Graph

圖論

講師

楊秉宇

交大資工系大二

detaomega

NYCU_LoTaTea

大綱

圖的介紹

圖的存法

圖的搜索

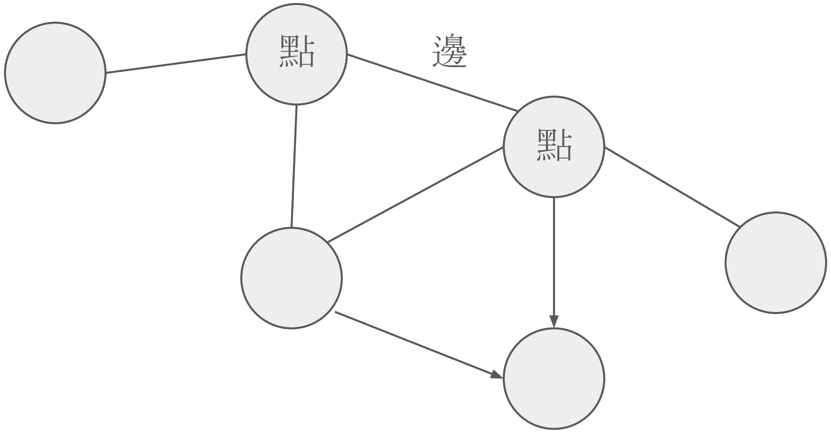
樹

有向無環圖(DAG)

一般圖最短路

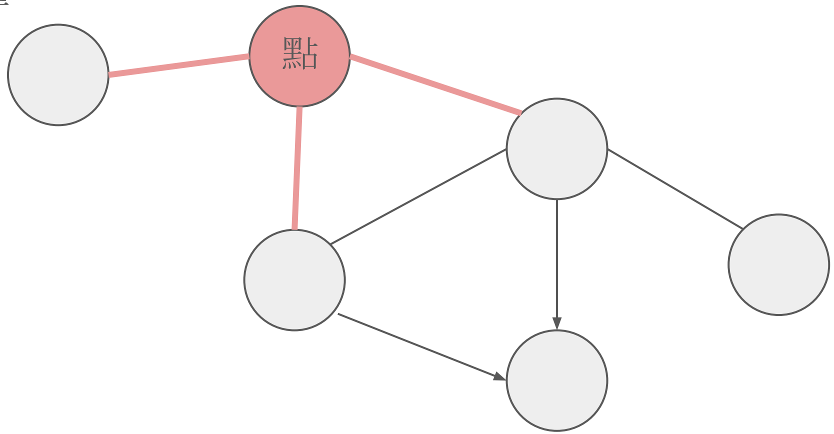
圖的介紹

- 由點(圓圈) 跟 邊(線條) 所成的結構



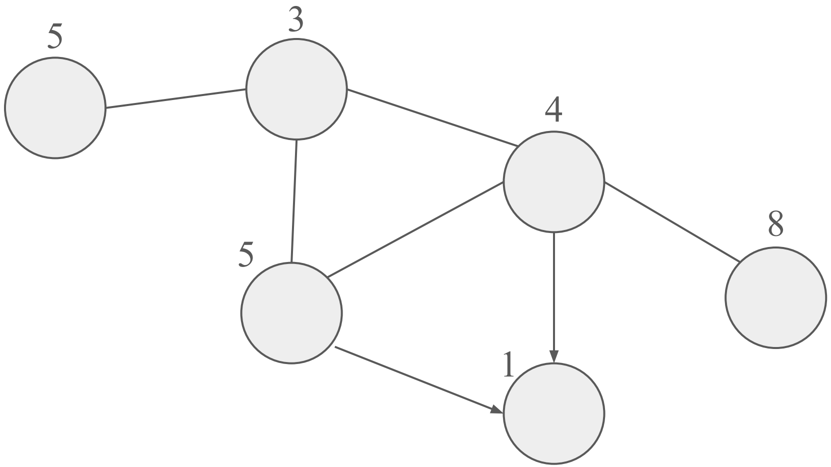
點 Node

- 一個點可以連出去很多邊
- 度數(degree)
  - 入度(x):有多少點指向點x的數量
  - 出度(x):點x向外連接點的數量



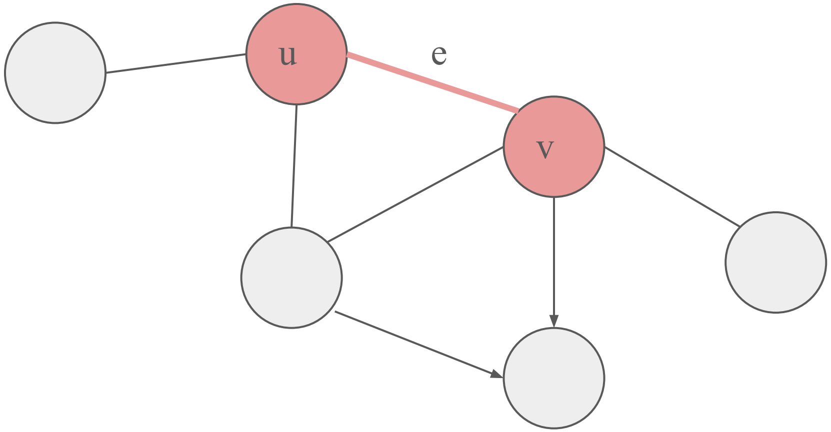
點 Node(Vertex)

- 點上有數字, 稱為點權重



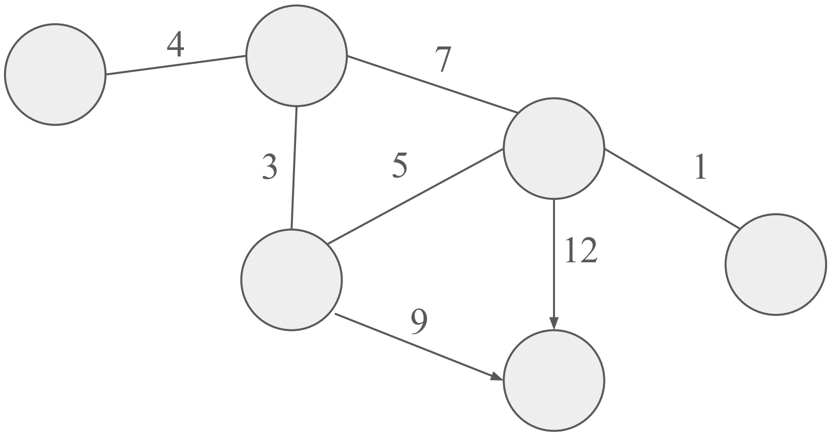
邊 Edge

- 一條邊只能連到兩個點(相同或相異)



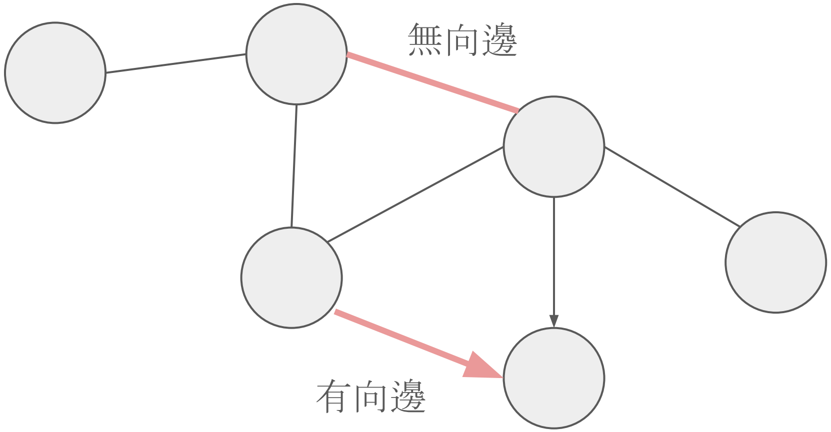
邊

- 邊上有數字, 稱為邊權重



邊

- 無方向的邊叫無向邊
- 有方向的邊叫有向邊

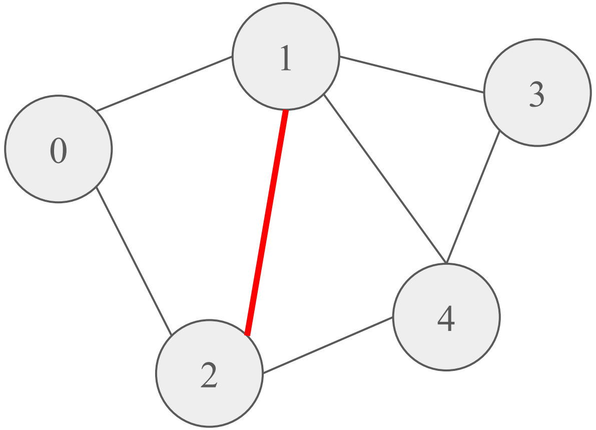


圖的存法

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

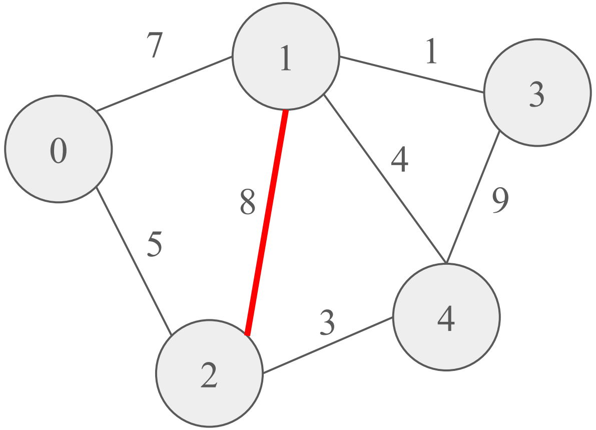
鄰接矩陣 Adjacency Matrix

	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



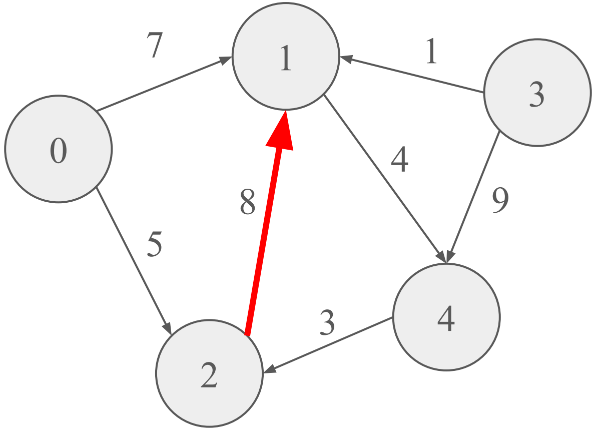
鄰接矩陣 存無向邊帶權重

	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

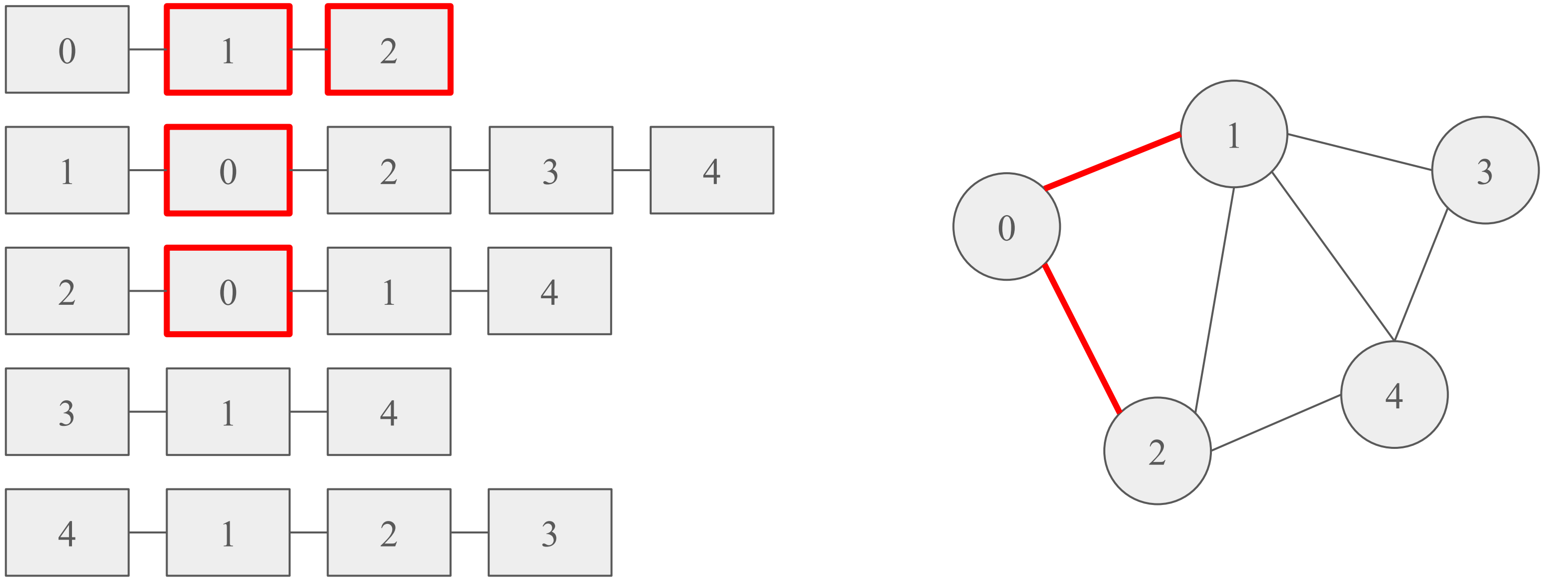
	0	1	2	3	4
0	0	7	5	0	0
1	0	0	0	0	4
2	0	8	0	0	0
3	0	1	0	0	9
4	0	0	3	0	0



鄰接矩陣 存有向邊帶權重 Code

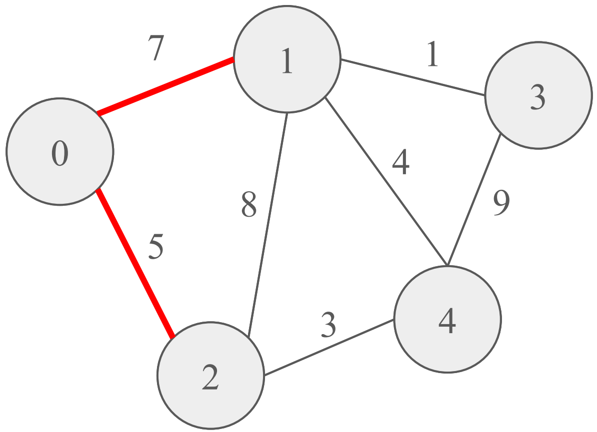
```
const int Vertex = 1000 //假設點有1000個
int Graph[Vertex][Vertex];
void AddEdge(int u, int v, int w) {
    Graph[u][v] = w;
    Graph[v][u] = w; // 無向邊記得加
}
```

鄰接串列 Adjacency List



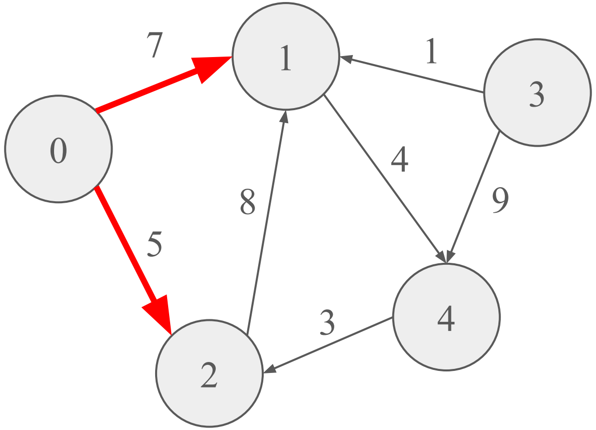
鄰接串列 存無向邊帶權重

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



鄰接串列 存有向邊帶權重

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

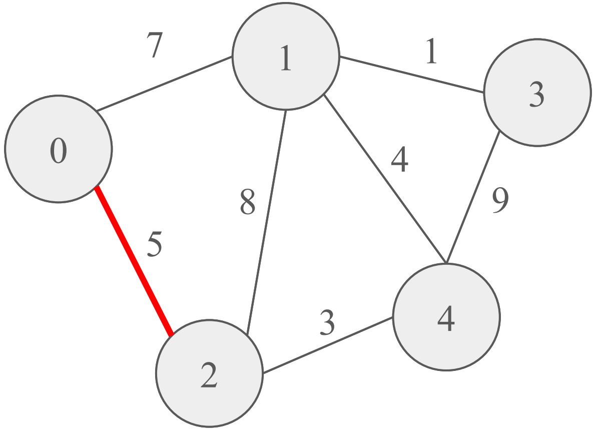


鄰接串列 存有向邊帶權重 Code

```
const int Vertex = 1000 //假設點有1000個
struct Edge {
	int v, w;
};
vector<Edge> Graph[Vertex];
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

