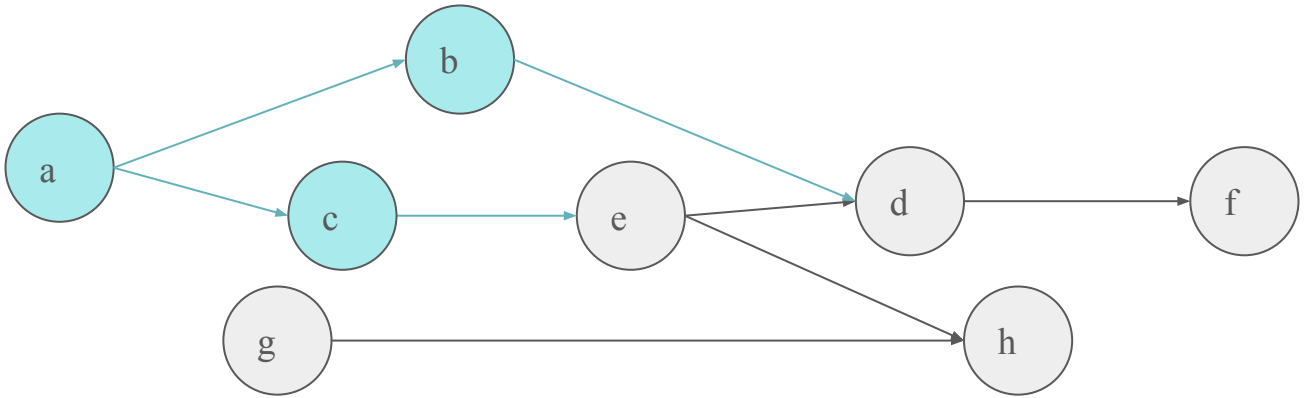


DAG 最短路

dist[x] = min(dist[x], dist[parent[x]] + 1)

dist[e] = min(dist[e], dist[c] + 1)

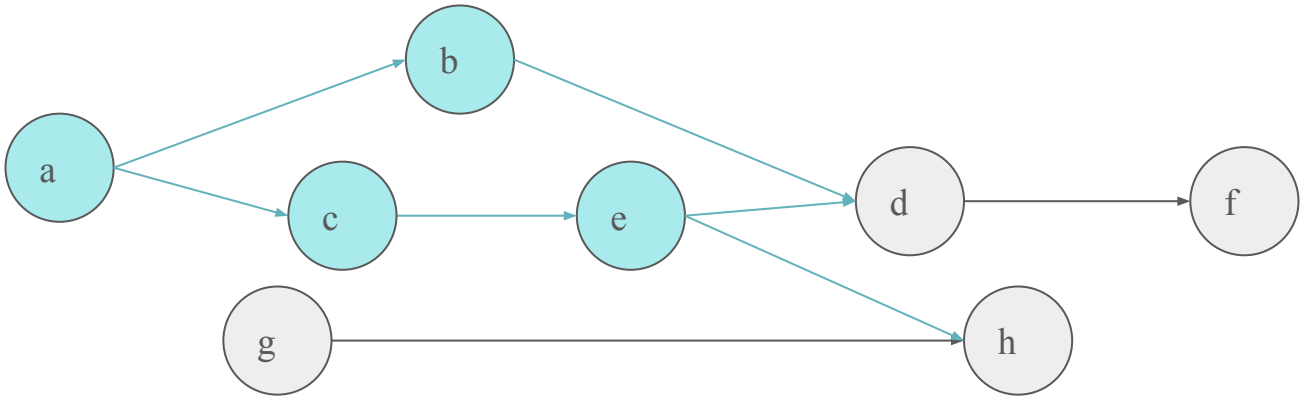
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	INF	INF	INF



DAG 最短路

dist[x] = min(dist[x], dist[parent[x]] + 1)
dist[d] = min(dist[d], dist[e] + 1)
dist[h] = min(dist[h], dist[e] + 1)

node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	INF	INF	3

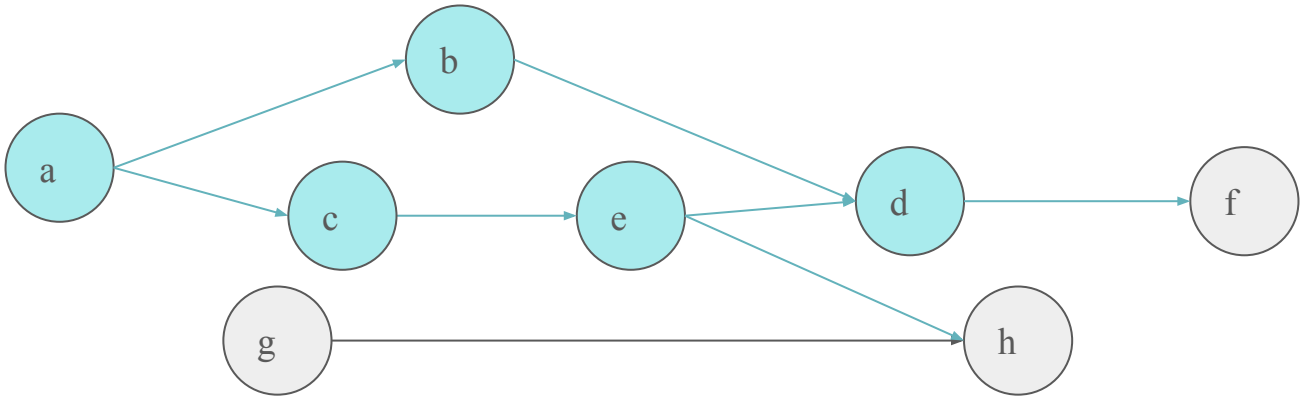


DAG 最短路

dist[x] = min(dist[x], dist[parent[x]] + 1)

dist[f] = min(dist[f], dist[d] + 1)

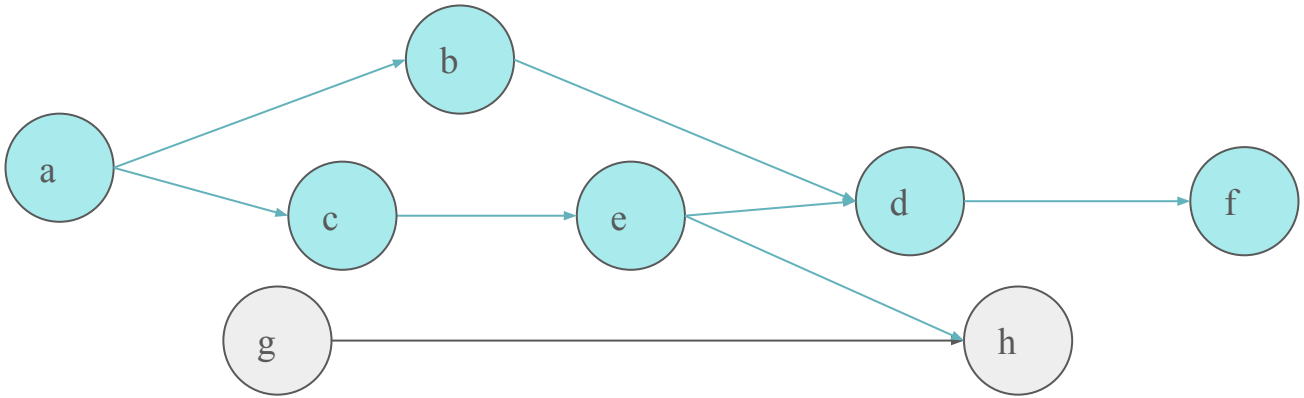
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	3	INF	3