```
struct Edge {
	int u, v, w;
};
vector<Edge> EdgeList;
void addEdge (int u, int v, int w) {
	EdgeList.push_back({u, v, w});
}
```

圖的存法比較

- 鄰接串列 Adjacency list
  - 空間複雜度 O(V + E)
- 鄰接矩陣 Adjacency matrix
  - 空間複雜度 O(V^2)
- 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

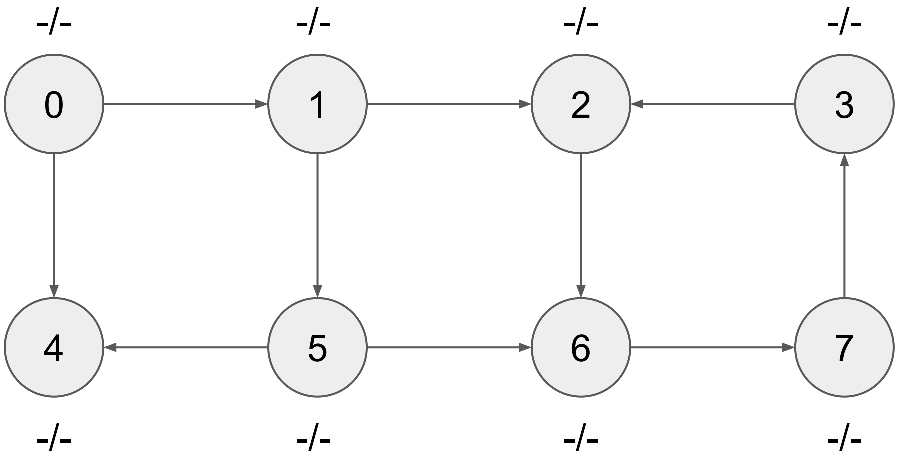
- 深度優先搜索(DFS)
- 廣度優先搜索(BFS)

深度優先搜索 DFS

- 又名 **Depth First Search**
- 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

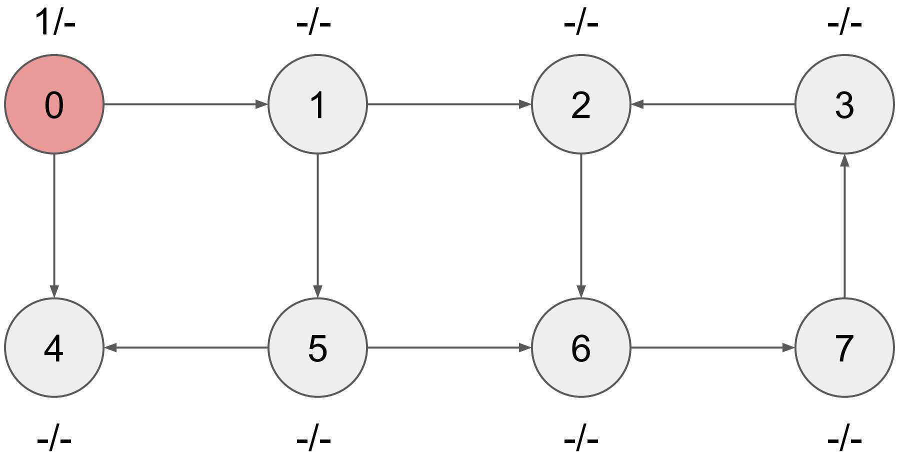
深度優先搜索 DFS

對這張圖做DFS



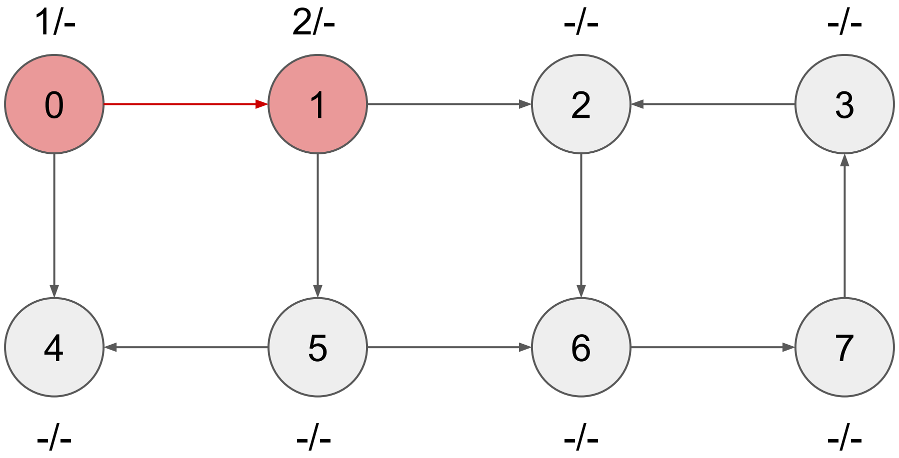
深度優先搜索 DFS

起點為節點0



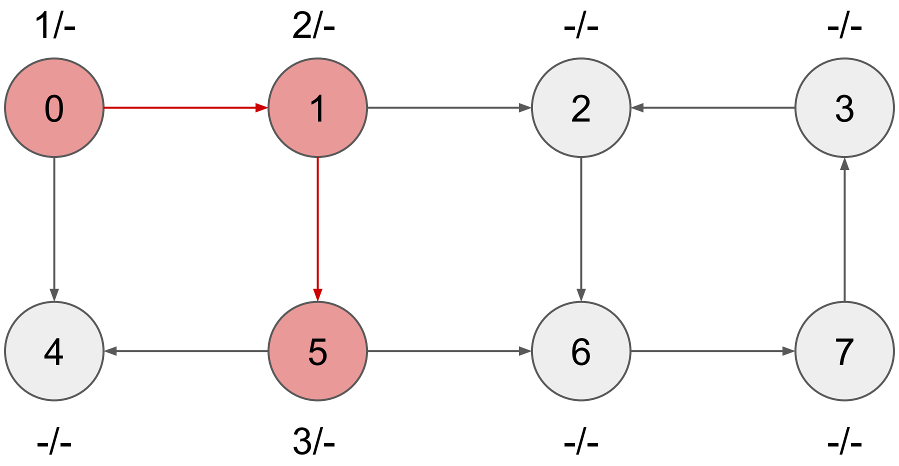
深度優先搜索 DFS

選出節點0連出去的邊中還未走訪的點 (節點1)



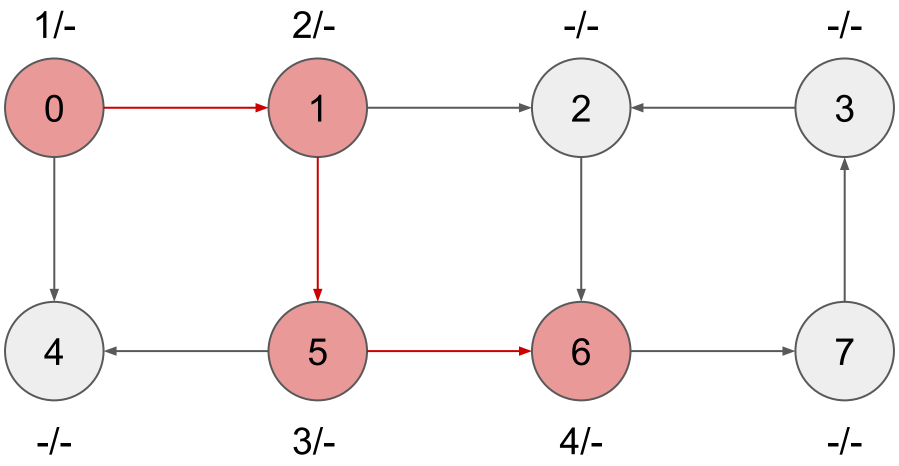
深度優先搜索 DFS

選出節點1連出去的邊中還未走訪的點 (節點5)



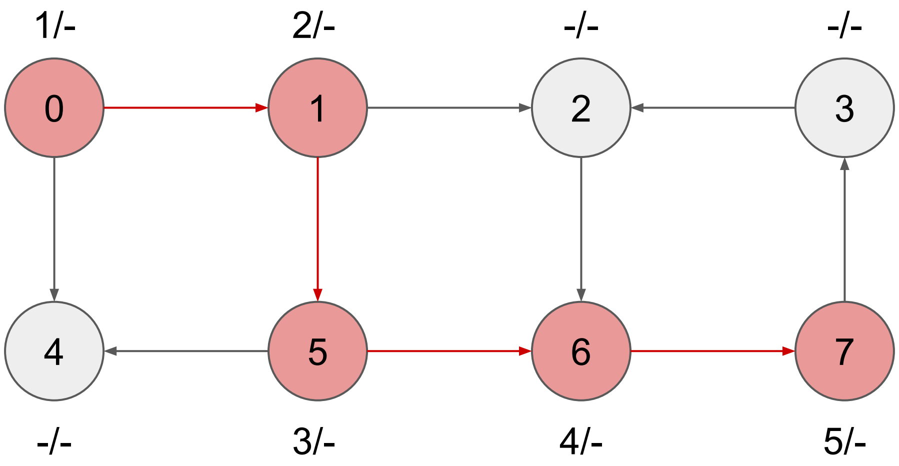
深度優先搜索 DFS

選出節點5連出去的邊中還未走訪的點 (節點6)



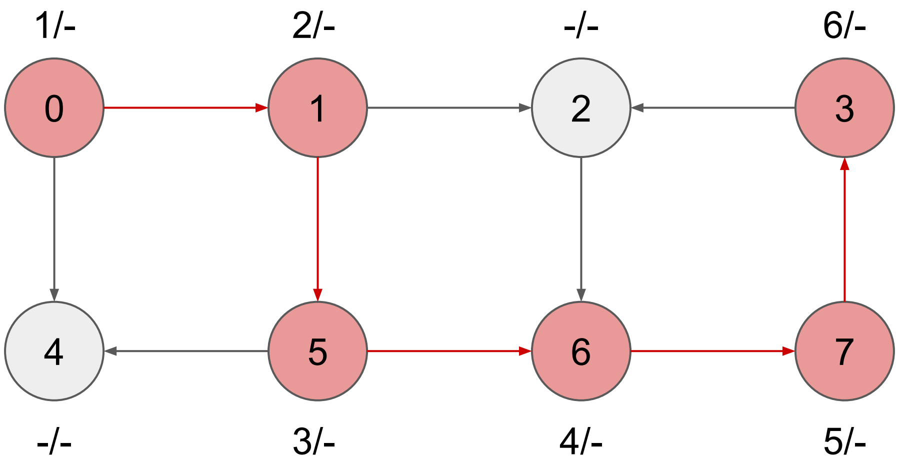
深度優先搜索 DFS

選出節點6連出去的邊中還未走訪的點 (節點7)



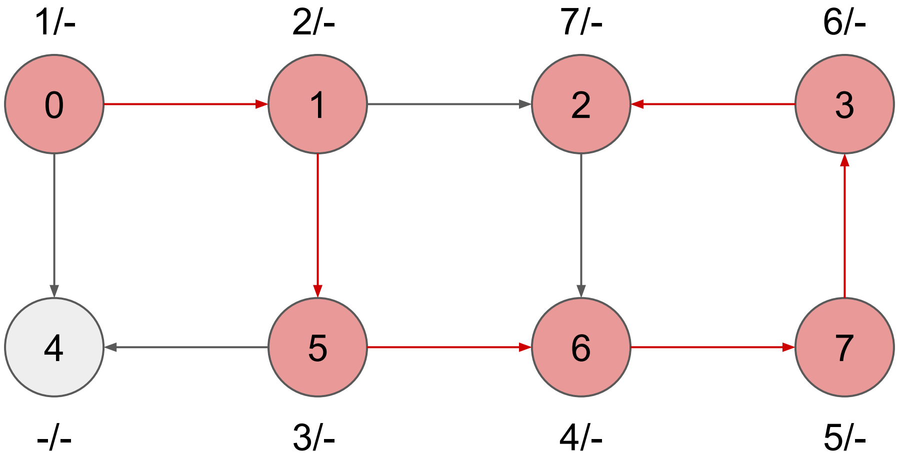
深度優先搜索 DFS

選出節點7連出去的邊中還未走訪的點 (節點3)



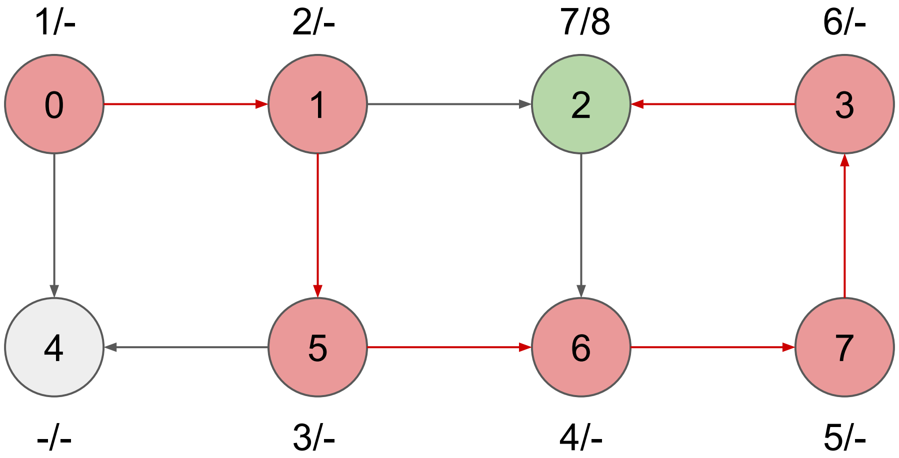
深度優先搜索 DFS

選出節點3連出去的邊中還未走訪的點 (節點2)



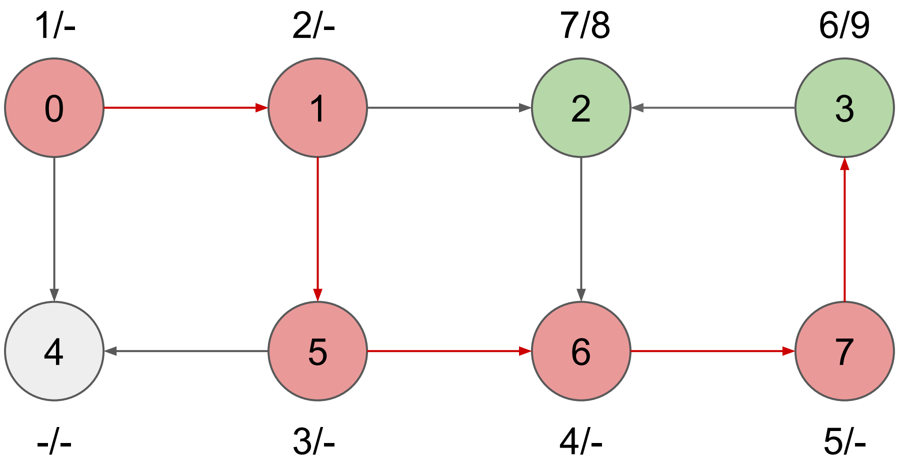
深度優先搜索 DFS

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。



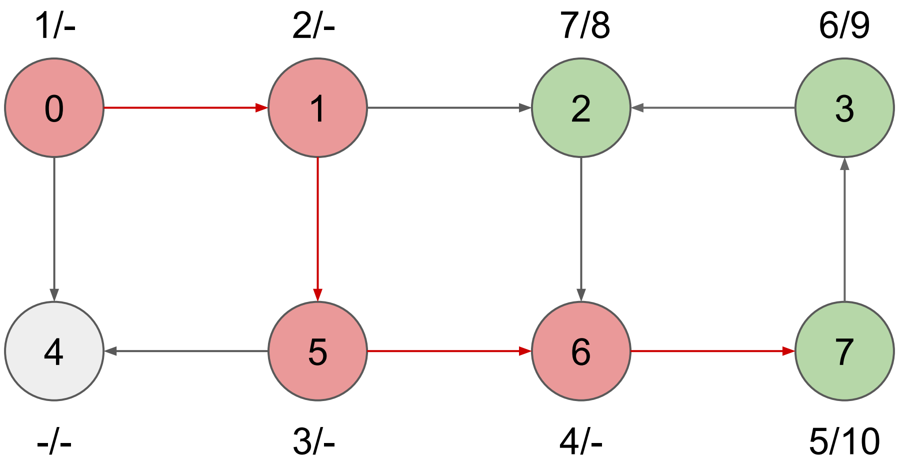
深度優先搜索 DFS

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。



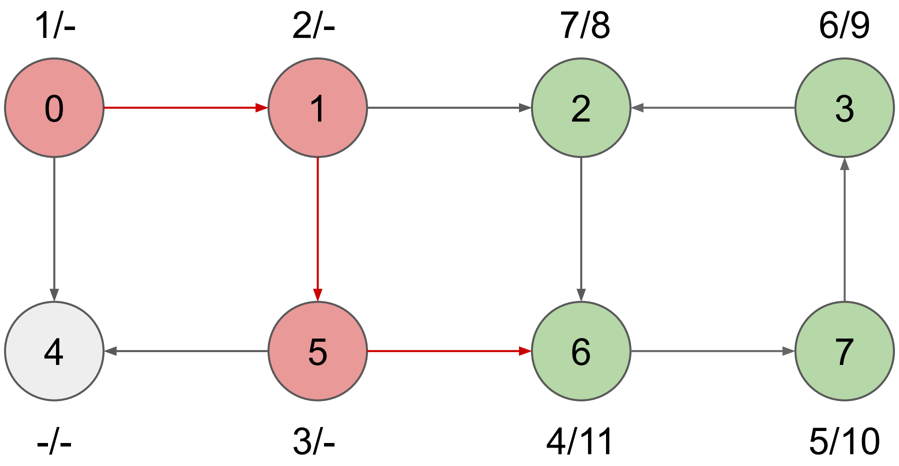
深度優先搜索 DFS

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。



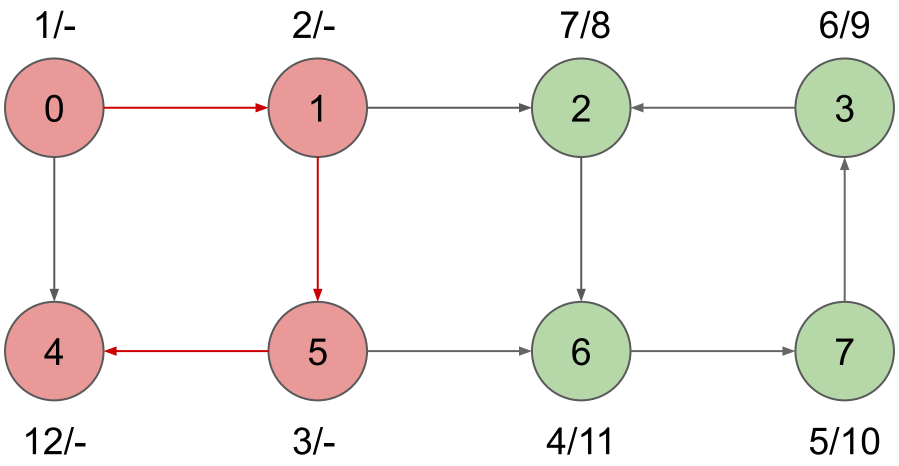
深度優先搜索 DFS

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。



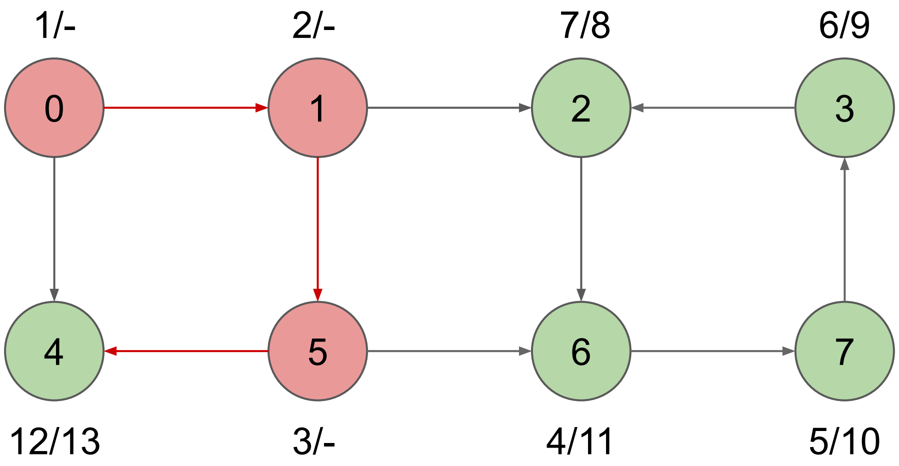
深度優先搜索 DFS

選出節點5連出去的邊中還未走訪的點 (節點4)



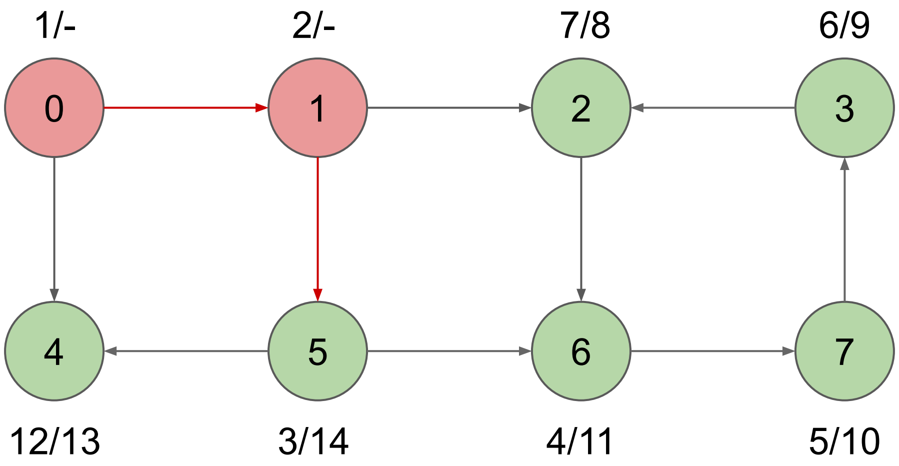
深度優先搜索 DFS

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。



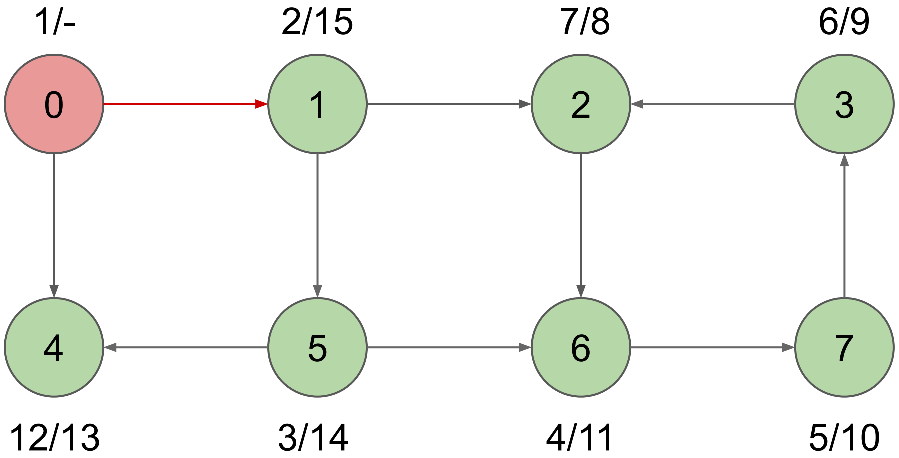
深度優先搜索 DFS

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。



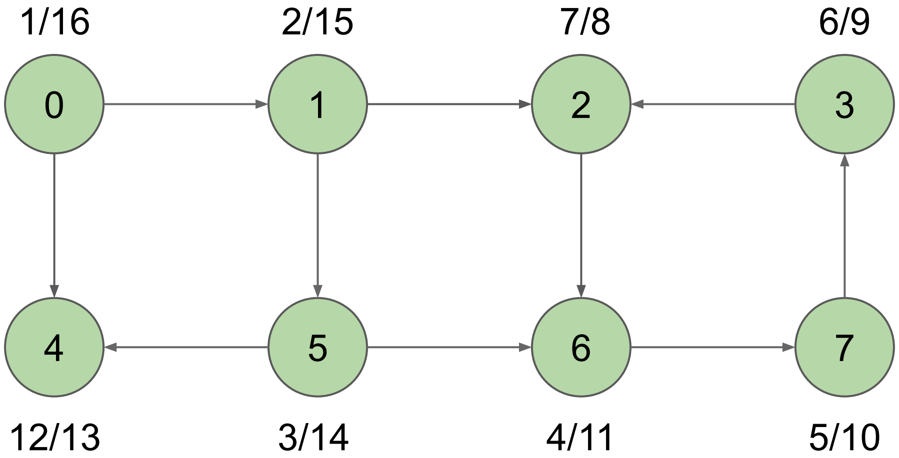
深度優先搜索 DFS

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。



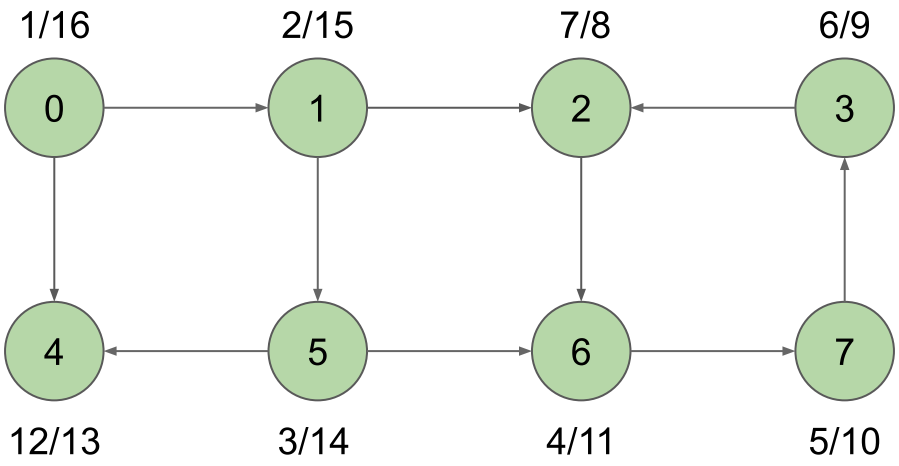
深度優先搜索 DFS

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

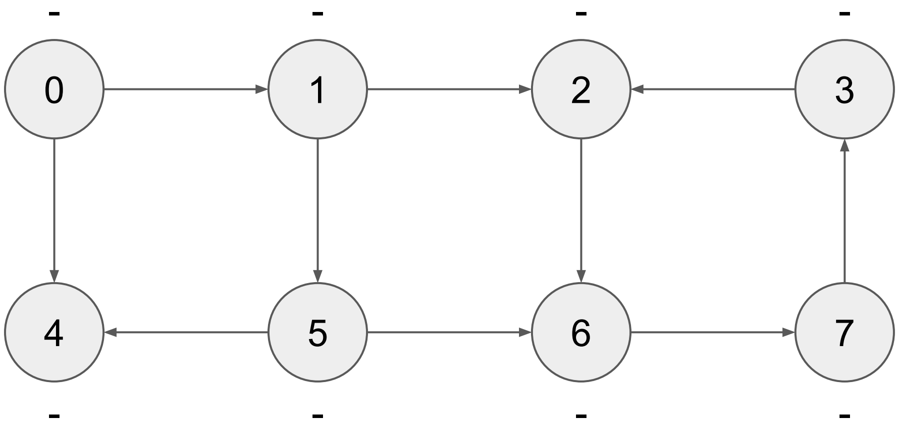
```cpp
const int Vertex = 1000;
int inStamp[Vertex], outStamp[Vertex], stamp;
vector<int> Graph[Vertex];
void init() {
    memset(inStamp, 0, sizeof(inStamp));
    memset(outStamp, 0, sizeof(outStamp));
    stamp = 1;
}
void dfs(int u) {
    inStamp[u] = stamp++;     //紀錄inStamp
    for (int v : Graph[u]) { //列舉所有相鄰的點
        if (inStamp[v] == 0) {//如果inStamp 是0代表還沒有拜訪過
            dfs(v);           //拜訪完這個點
        }
    }
    outStamp[u] = stamp++; // 拜訪玩紀錄outStamp
}
```

廣度優先搜索 BFS

- 又名 **Breadth First Search**
- 概念和淹水相同
    - 從原點擴散出去
- 重要的資訊
    - level值(第幾層)
    - 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

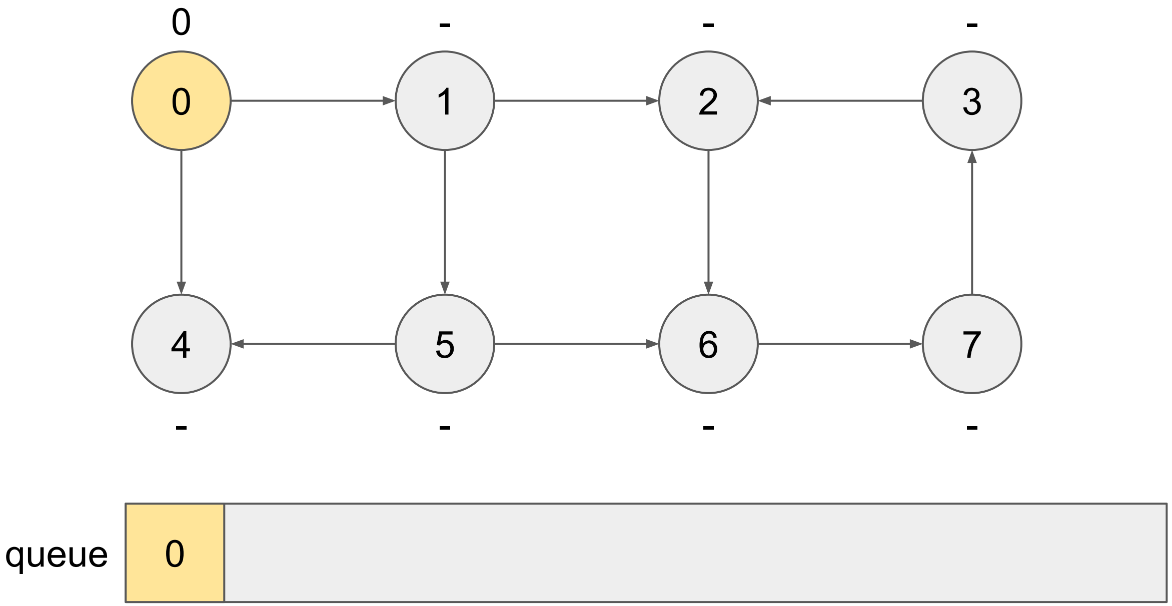