DAG 最短路

dist[x] = min(dist[x], dist[parent[x]] + 1)

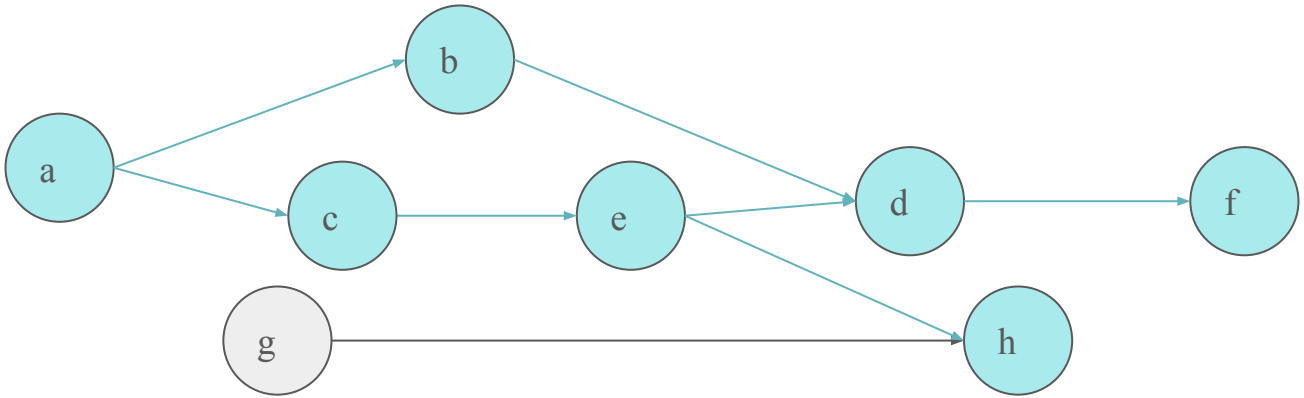
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	3	INF	3



DAG 最短路

dist[x] = min(dist[x], dist[parent[x]] + 1)

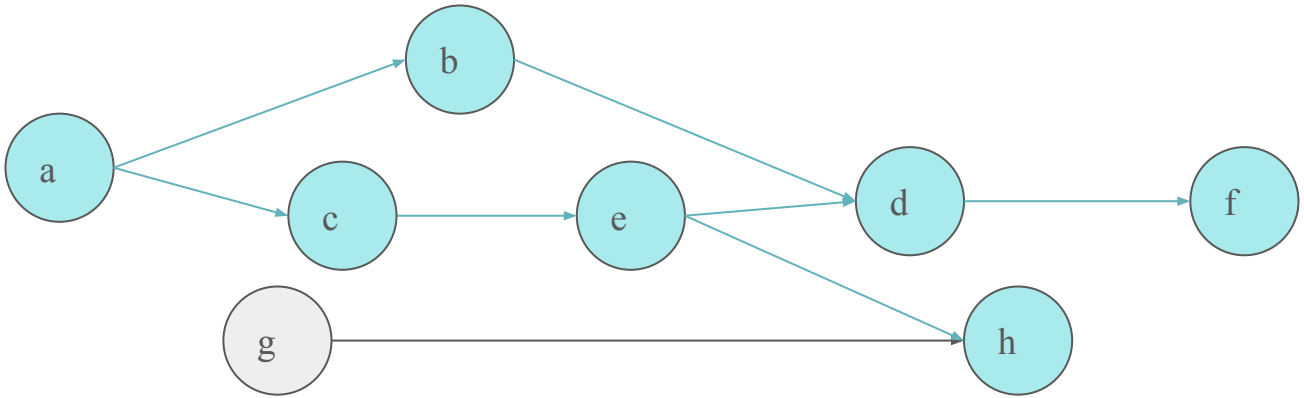
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	3	INF	3



DAG 最短路

dist[x] = min(dist[x], dist[parent[x]] + 1)

node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	3	INF	3



cannot be reached

一般圖正權重的最短路

- 一般圖多出了環, 還可以一次拔拔樂搞定嗎？

一般圖正權重的最短路

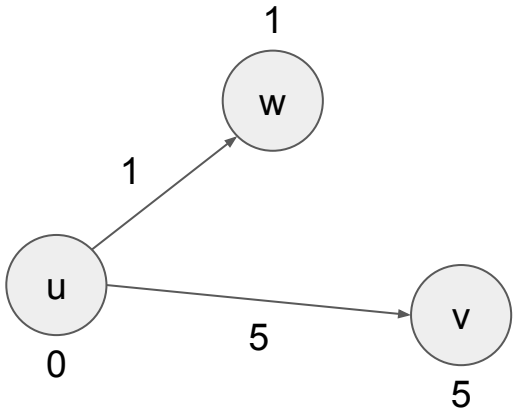
- 一般圖多出了環, 還可以一次拔拔樂搞定嗎？
  - 如果拔到剩環, 就不會有 in degree是0的點可以拔, 所以沒辦法

一般圖正權重的最短路

- 一般圖多出了環, 還可以一次拔拔樂搞定嗎？
  - 如果拔到剩環, 就不會有 in degree是0的點可以拔, 所以沒辦法
- relaxation操作

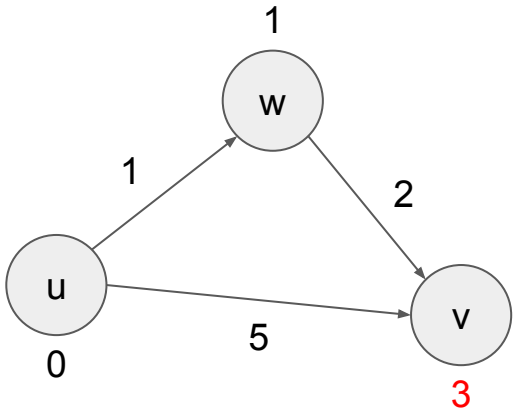
一般圖正權重的最短路

- 一般圖多出了環, 還可以一次拔拔樂搞定嗎？
  - 如果拔到剩環, 就不會有 in degree是0的點可以拔, 所以沒辦法
- relaxation操作
  - 原本已知的最短路為右圖



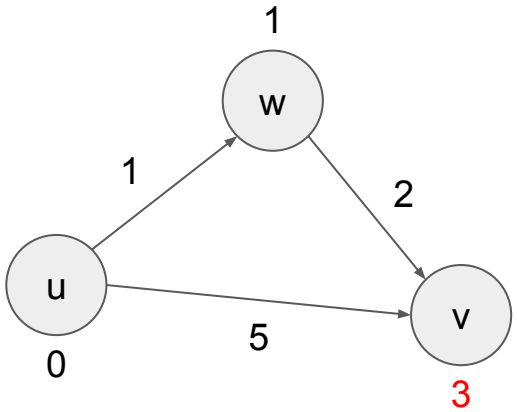
一般圖正權重的最短路

- 一般圖多出了環, 還可以一次拔拔樂搞定嗎？
  - 如果拔到剩環, 就不會有 in degree是0的點可以拔, 所以沒辦法
- relaxation操作
  - 原本已知的最短路為右圖
  - 找到一條更短的路, 讓當前最短路更短了
    - dis[i]為從原點到 i 的當前最短路徑
    - dis[v] = min(dis[v], dis[u] + cost(w, v))



一般圖正權重的最短路

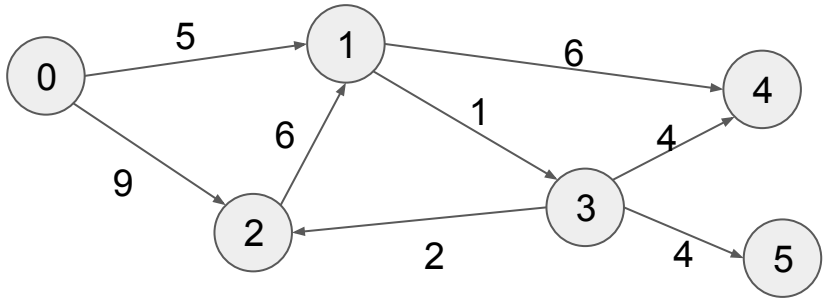
- 一般圖多出了環, 還可以一次拔拔樂搞定嗎？
  - 如果拔到剩環, 就不會有 in degree是0的點可以拔, 所以沒辦法
- relaxation操作
  - 原本已知的最短路為右圖
  - 找到一條更短的路, 讓當前最短路更短了
    - dis[i]為從原點到 i 的當前最短路徑
    - dis[v] = min(dis[v], dis[u] + cost(w, v))
- Djikstra演算法
  - 貪心性質, 如果該節點距離是尚未被選取的點中最小的, 那他就是最短路徑



Dijkstra

初始化dis陣列 (dis[i] := 起點到 i 的最短路徑)。

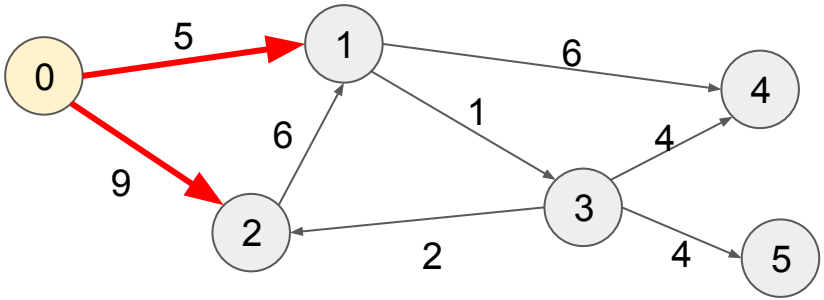
i	0	1	2	3	4	5
dis[i]	0	INF	INF	INF	INF	INF



Dijkstra

選出最小的 dis[i] (節點0), 並對他的鄰居做relaxation操作

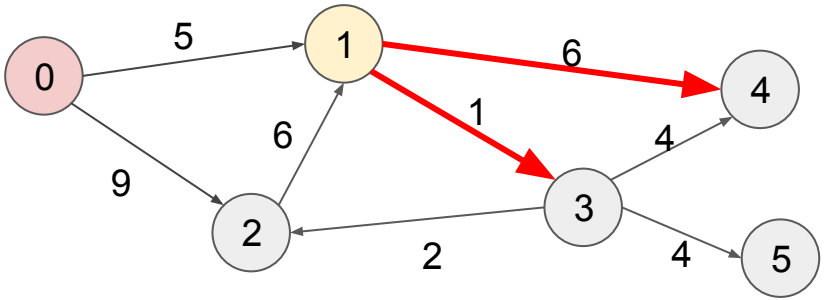
i	0	1	2	3	4	5
dis[i]	0	5	9	INF	INF	INF



Dijkstra

選出最小的 dis[i] (節點1), 並對他的鄰居做relaxation操作

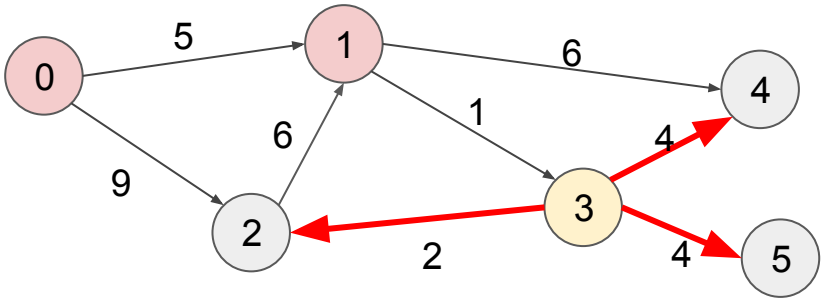
i	0	1	2	3	4	5
dis[i]	0	5	9	6	11	INF



Dijkstra

選出最小的 dis[i] (節點3), 並對他的鄰居做relaxation操作

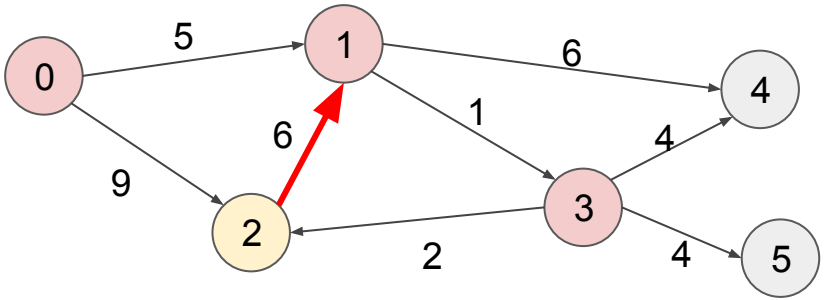
i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Dijkstra

選出最小的 dis[i] (節點2), 並對他的鄰居做relaxation操作

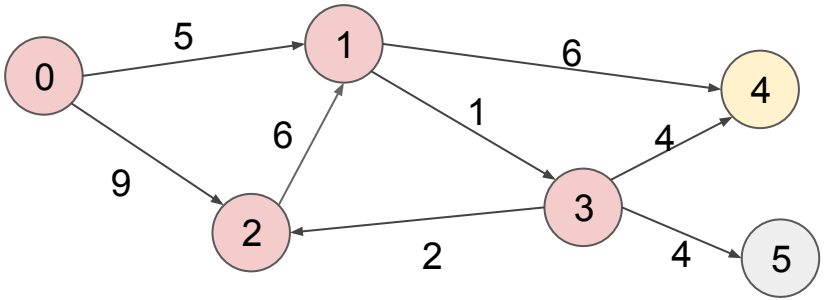
i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Dijkstra

選出最小的 dis[i] (節點4), 沒有鄰居不動作

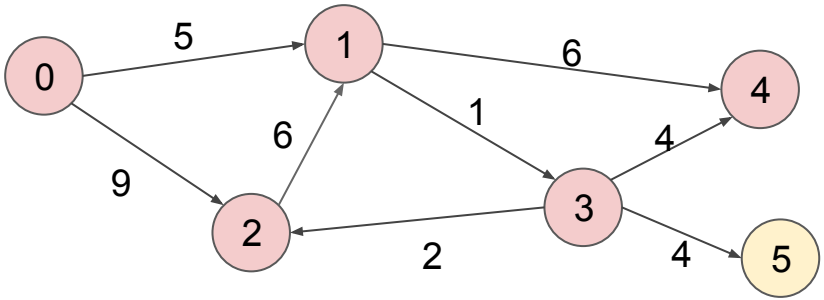
i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Dijkstra