對這張圖做BFS



queue

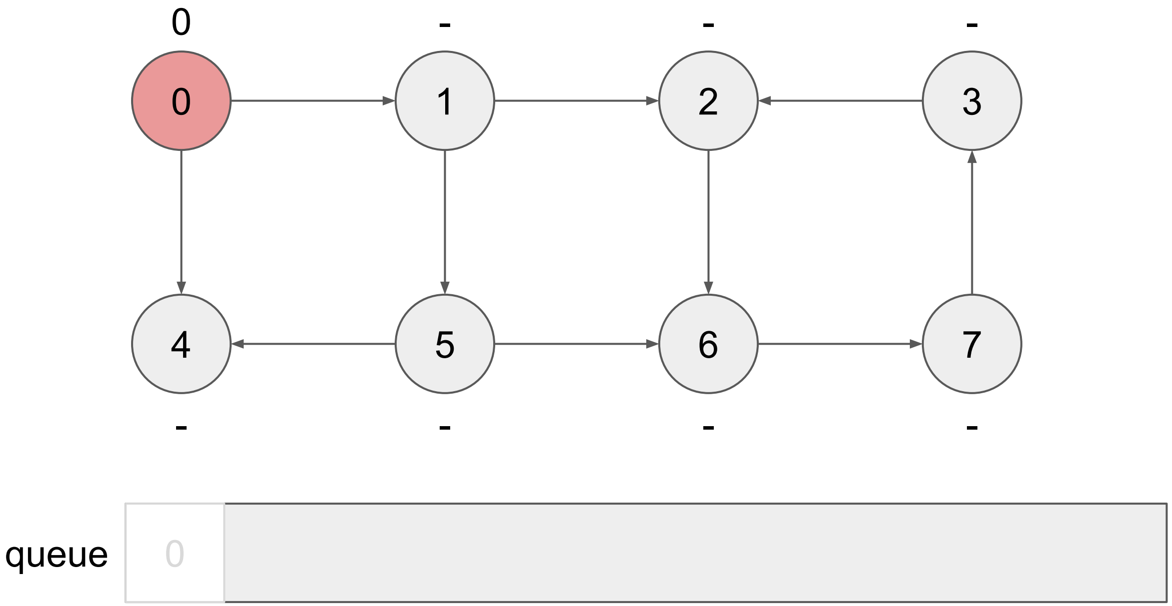
廣度優先搜索 BFS

節點0為起點，level值設為0，並丟入queue



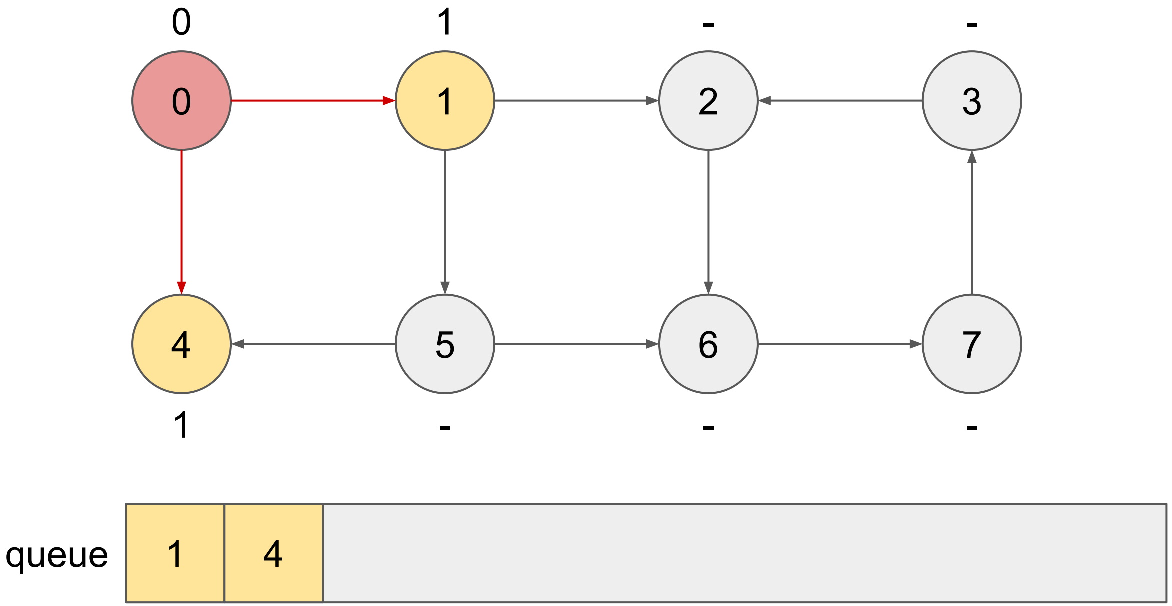
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



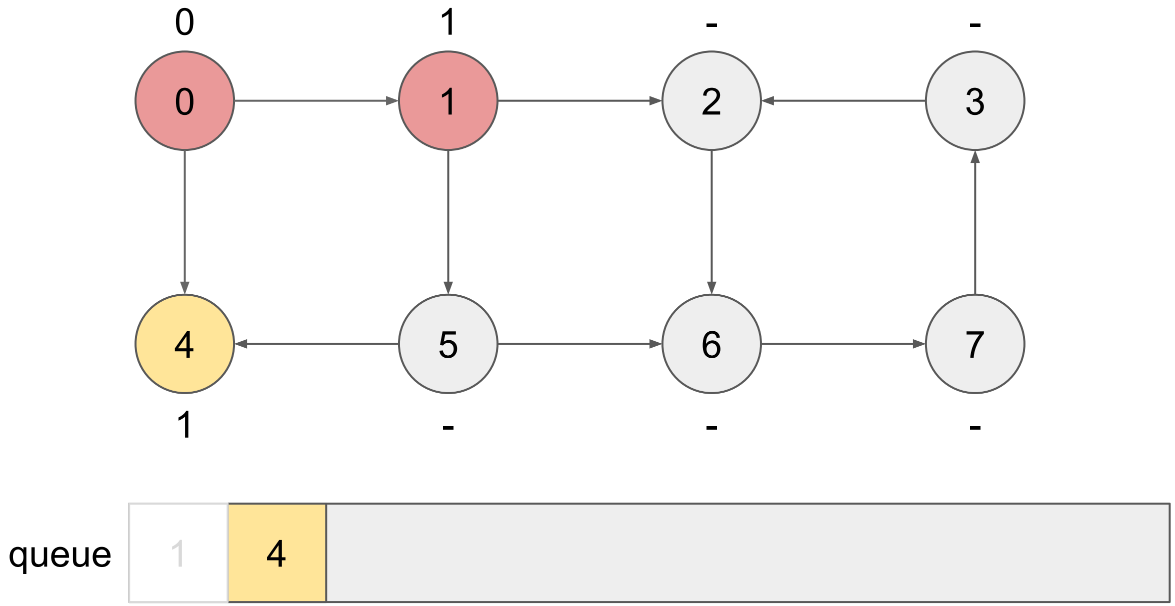
廣度優先搜索 BFS

將節點0相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



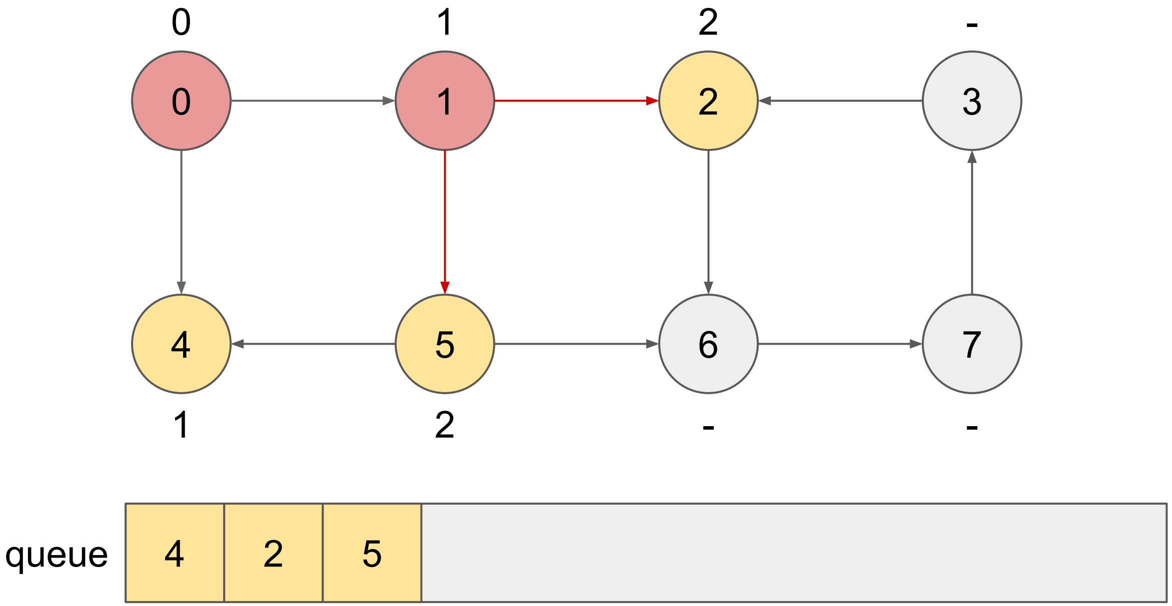
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



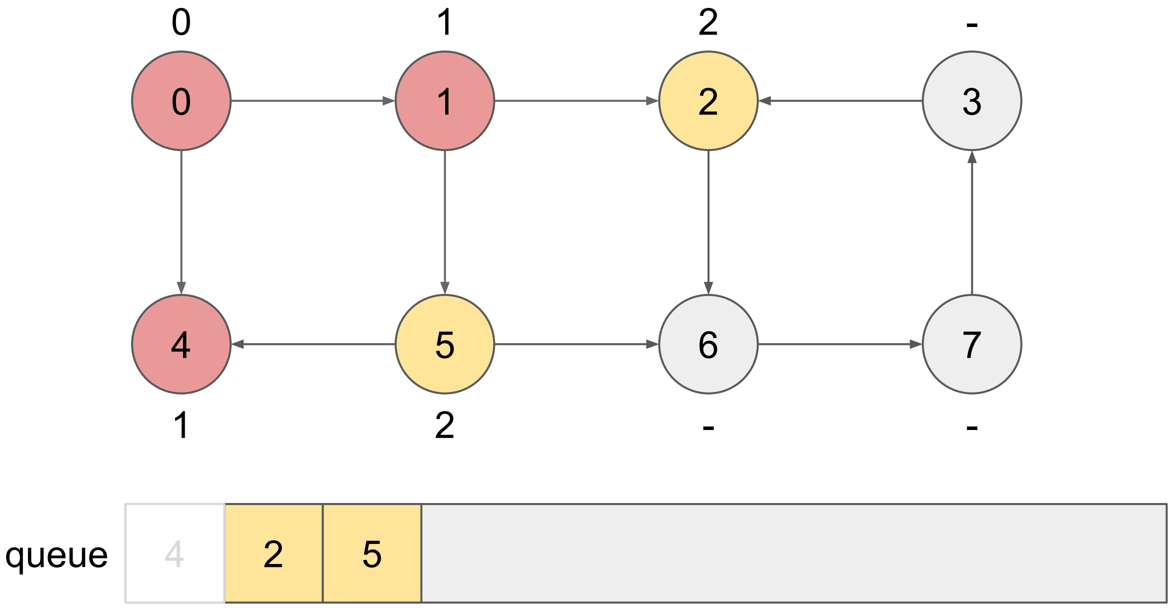
廣度優先搜索 BFS

將節點1相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



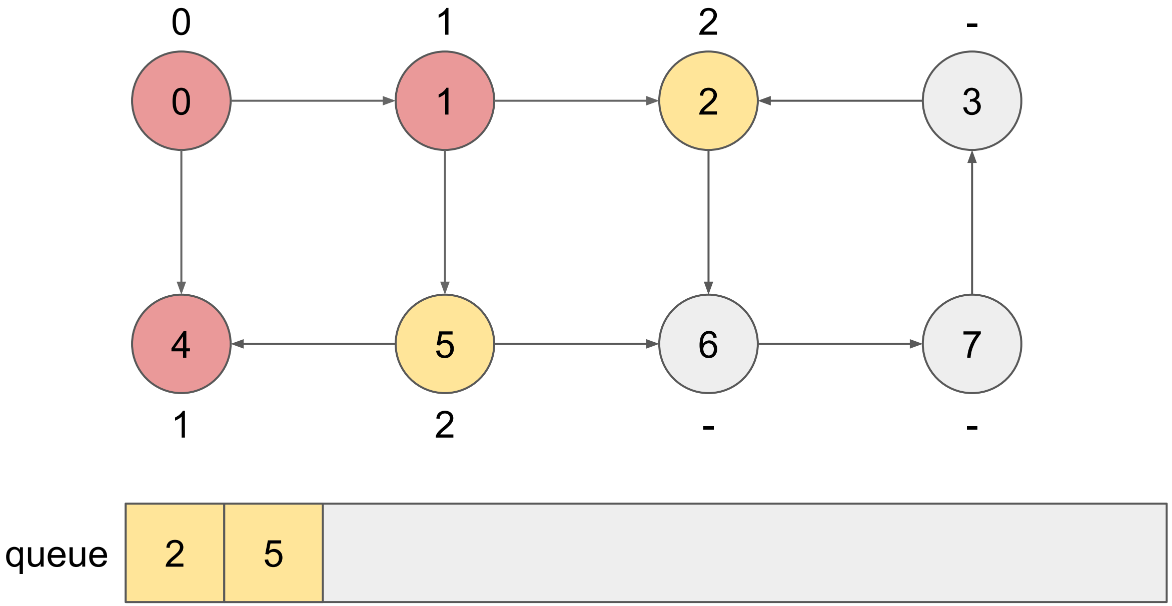
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



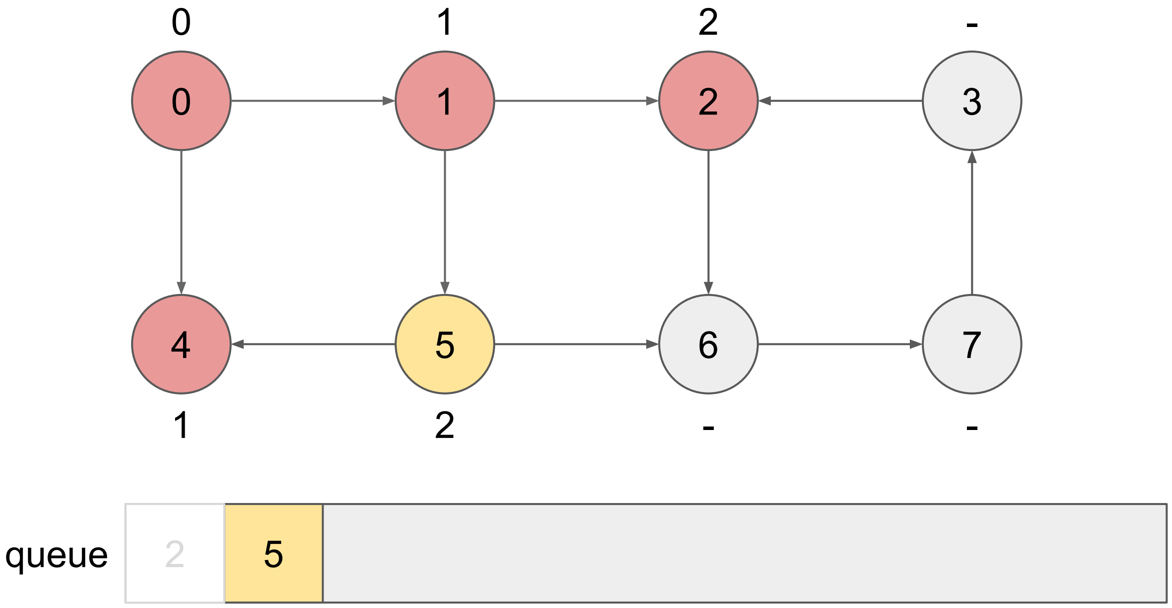
廣度優先搜索 BFS

節點4沒有相鄰且沒有被拜訪過的點，不動作



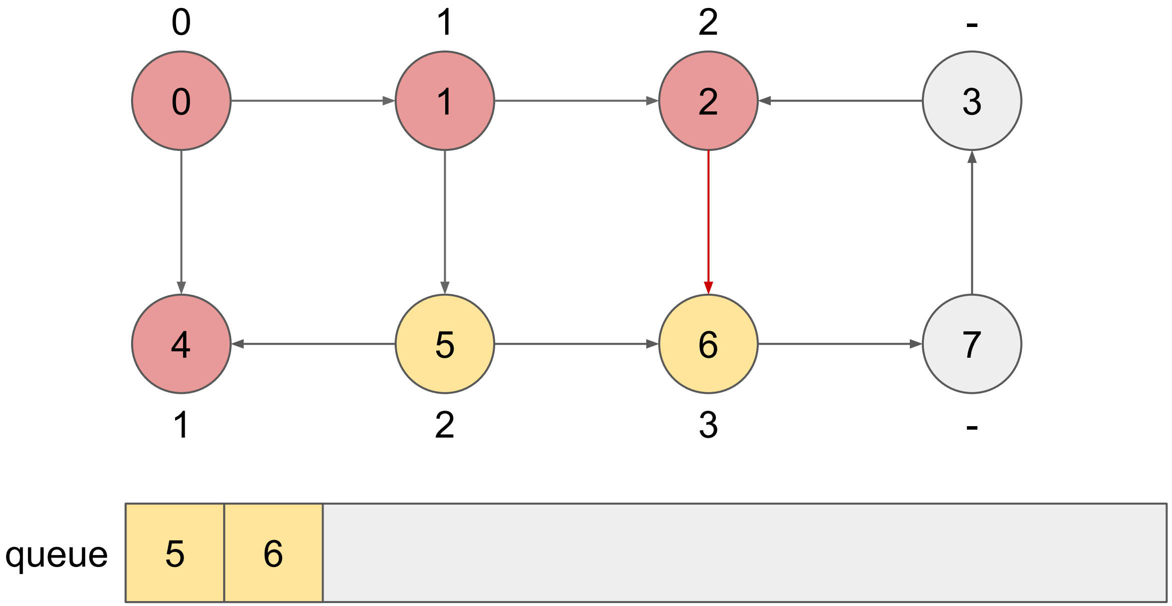
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



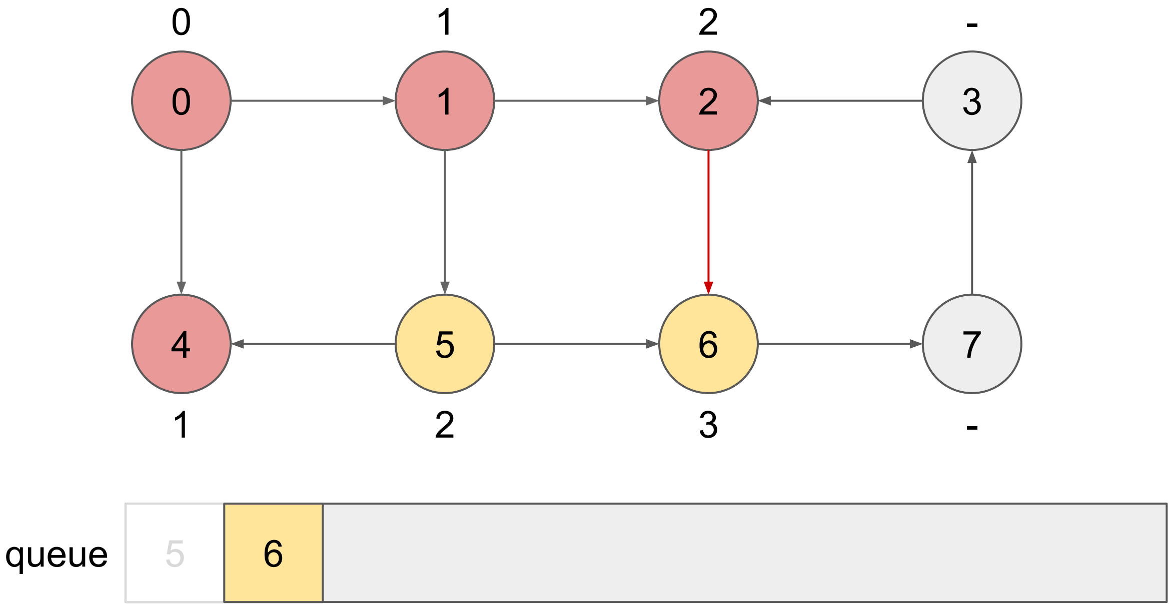
廣度優先搜索 BFS

將節點2相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



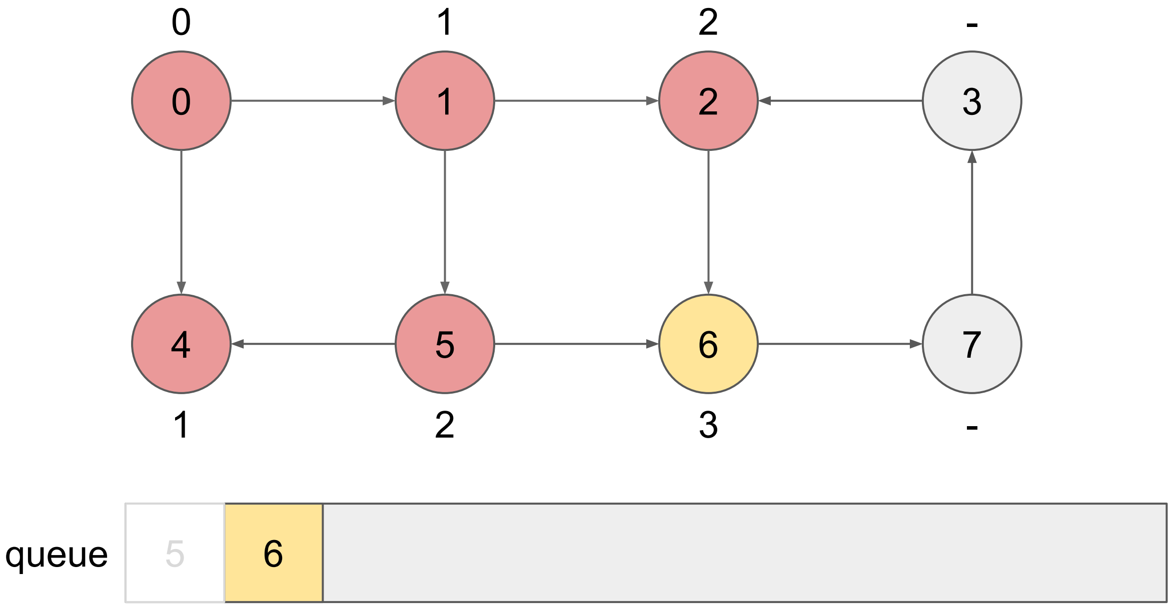
廣度優先搜索 BFS

將節點2相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



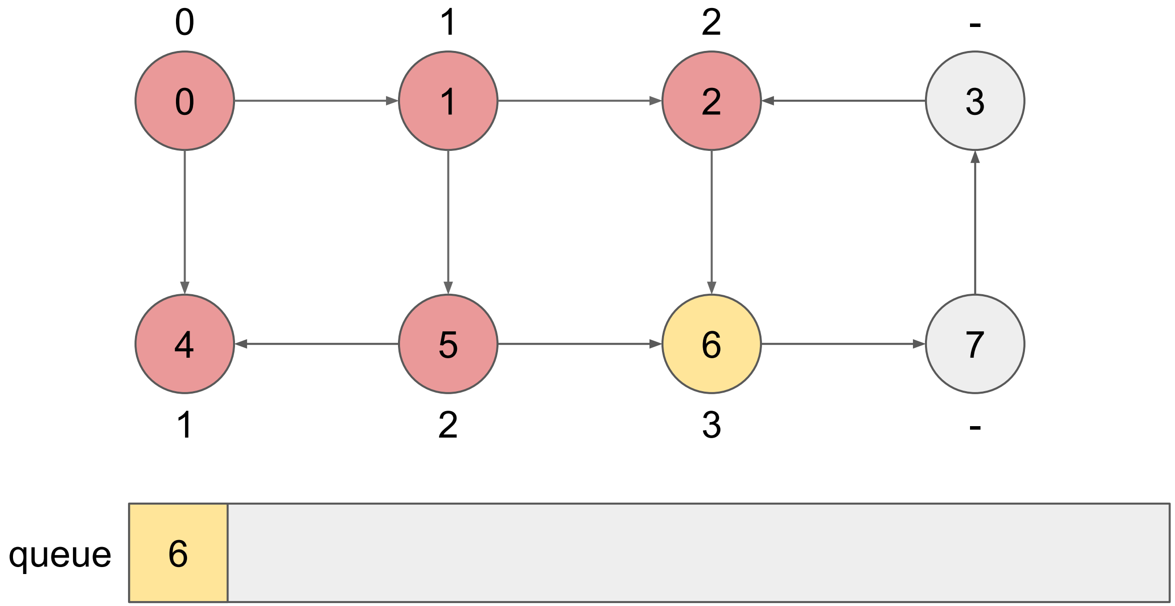
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



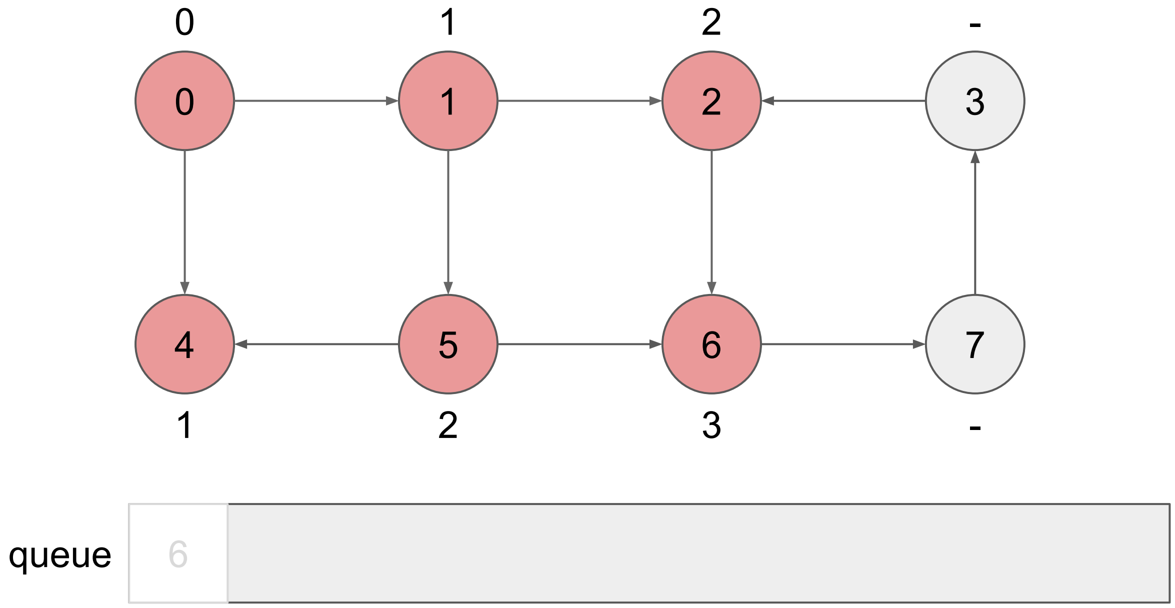
廣度優先搜索 BFS

節點5沒有相鄰且沒有被拜訪過的點，不動作



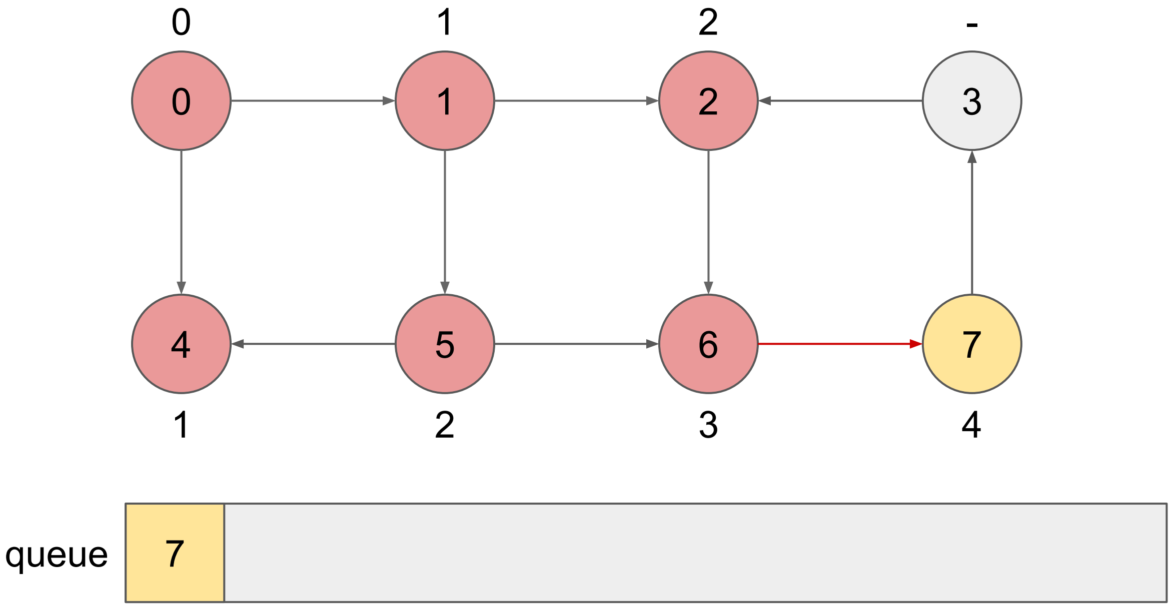
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



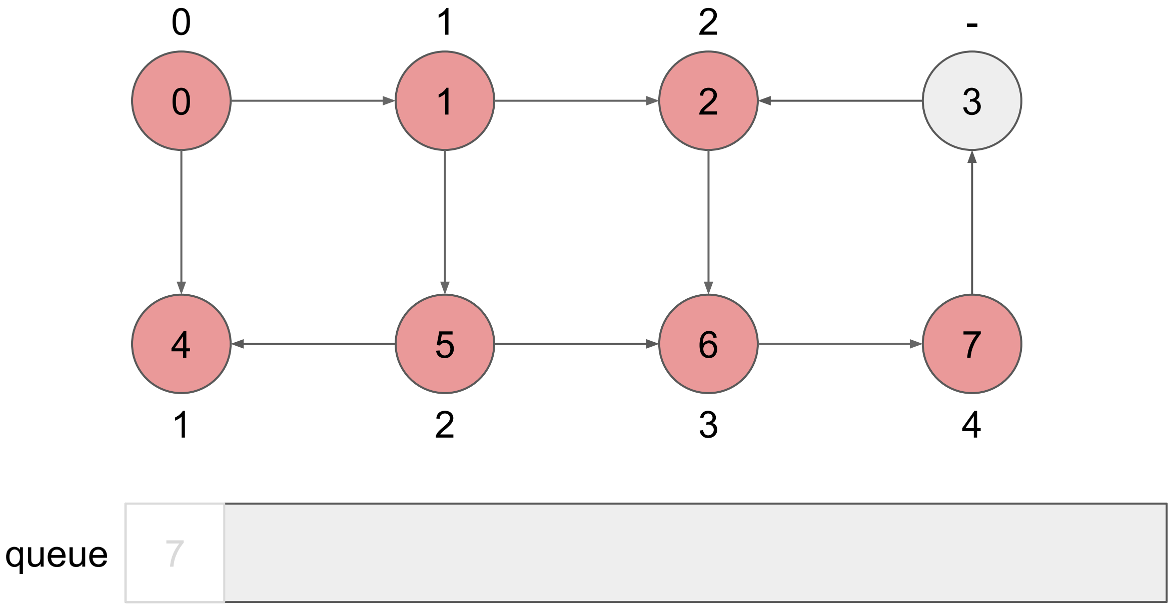