選出最小的 dis[i] (節點5), 沒有鄰居不動作

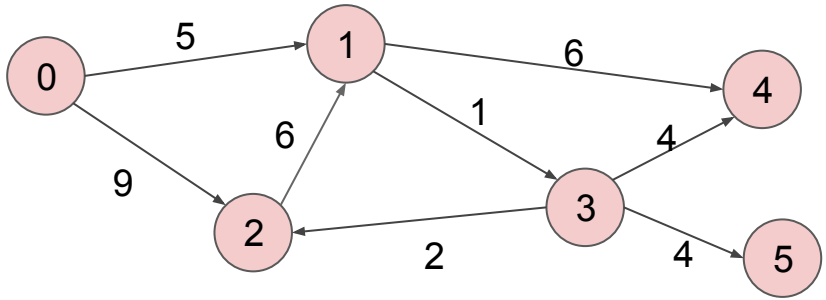
i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Dijkstra

所有點都被選到了，Dijkstra結束

i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Dijkstra 實作

- 每次都挑選當前dis陣列中最小的節點

Dijkstra 實作

- 每次都挑選當前dis陣列中最小的節點
  - 需要一個可以支援插入新東西並排好序的

Dijkstra 實作

- 每次都挑選當前dis陣列中最小的節點
  - 需要一個可以支援插入新東西並排好序的
  - priority_queue

Dijkstra 實作

- 每次都挑選當前dis陣列中最小的節點
  - 需要一個可以支援插入新東西並排好序的
  - priority_queue
- 複雜度分析

Dijkstra 實作

- 每次都挑選當前dis陣列中最小的節點
  - 需要一個可以支援插入新東西並排好序的
  - priority_queue
- 複雜度分析
  - 最壞的情況每個點每個邊都需要被丟進去 priority_queue
    - O((E+V)lgV)

Dijkstra 實作

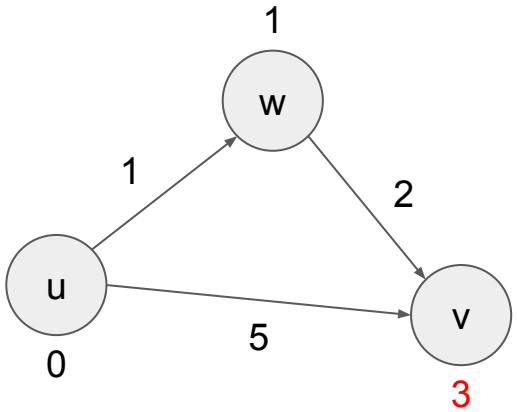
- 每次都挑選當前dis陣列中最小的節點
  - 需要一個可以支援插入新東西並排好序的
  - priority_queue
- 複雜度分析
  - 最壞的情況每個點每個邊都需要被丟進去 priority_queue
    - O((E+V)lgV)
  - 斐波那契堆
    - decrease key: O(1)
    - O(E+VlgV)

Dijkstra Code

```cpp
typedef struct Edge {
    int v; LL w;
    bool operator > (const Edge &b) const {     // 定義Edge的排序標準
        return w > b.w;
    }
} State;
const LL INF = 0x3f3f3f3f3f3f3f3fLL;
void Dijkstra(int n, vector<vector<Edge> > &G, vector<LL> &d, int s, int t = -1) {
    // 建立一個priority_queue
    static priority_queue<State, vector<State>, greater<State> > pq;
    d.clear(); d.resize(n);                 // 初始化d陣列，存放距離
    while (pq.size()) pq.pop();             // 清空pq
    for (auto &num : d) num = INF;
    d[s] = 0; pq.push({s, d[s]});
    while (pq.size()) {                     // 當pq還沒空就繼續做
        auto p = pq.top(); pq.pop();
        int u = p.v;
        if (d[u] < p.w) continue;           // 如果比dis還要大代表是過期的
        if (u == t) return ;
        for (auto &e : G[u]) {              // 列舉所有相鄰的邊
            if (d[e.v] > d[u] + e.w) {      // relaxation操作
                d[e.v] = d[u] + e.w;
                pq.push({e.v, d[e.v]}); // 將更新的State丟進去priority_queue裡面
            }
        }
    }
}
```