廣度優先搜索 BFS

將節點6相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



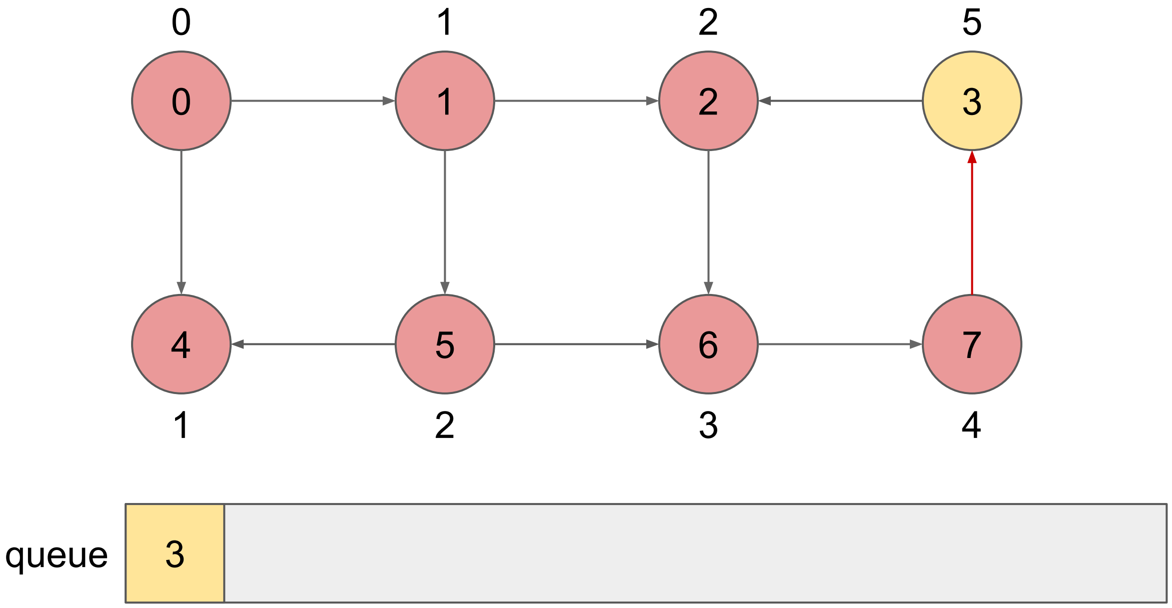
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



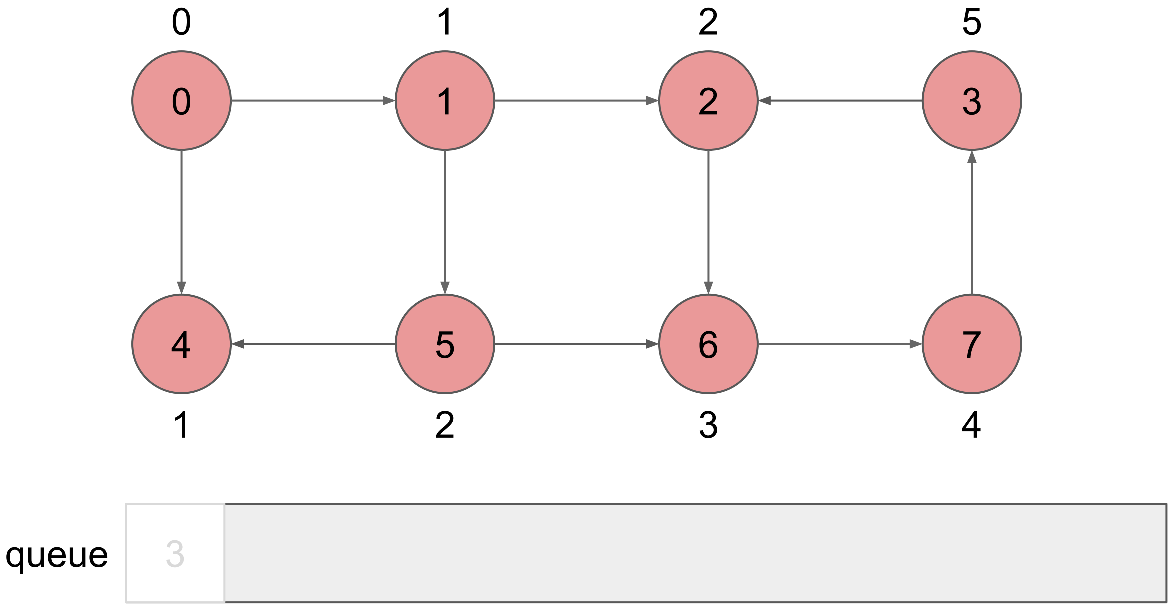
廣度優先搜索 BFS

將節點7相鄰且沒有被拜訪過的點丟進 queue，並將該level值+1



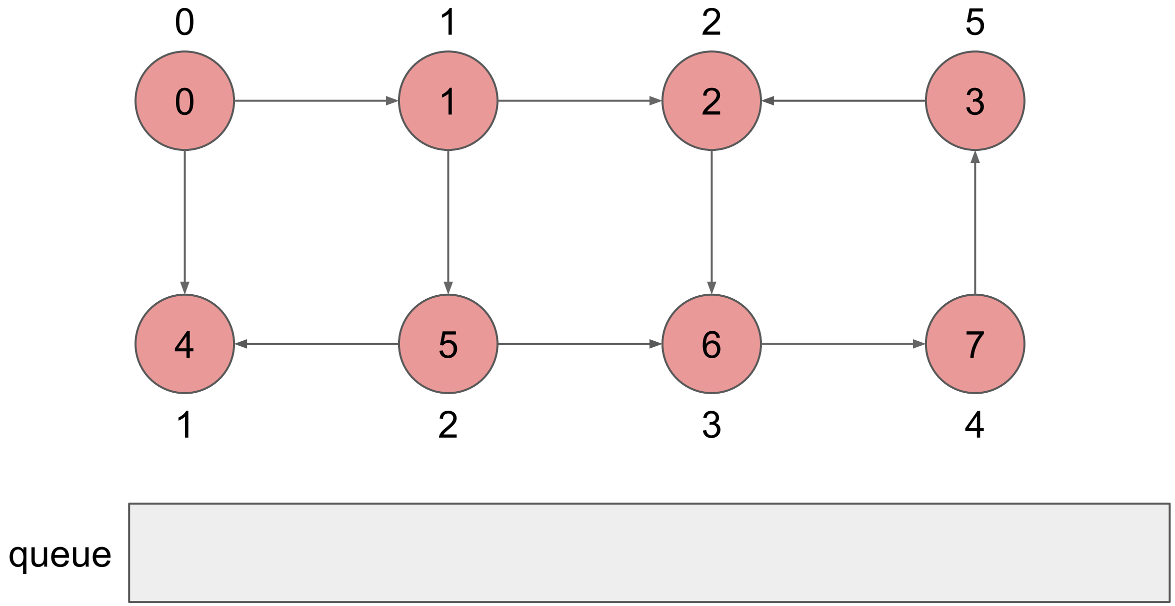
廣度優先搜索 BFS

從queue中取最前面的節點，並 pop掉該節點



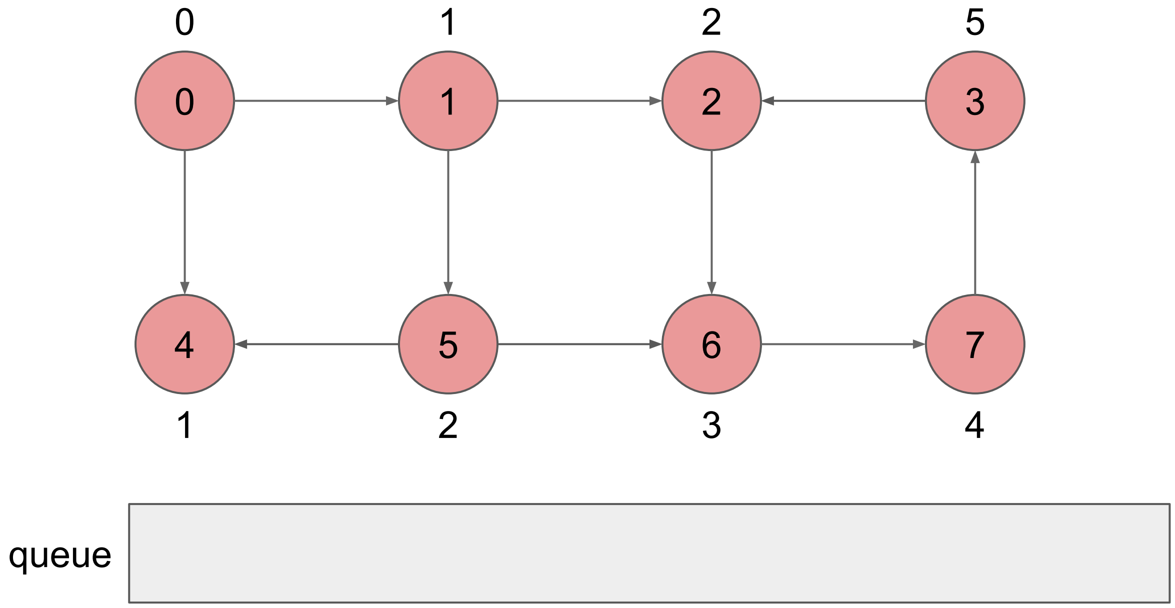
廣度優先搜索 BFS

節點3沒有相鄰且沒有被拜訪過的點，不動作



廣度優先搜索 BFS

queue為空且每個點都走訪完畢，完成 BFS



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

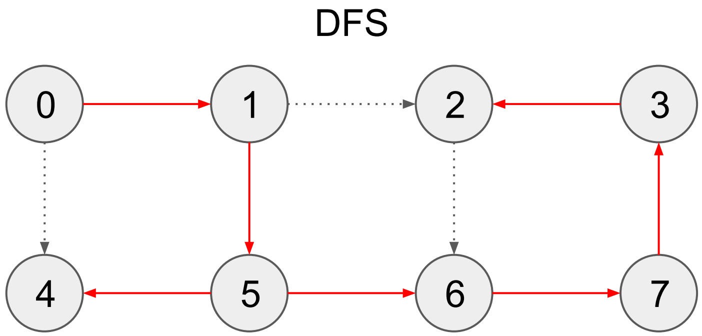
```
const int Vertex = 1000;
vector<int> Graph[Vertex];
int level[Vertex];
void init() {
    memset(level, -1, sizeof(level));
}
void bfs(int s) {
    queue<int> q;
    q.push(s); level[s] = 0;         //將起點丟到起點裡面
    while (q.size()) {               //直到queue是空的才停止
        int u = q.front(); q.pop();  //拿出qeueu最前面的頂點
        for (int v : Graph[u]) {     //遍尋與u連接的所有點
            if (level[u] != -1) {    //此點拜訪過就不理
                continue;
            }
            level[v] = level[u] + 1; //更新level
            q.push(v);
        }
    }
}
```

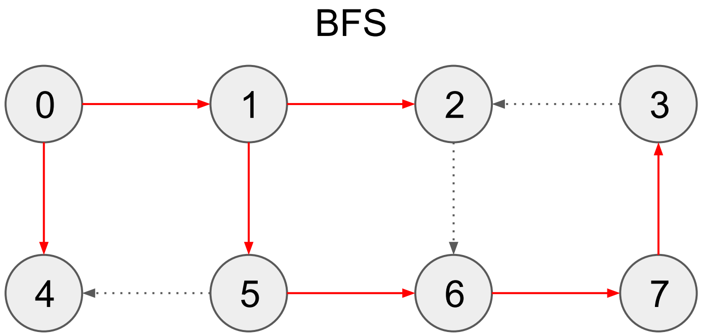
DFS & BFS

- 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



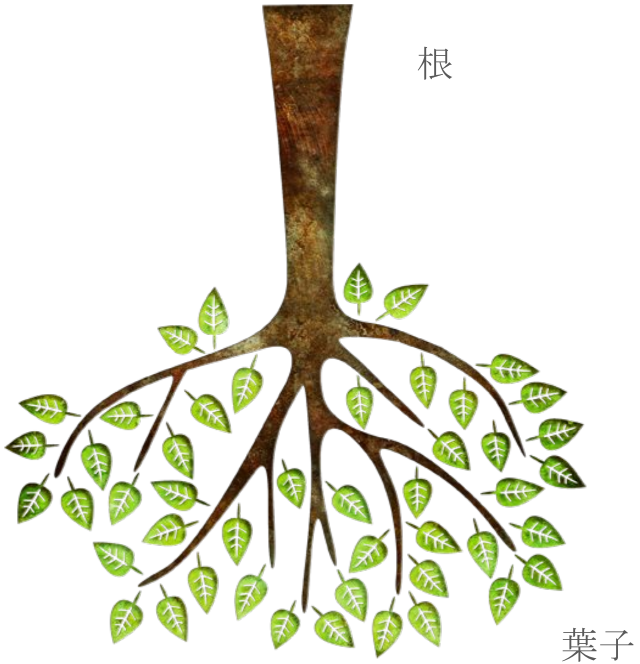


樹

- 這是一棵仿真實世界的樹

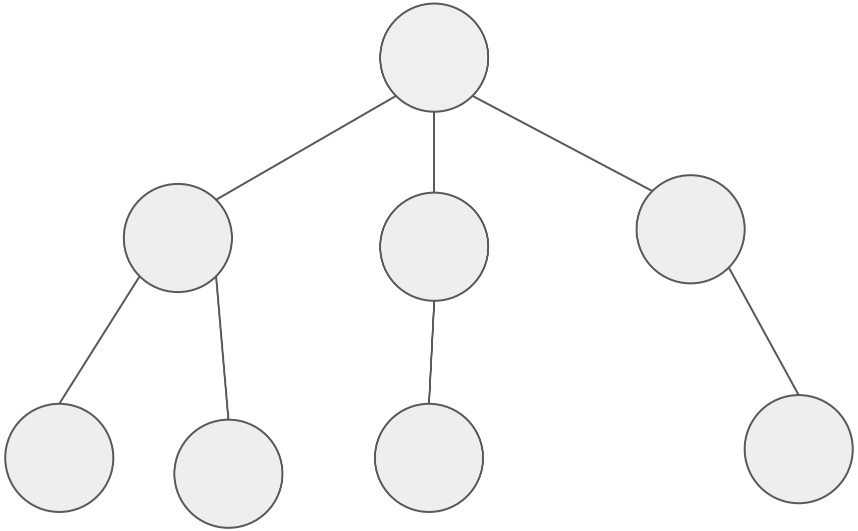
樹

- 這是一棵仿真實世界的樹
- 但資工的樹通常是反過來的



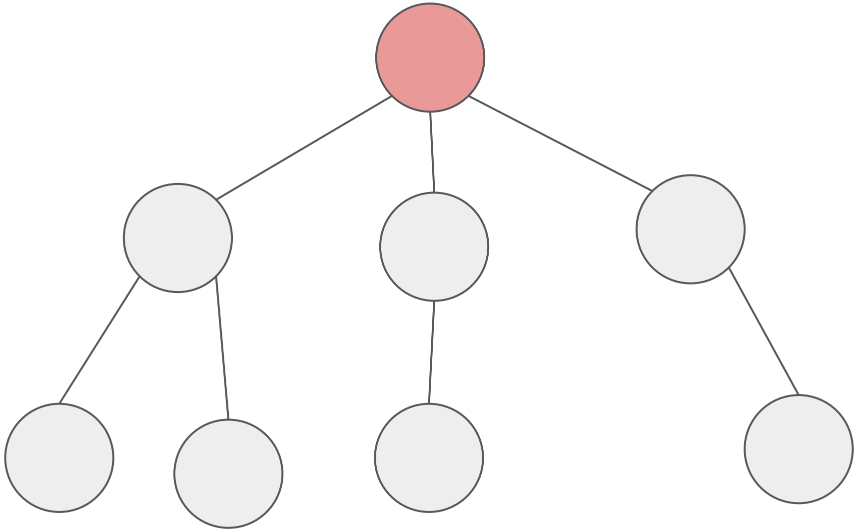
樹

- 樹的一些重要名詞



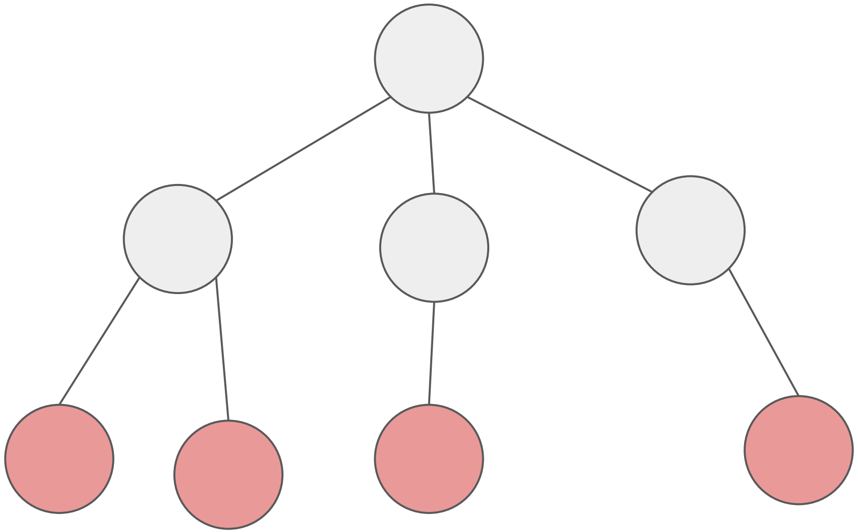
樹

- 根
  - 位於樹的最頂端



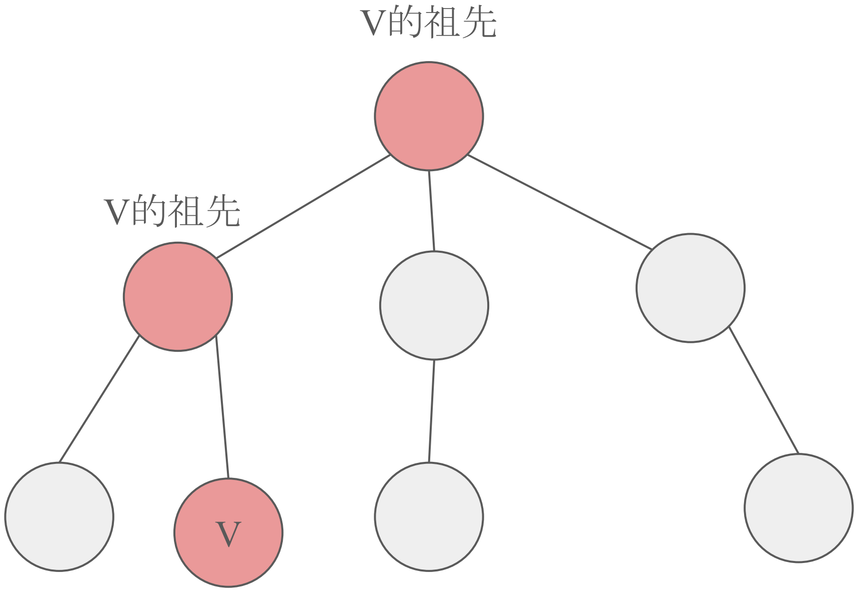
樹

- 葉子
  - 位於樹的最底端



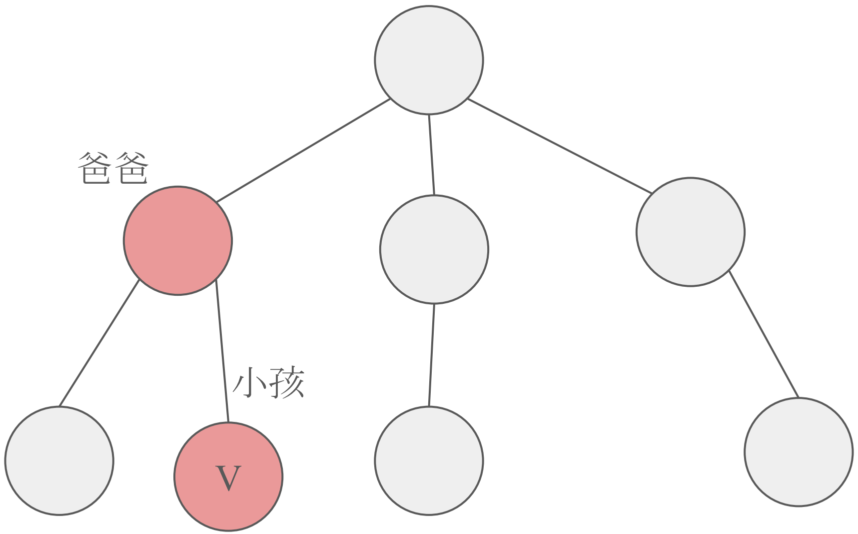
樹

- 點和點關係
  - 祖先
    - 該點到根的路徑經過的點



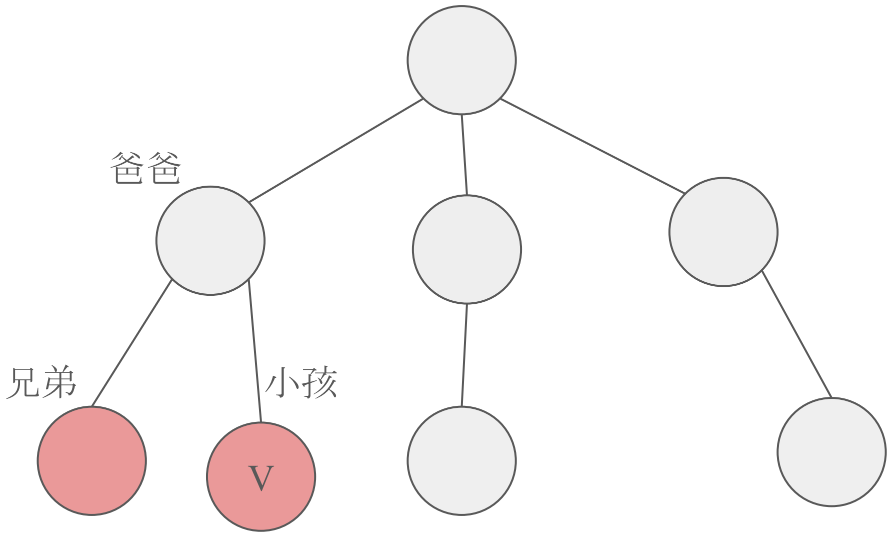
樹

- 點和點關係
  - 小孩和爸爸
    - 該點的上一層



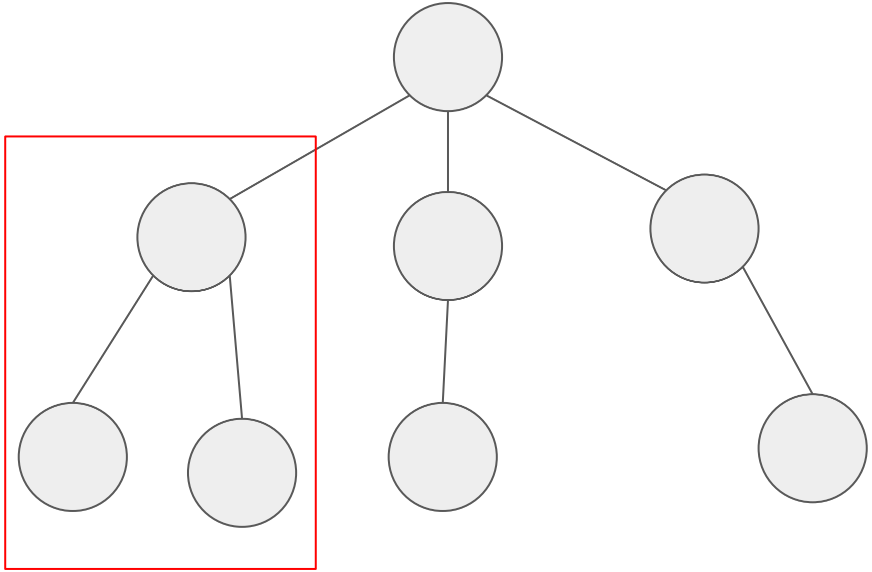
樹

- 點和點關係
  - 兄弟姊妹
    - 同一個爸爸



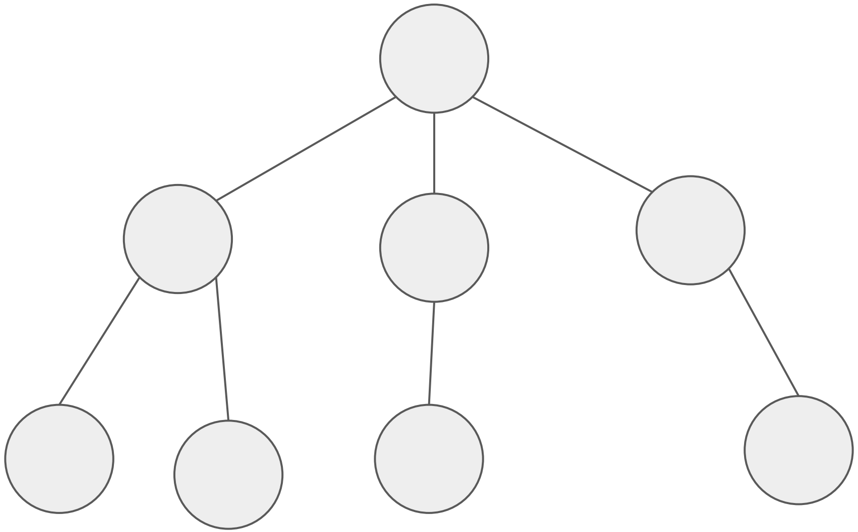
樹

- 子樹
  - 樹的某一部份切下來也會是一棵樹



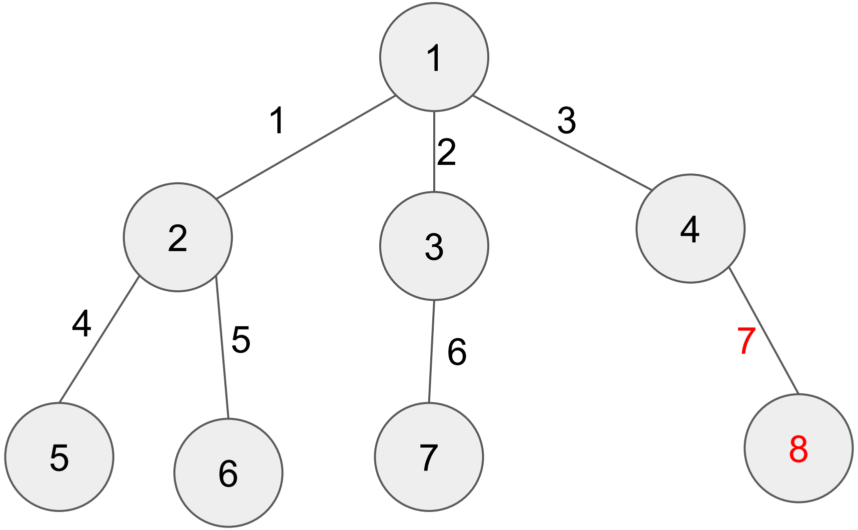
樹

- 樹的一些重要性質



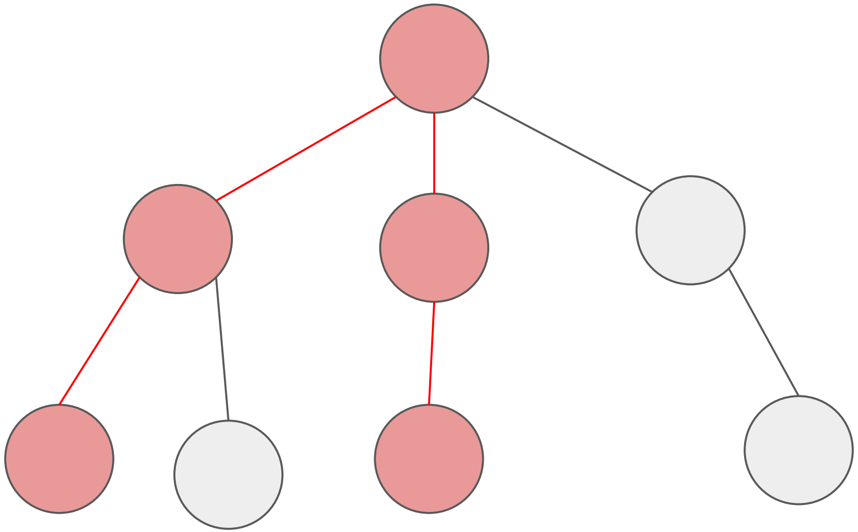
樹

- 樹的一些重要性質
  - n個點的樹僅有n-1條邊



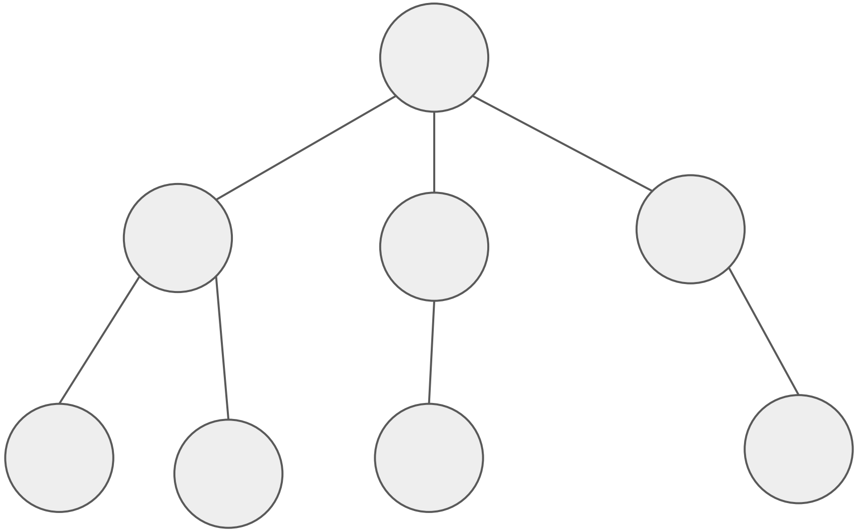
樹

- 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



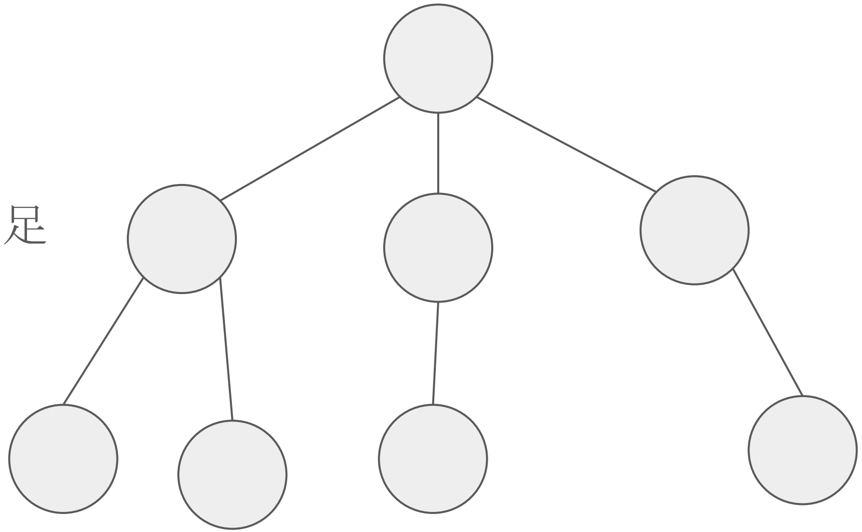
樹

- 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



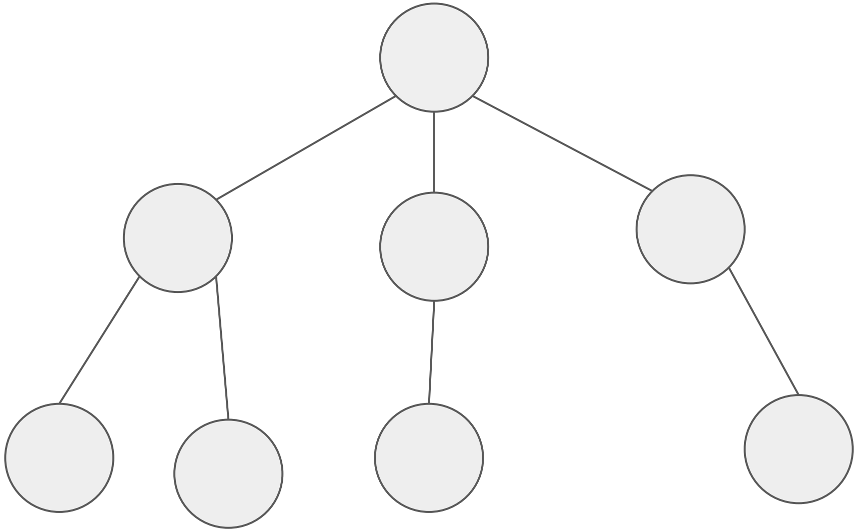
樹

- 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



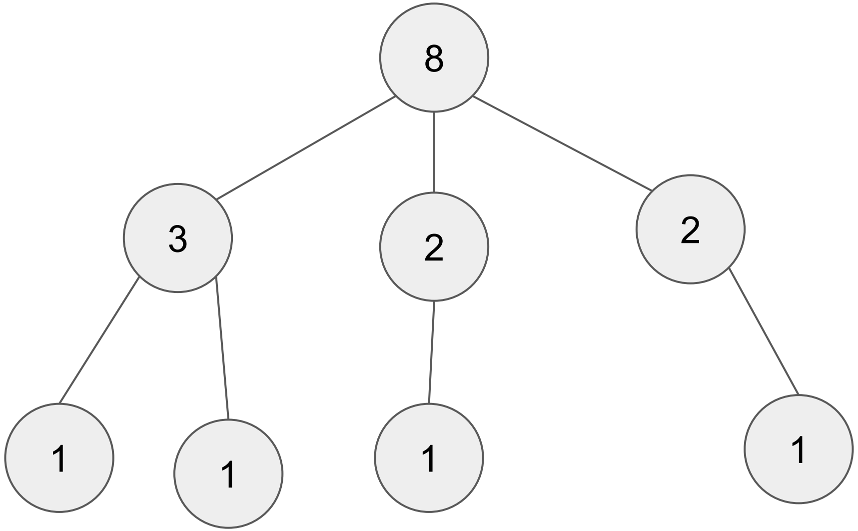
樹的問題 (子樹的size)

- 建立出每一個子樹他的size有多大



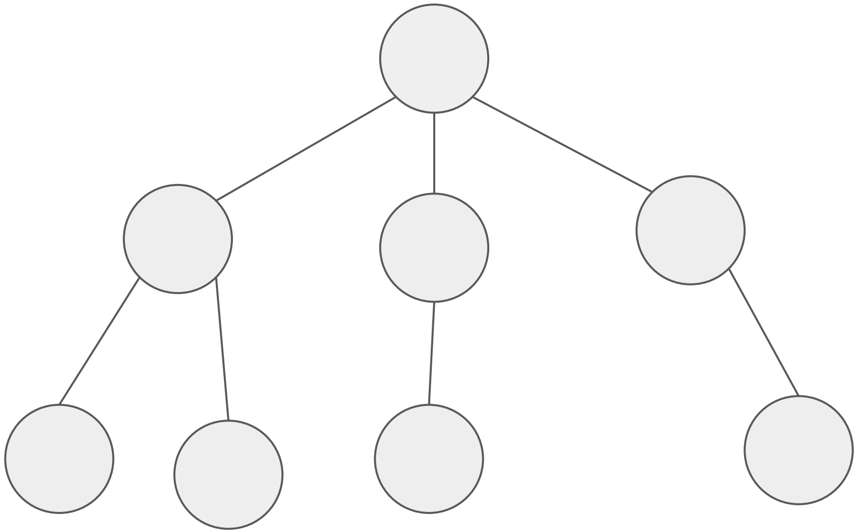
樹的問題 (子樹的size)

- 建立出每一個子樹他的size有多大
- 以右圖來說是這樣



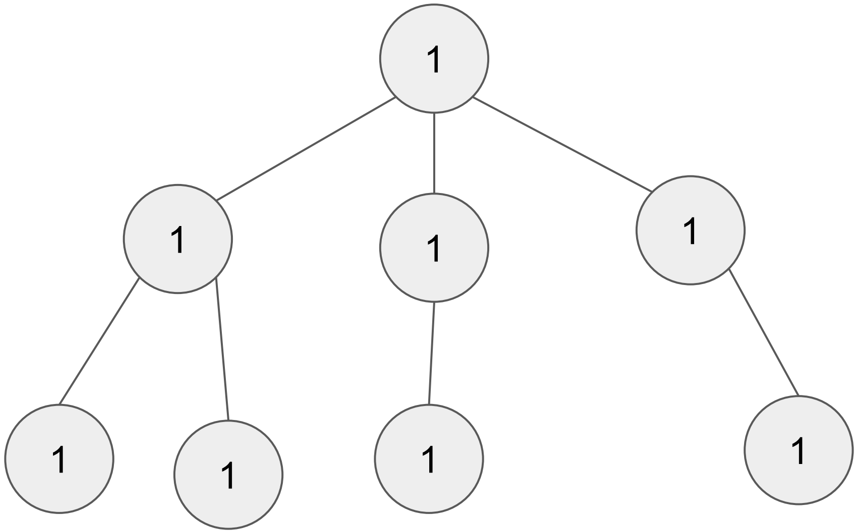
樹的問題 (子樹的size)

- 方法:一次DFS



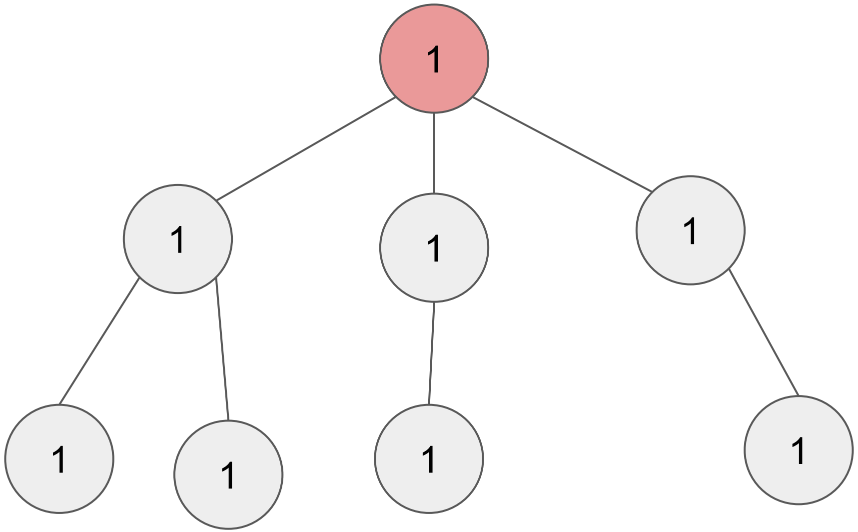
樹的問題 (子樹的size)

- 方法: 一次DFS
- 先將每一個節點size值設成1



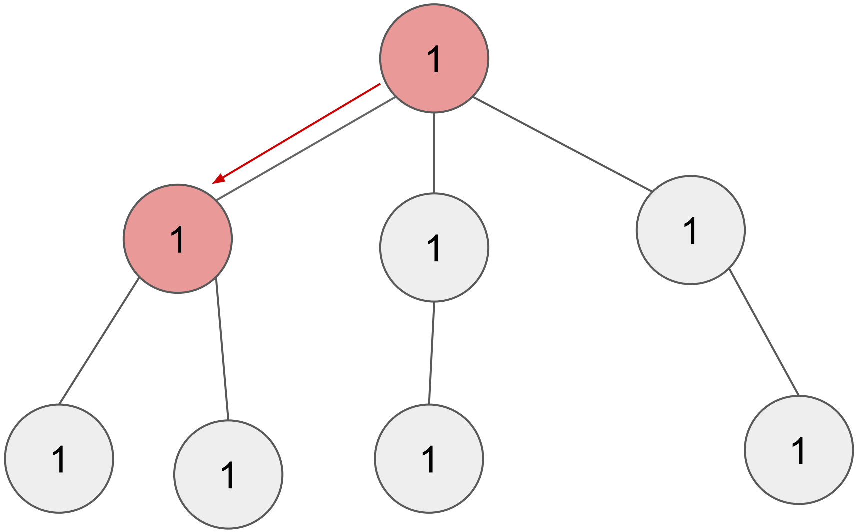
樹的問題 (子樹的size)

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