一般圖正權重的最短路

- Bellman-Ford演算法
  - 每一回合都讓每條邊都 relaxation一次
  - 做 n - 1 回合就完成單源點最短路

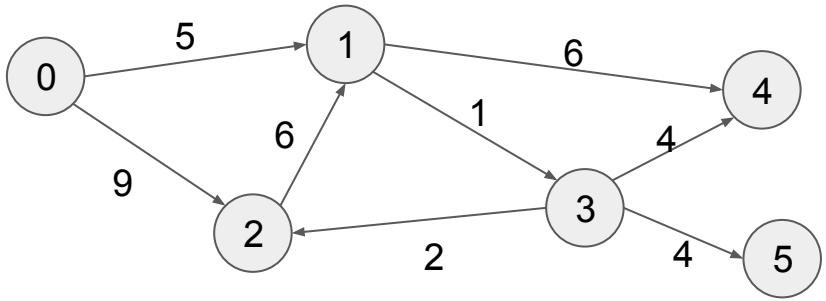


一般圖正權重的最短路

用 Edge List 的方式存起來(Adjacency List也可以, 作法大同小異), 初始化dis陣列

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	INF	INF	INF	INF	INF

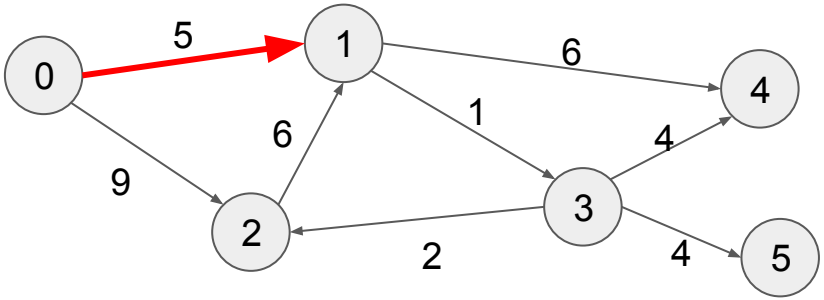


一般圖正權重的最短路

對第一條邊做relaxation操作

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	INF	INF	INF	INF

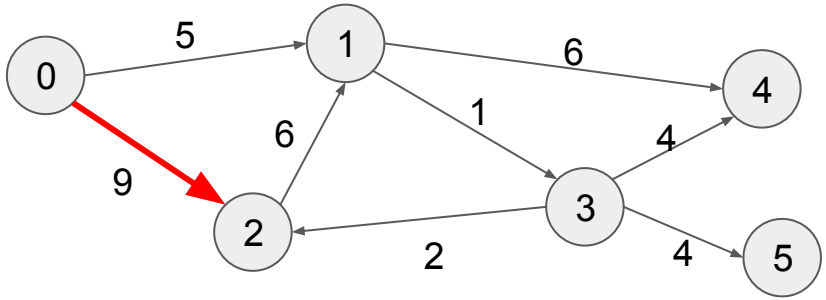


一般圖正權重的最短路

對第二條邊做relaxation操作

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	9	INF	INF	INF

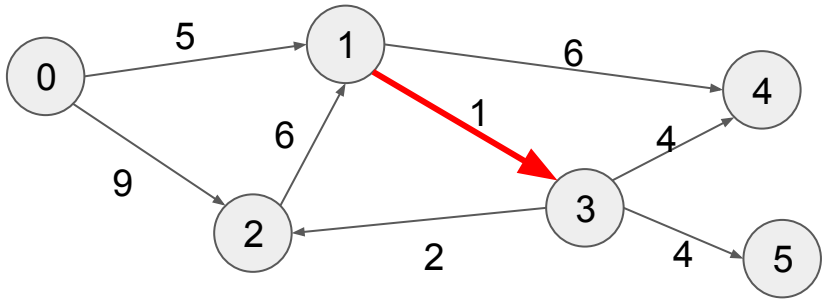


一般圖正權重的最短路

對第三條邊做relaxation操作

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	9	6	INF	INF

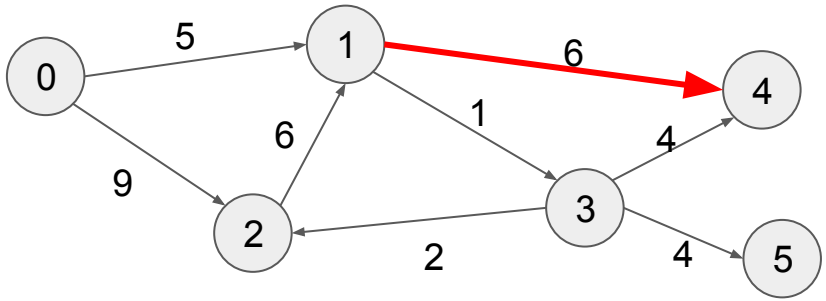


一般圖正權重的最短路

對第四條邊做relaxation操作

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	9	6	11	INF

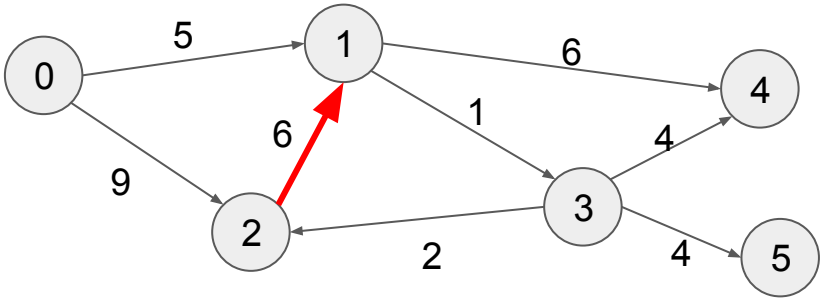


一般圖正權重的最短路

對第五條邊做relaxation操作

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	9	6	11	INF

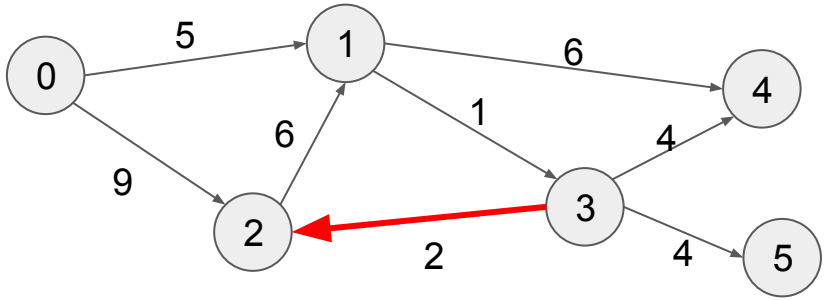


一般圖正權重的最短路

對第六條邊做relaxation操作

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	8	6	11	INF

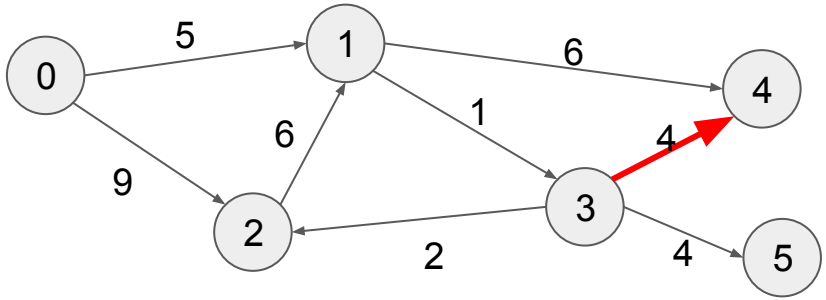


一般圖正權重的最短路

對第七條邊做relaxation操作

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	INF

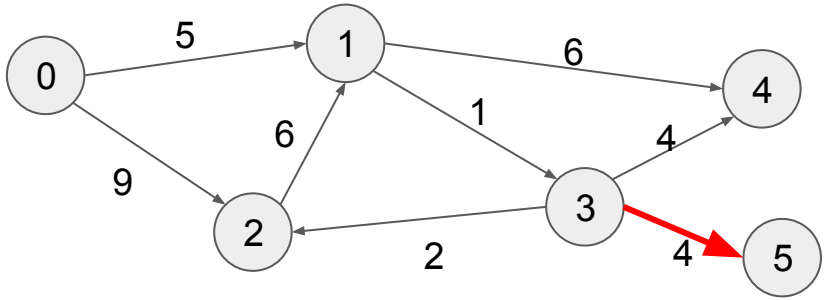


一般圖正權重的最短路

對第八條邊做relaxation操作, 完成第一個回合

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10

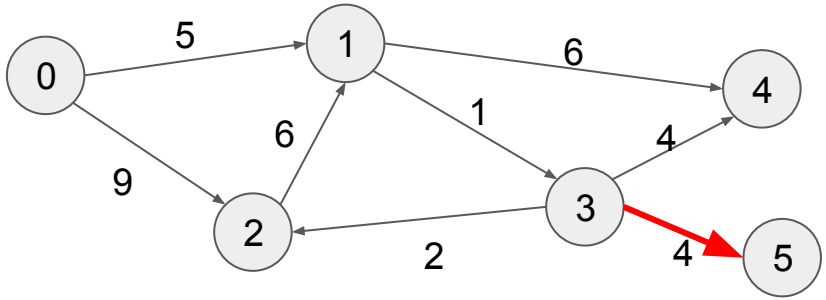


一般圖正權重的最短路

對第八條邊做relaxation操作，完成第n - 1個回合，結束

u	v	w
0	1	5
0	2	9
1	3	1
1	4	6
2	1	6
3	2	2
3	4	4
3	5	4

i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Bellman Ford

```
 5 const int INF = 0x3f3f3f3f;
 6 int n;                    // 共有幾個節點
 7 int dis[MAXN];            // dis陣列
 8 vector<Edge> edgeList     // 邊列表
 9 void relaxation(Edge &e) {                // relaxation操作
10     dis[e.v] = min(dis[e.v], dis[e.u] + e.w);
11 }
12 void BellmanFord(int s) {
13     memset(dis, INF, sizeof(dis));
14     dis[s] = 0;
15     for (int i = 0 ; i < n - 1 ; i++)    // 做 n - 1 次
16         for (auto &e : edgeList)         // 每次跑過整個Edge List
17             relaxation(e);               // 對每個 Edge 做 relaxation 操作
18 }
```

Bellman Ford

- 為什麼最多要做 n - 1 次

Bellman Ford

- 為什麼最多要做 n - 1 次
  - n 個點的圖中的路徑最多經過 n 個點 (每個點都走到)

Bellman Ford

- 為什麼最多要做 n - 1 次
  - n 個點的圖中的路徑最多經過 n 個點 (每個點都走到)
  - 每一次relaxation最多讓當前最短路徑多增加一個點

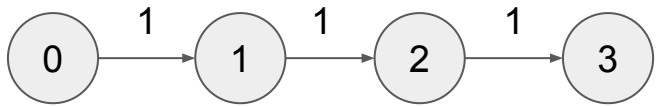
Bellman Ford

- 為什麼最多要做 n - 1 次
  - n 個點的圖中的路徑最多經過 n 個點 (每個點都走到)
  - 每一次relaxation最多讓當前最短路徑多增加一個點
  - 因此每個點最多需要做 n - 1 次的relaxation才會形成 n 個點的路徑

Bellman Ford

對這張圖做Bellman Ford

u	v	w
2	3	1
1	2	1
0	1	1

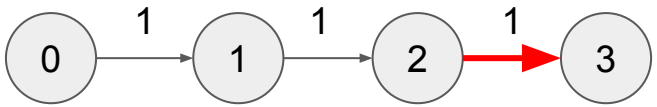


i	0	1	2	3
dis[i]	0	INF	INF	INF

Bellman Ford

對第一條邊做relaxation

u	v	w
2	3	1
1	2	1
0	1	1

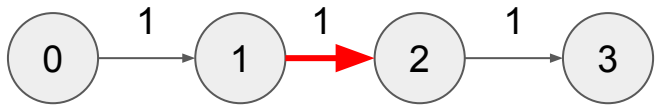


i	0	1	2	3
dis[i]	0	INF	INF	INF

Bellman Ford

對第二條邊做relaxation

u	v	w
2	3	1
1	2	1
0	1	1

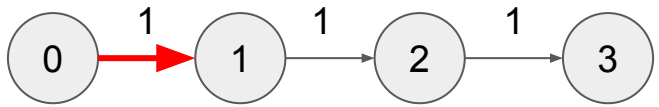


i	0	1	2	3
dis[i]	0	INF	INF	INF

Bellman Ford

對第三條邊做relaxation, 完成第一次迭代

u	v	w
2	3	1
1	2	1
0	1	1

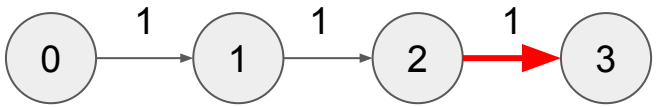


i	0	1	2	3
dis[i]	0	1	INF	INF

Bellman Ford

對第一條邊做relaxation

u	v	w
2	3	1
1	2	1
0	1	1

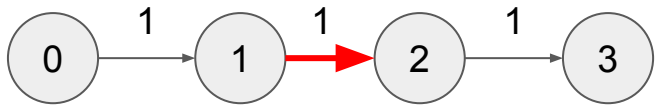


i	0	1	2	3
dis[i]	0	1	INF	INF

Bellman Ford

對第二條邊做relaxation

u	v	w
2	3	1
1	2	1
0	1	1

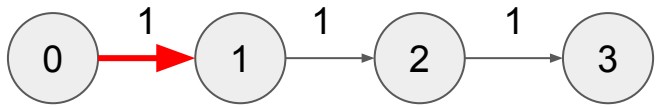


i	0	1	2	3
dis[i]	0	1	2	INF

Bellman Ford

對第三條邊做relaxation, 完成第二次迭代

u	v	w
2	3	1
1	2	1
0	1	1

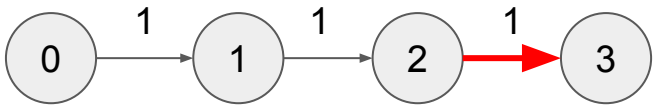


i	0	1	2	3
dis[i]	0	1	2	INF

Bellman Ford

對第一條邊做relaxation

u	v	w
2	3	1
1	2	1
0	1	1

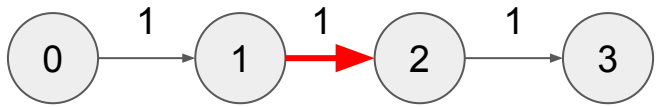


i	0	1	2	3
dis[i]	0	1	2	3

Bellman Ford

對第二條邊做relaxation

u	v	w
2	3	1
1	2	1
0	1	1

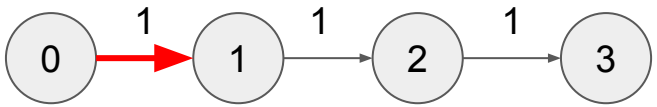


i	0	1	2	3
dis[i]	0	1	2	3

Bellman Ford

對第三條邊做relaxation, 完成第三次迭代

u	v	w
2	3	1
1	2	1
0	1	1

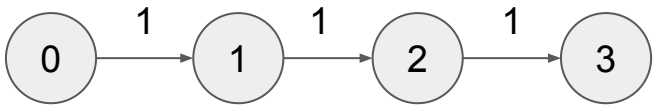


i	0	1	2	3
dis[i]	0	1	2	3

Bellman Ford

完成Bellman Ford, 最多需要做3 = (4 - 1) 次

u	v	w
2	3	1
1	2	1
0	1	1



i	0	1	2	3
dis[i]	0	1	2	3

Bellman Ford

- 複雑度分析

Bellman Ford

- 複雜度分析
  - 一次relaxation的cost為 O(1)

Bellman Ford

- 複雜度分析
  - 一次relaxation的cost為 O(1)
  - 每一回合都會做O(E)次 relaxation

Bellman Ford

- 複雜度分析
  - 一次relaxation的cost為 O(1)
  - 每一回合都會做O(E)次 relaxation
  - 總共要做O(V - 1)回合

Bellman Ford

- 複雜度分析
  - 一次relaxation的cost為 O(1)
  - 每一回合都會做O(E)次 relaxation
  - 總共要做O(V - 1)回合
- 總共是 O((V - 1) * E * 1) = O(VE)

一般圖負權邊的最短路

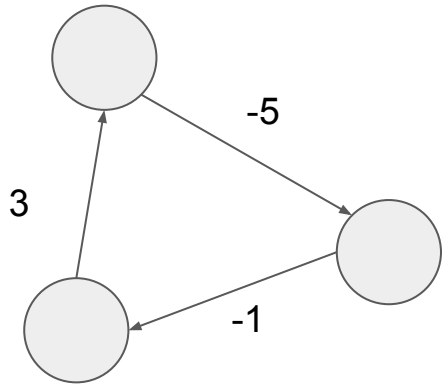
- 權重出現負權重, BellmanFord 和 Dijkstra還可以運作嗎？

一般圖負權邊的最短路

- 權重出現負權重, BellmanFord 和 Dijkstra還可以運作嗎？
  - Dijkstra 不行 (可以想想看為什麼 Hint:貪心性質還會不會成立)
  - BellmanFord 不一定