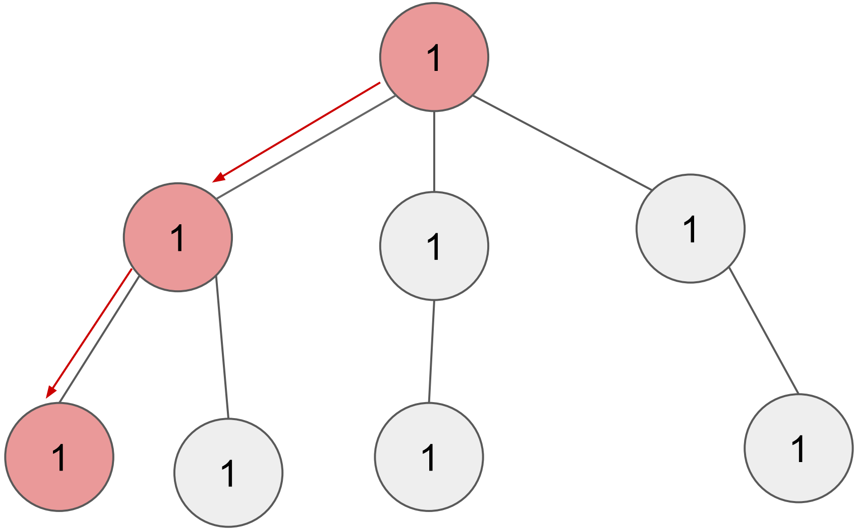
樹的問題 (子樹的size)

- 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



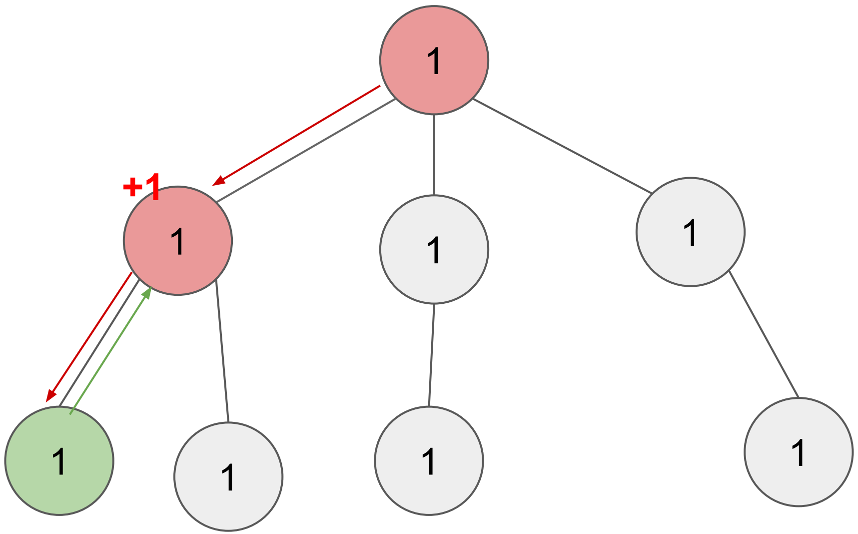
樹的問題 (子樹的size)

- 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



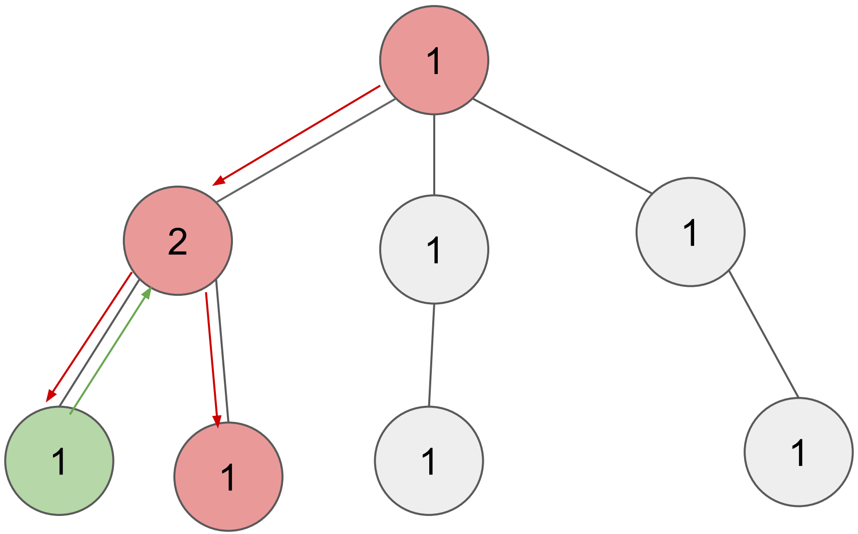
樹的問題 (子樹的size)

- 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



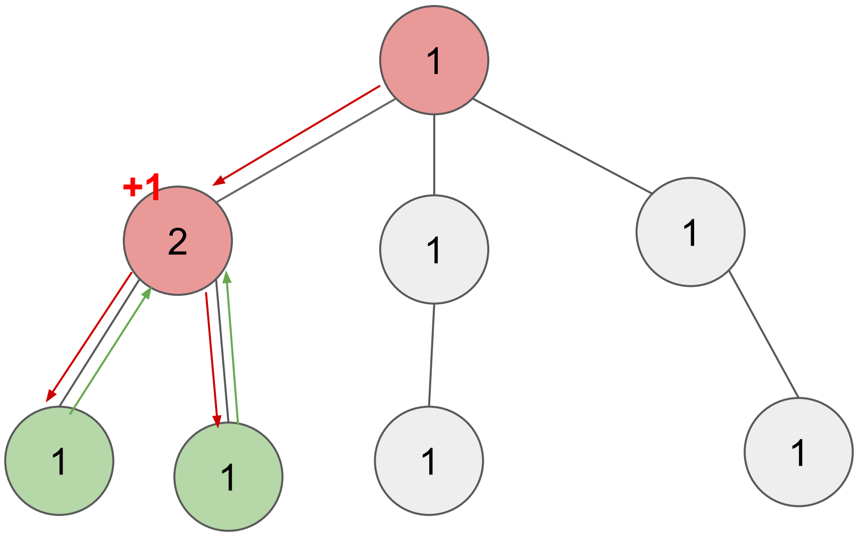
樹的問題 (子樹的size)

- 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



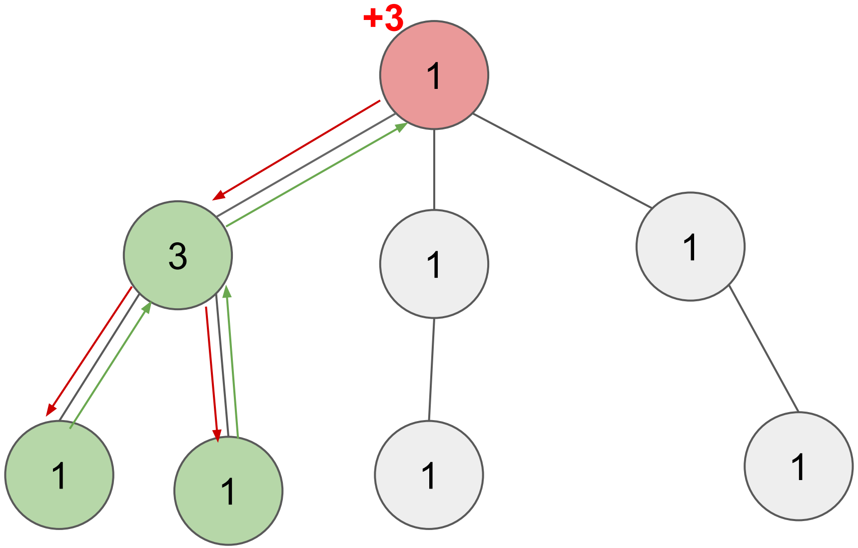
樹的問題 (子樹的size)

- 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



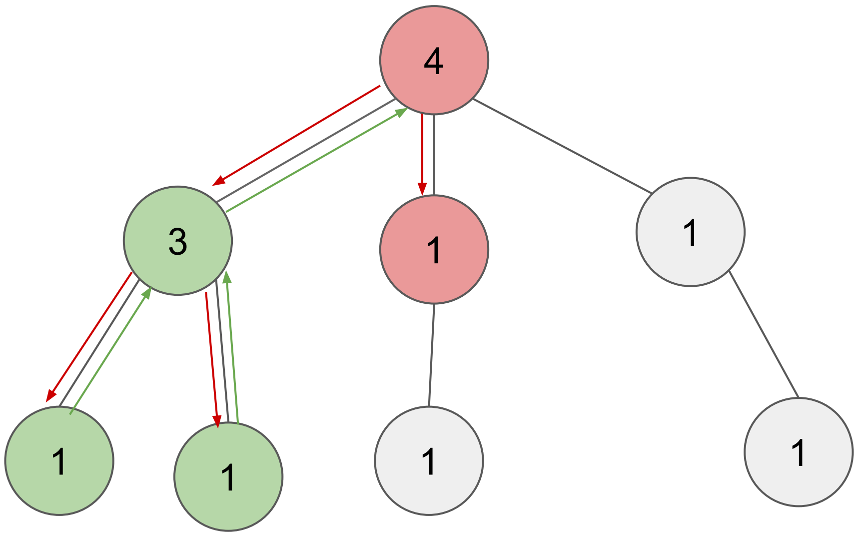
樹的問題 (子樹的size)

- 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



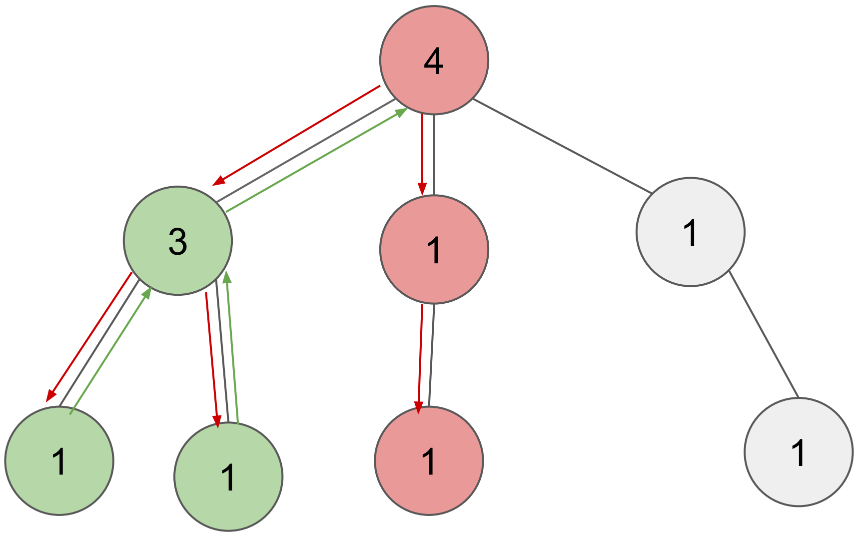
樹的問題 (子樹的size)

- 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



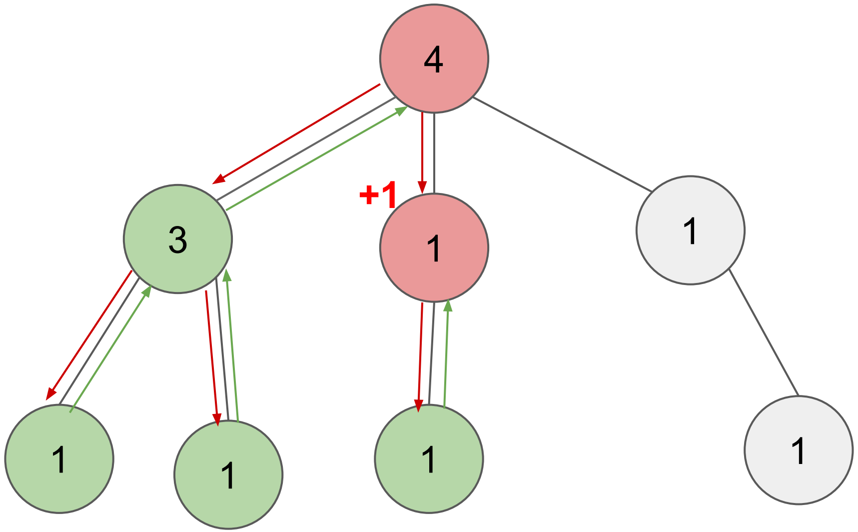
樹的問題 (子樹的size)

- 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



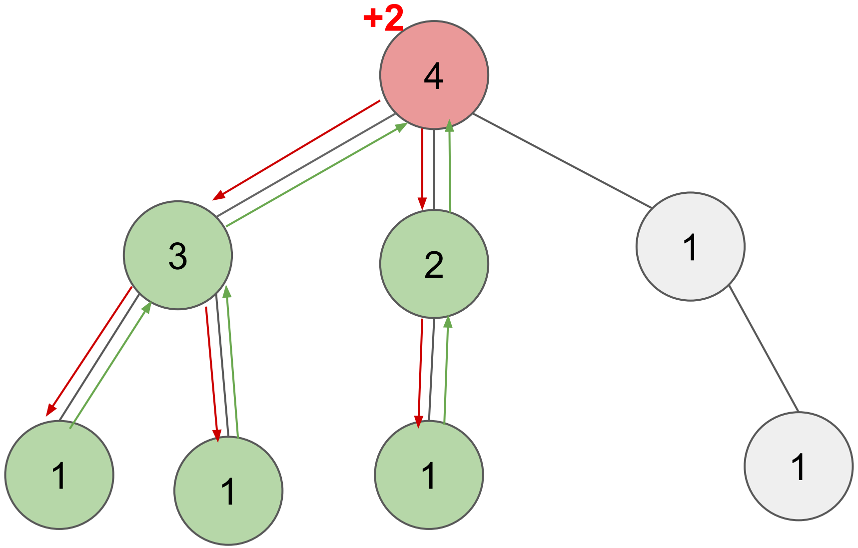
樹的問題 (子樹的size)

- 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



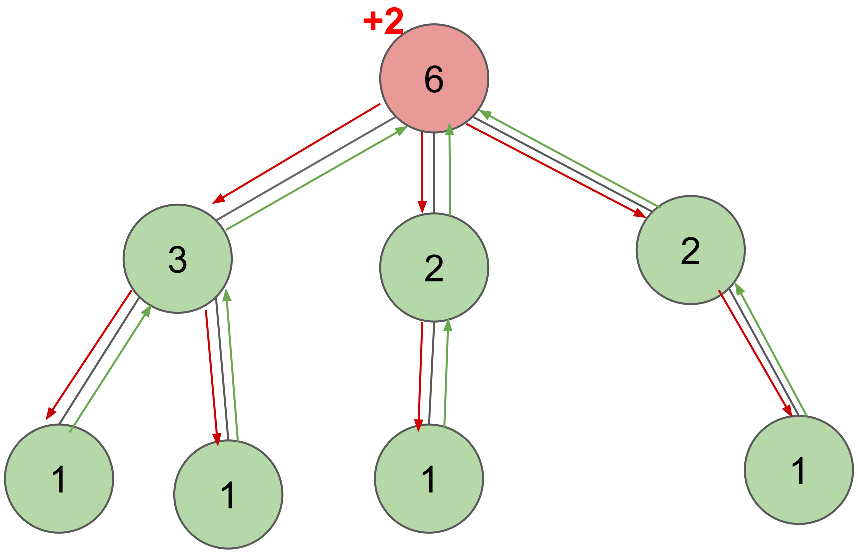
樹的問題 (子樹的size)

- 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



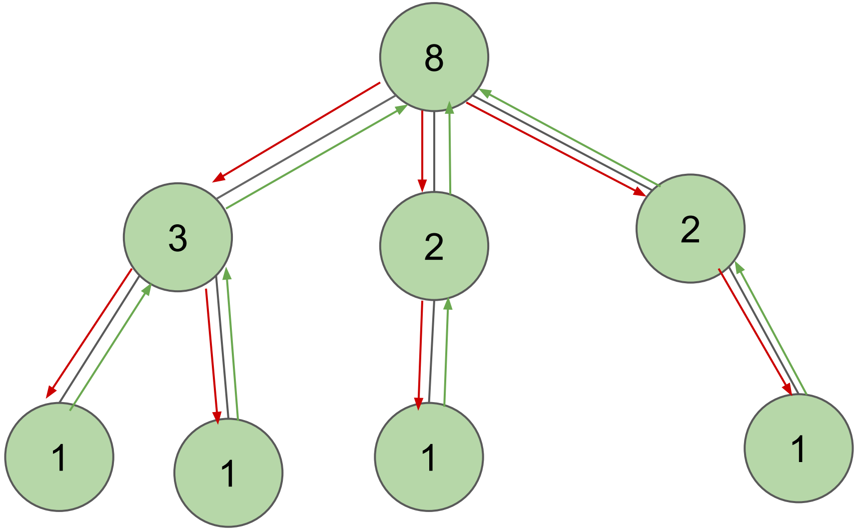
樹的問題 (子樹的size)

- 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)

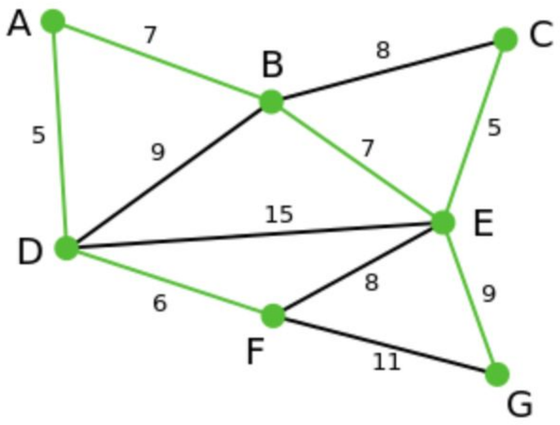


樹的問題 (子樹的size) Code