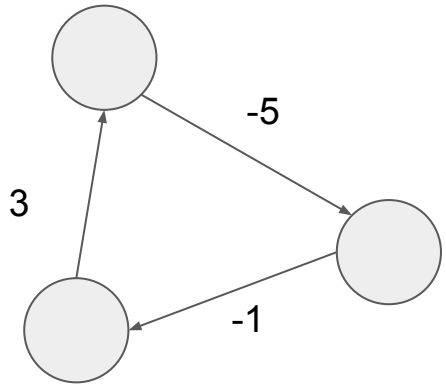
一般圖負權邊的最短路

- 權重出現負權重, BellmanFord 和 Dijkstra還可以運作嗎？
  - Dijkstra 不行 (可以想想看為什麼 Hint:貪心性質還會不會成立)
  - BellmanFord 不一定
- 負環
  - 找到一個環, 他的總和是負數



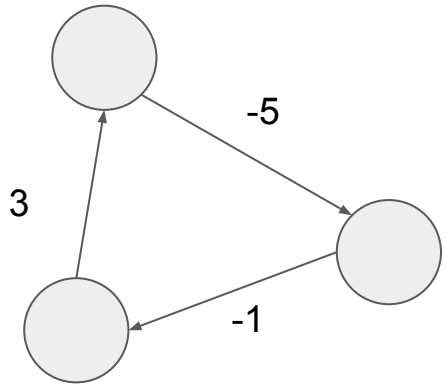
一般圖負權邊的最短路

- 權重出現負權重, BellmanFord 和 Dijkstra還可以運作嗎？
  - Dijkstra 不行 (可以想想看為什麼 Hint:貪心性質還會不會成立)
  - BellmanFord 不一定
- 負環
  - 找到一個環, 他的總和是負數
  - 在有負環的圖中, 多繞幾圈他會形成更短的路徑
    - 所以不存在最短路徑



一般圖負權邊的最短路

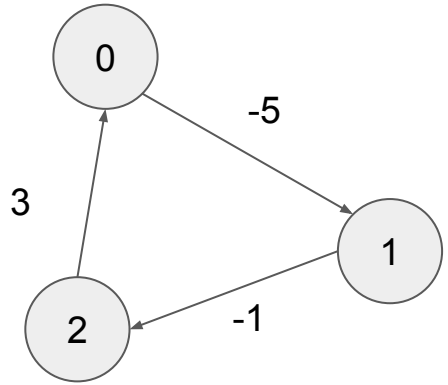
- 權重出現負權重, BellmanFord 和 Dijkstra還可以運作嗎？
  - Dijkstra 不行 (可以想想看為什麼 Hint:貪心性質還會不會成立)
  - BellmanFord 不一定
- 負環
  - 找到一個環, 他的總和是負數
  - 在有負環的圖中, 多繞幾圈他會形成更短的路徑
    - 所以不存在最短路徑
- 若圖沒有出現可以到達的負環, BellmanFord依然可以照常運作。



Bellman Ford遇到負環

u	v	w
0	1	-5
1	2	-1
2	0	3

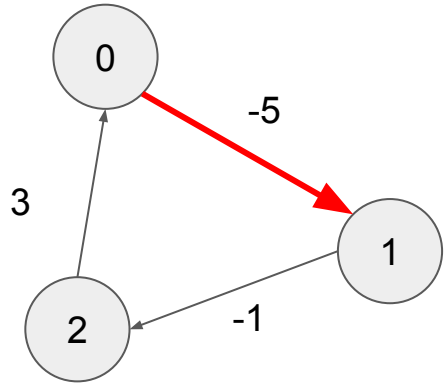
i	0	1	2
dis[i]	0	INF	INF



Bellman Ford遇到負環

u	v	w
0	1	-5
1	2	-1
2	0	3

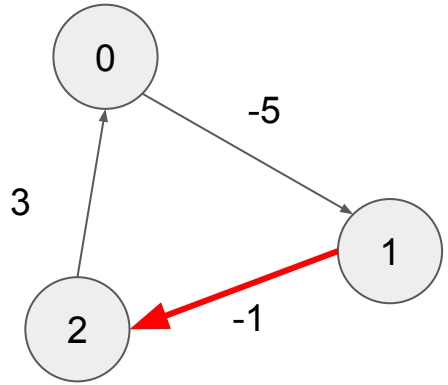
i	0	1	2
dis[i]	0	-5	INF



Bellman Ford遇到負環

u	v	w
0	1	-5
1	2	-1
2	0	3

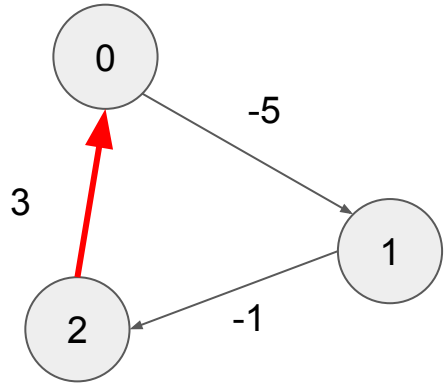
i	0	1	2
dis[i]	0	-5	-6



Bellman Ford遇到負環

u	v	w
0	1	-5
1	2	-1
2	0	3

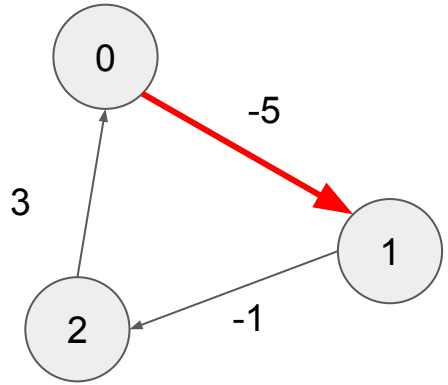
i	0	1	2
dis[i]	-3	-5	-6



Bellman Ford遇到負環

u	v	w
0	1	-5
1	2	-1
2	0	3

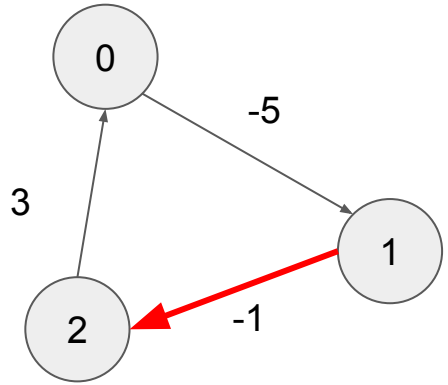
i	0	1	2
dis[i]	-3	-8	-6



Bellman Ford遇到負環

u	v	w
0	1	-5
1	2	-1
2	0	3

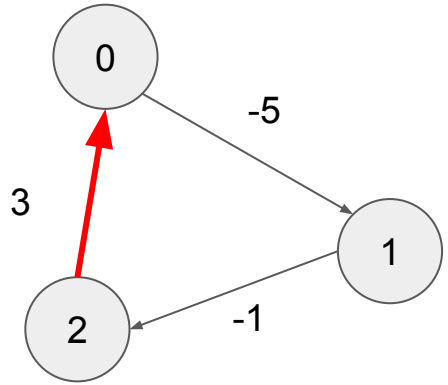
i	0	1	2
dis[i]	-3	-8	-9



Bellman Ford遇到負環

u	v	w
0	1	-5
1	2	-1
2	0	3

i	0	1	2
dis[i]	-6	-8	-9

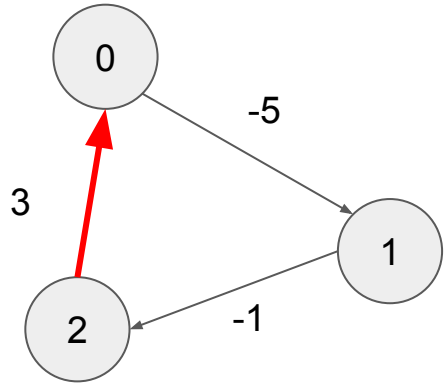


Bellman Ford遇到負環

沒辦法收斂

u	v	w
0	1	-5
1	2	-1
2	0	3

i	0	1	2
dis[i]	-6	-8	-9



Bellman Ford偵測負環

- Bellman Ford 在沒有負環的圖做 n - 1 次的迭代就會收斂

Bellman Ford偵測負環

- Bellman Ford 在沒有負環的圖做 n - 1 次的迭代就會收斂
  - 做 n - 1 次迭代還沒有收斂, 表示有負環

Bellman Ford偵測負環

- Bellman Ford 在沒有負環的圖做 n - 1 次的迭代就會收斂
  - 做 n - 1 次迭代還沒有收斂, 表示有負環
- 紀錄每個點被更新幾次, 更新超過 n - 1 次代表這張圖有負環

Bellman Ford偵測負環 Code

```
11 const int INF = 0x3f3f3f3f;
12 int n;
13 int dis[MAXN];
14 vector<Edge> edgeList;
15 int updateTimes[MAXN];                    // 該點更新的次數
16 bool relaxation(Edge &e) {                // 回傳是否有被relaxation
17     if (dis[e.v] > dis[e.u] + e.w) {
18         dis[e.v] = dis[e.u] + e.w;
19         return true;
20     }
21     return false;
22 }
```

Bellman Ford偵測負環 Code

```
23 bool negative_cycle_base_on_BellmanFord() {
24     memset(updateTimes, 0, sizeof(updateTimes));          // 初始化更新次數
25     memset(dis, INF, sizeof(dis));                        // 初始化dis陣列
26     dis[0] = 0;                                           // 起點dis設定為0
27     while (1) {
28         bool change = false;                              // 紀錄有沒有改變dis值
29         for (auto &e : edgeList) {                        // 跑過整個edgeList
30             if (relaxation(e)) {                          // 如果有被relaxation
31                 change = true;                            // 還沒有收斂
32                 updateTimes[e.v]++;                       // 更新該點的更新次數
33             }
34             if (updateTimes[e.v] >= n) return true;       // 如果超過n - 1次代表有負環
35         }
36         if (!change) return false;                        // 跑過一次發現沒變就代表沒有負環
37     }
38 }
```

Bellman Ford優化(SPFA)

- 讓我們回想一下Dijkstra

Bellman Ford優化(SPFA)

- 讓我們回想一下Dijkstra
- 只要更新有可能距離變短的點就可以了

Bellman Ford優化(SPFA)

- 讓我們回想一下Dijkstra
- 只要更新有可能距離變短的點就可以了
  - 用一個queue存有哪些點可能變短
  - 一旦有relaxation讓距離變短, 就把他放進去 queue裡面

Bellman Ford優化(SPFA)

- 讓我們回想一下Dijkstra
- 只要更新有可能距離變短的點就可以了
  - 用一個queue存有哪些點可能變短
  - 一旦有relaxation讓距離變短，就把他放進去 queue裡面
  - 概念類似Dijkstra，只是把priority_queue改成一般的queue

Bellman Ford優化(SPFA) Code

```cpp
const int MAXN = 1e3 + 5;
const int INF = 0x3f3f3f3f;
vector<Edge> G[MAXN];
int dis[MAXN];
bool relaxation(Edge e) {
    if (dis[e.v] < dis[e.u] + e.w) {
        dis[e.v] = dis[e.u] + e.w;
        return true;
    }
    return false;
}
void SPFA(int s) {
    queue<int> q; q.push(s);
    memset(dis, INF, sizeof(dis));
    dis[s] = 0;
    while (q.size()) {                  // queue還沒空就繼續做
        int u = q.front(); q.pop();     // 拿出最前面的節點
        for (auto &e : G[u]) {          // 列舉每個相鄰的邊
            if (relaxation(e))          // relaxation操作
                q.push(e.v);            // 如果他有被relaxation就丟進去queue裡面
                                        // 代表有可能會有更短的路徑
        }
    }
}
```

SPFA優化

- queue重複點優化
  - 要丟進queue之前先判斷該點是否已經在 queue裡面了
  - 重複的點在queue裡面並沒有幫助, 只會增加要消化的 queue長度

SPFA優化 重複點優化 Code

```
 5 const int MAXN = 1e3 + 5;
 6 const int INF = 0x3f3f3f3f;
 7 vector<Edge> G[MAXN];
 8 int dis[MAXN];
 9 bool relaxation(Edge e) {
10     if (dis[e.v] < dis[e.u] + e.w) {
11         dis[e.v] = dis[e.u] + e.w;
12         return true;
13     }
14     return false;
15 }
16 bool inq[MAXN];                    // inq[i] := 節點i是否在queue裡面了
17 void SPFA(int s) {
18     queue<int> q; q.push(s);
19     memset(dis, INF, sizeof(dis));
20     memset(inq, false, sizeof(inq));
21     dis[s] = 0; inq[s] = true;      // 維護好inq陣列
22     while (q.size()) {
23         int u = q.front(); q.pop();
24         inq[u] = false;
25         for (auto &e : G[u]) {
26             if (relaxation(e)) {
27                 inq[e.v] = true;    // 加進queue記得把inq設定為True
28                 q.push(e.v);
29             }
30         }
31     }
32 }
```

SPFA優化

- 把queue改成deque
  - deque有pop_front()和pop_back()的操作
  - 如果dis[back()]比較dis[front()]小，就pop_back()，否則就pop_front();

SPFA優化 Code

```
 5 const int MAXN = 1e3 + 5;
 6 const int INF = 0x3f3f3f3f;
 7 vector<Edge> G[MAXN];
 8 int dis[MAXN];
 9 bool relaxation(Edge e) {
10     if (dis[e.v] < dis[e.u] + e.w) {
11         dis[e.v] = dis[e.u] + e.w;
12         return true;
13     }
14     return false;
15 }
16 bool inq[MAXN];
17 void SPFA(int s) {
18     deque<int> q; q.push_back(s);
19     memset(dis, INF, sizeof(dis));
20     memset(inq, false, sizeof(inq));
21     dis[s] = 0; inq[s] = true;
22     while (q.size()) {
23         int u;
24         if (dis[q.front()] < dis[q.back()]) {    // front()比較小
25             u = q.front();
26             q.pop_front();
27         } else {                                 // back()比較小
28             u = q.back();
29             q.pop_back();
30         }
31         inq[u] = false;
32         for (auto &e : G[u]) {
33             if (relaxation(e)) {
34                 inq[e.v] = true;
35                 q.push_back(e.v);
36             }
37         }
38     }
39 }
```

延伸題材

- 其他圖論問題
  - 割點和橋
  - 樹上LCA
  - 各種連通分量
    - 點雙連通、邊雙連通、強連通分量
  - 配對問題
  - 最大流與最小割
  - 最大團與最大獨立集
  -
  - ……