```cpp
const int Vertex = 1000;
vector<int> Graph[Vertex];
int sz[Vertex];
int dfs(int u, int parent) {
    sz[u] = 1;                    //初始化每一個節點的sz
    for (int v : Graph[u]) {//dfs 每一個節點
        if (v != parent) {  //如果v不是u 的parent那就dfs這個點
            sz[u] += dfs(v, u); //把小孩v的sz加到sz[u]裡面
        }
    }
    return sz[u];
}
```

樹的問題 (最小生成樹)

- 給定有權重的聯通圖找出一張子圖，在這張的子圖中選出**權重總和最小的生成樹**。例如下圖中綠色的部分，就是這個圖的最小生成樹

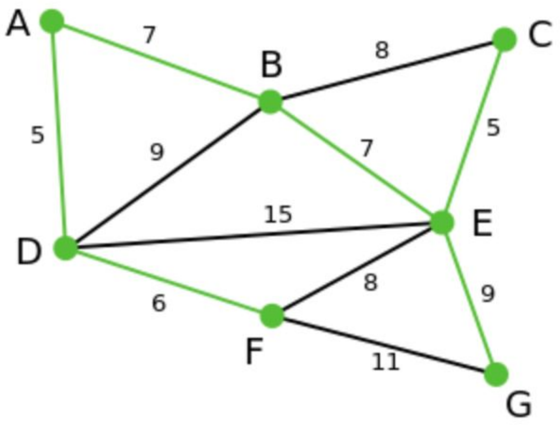


樹的問題 (最小生成樹)

- 方法:

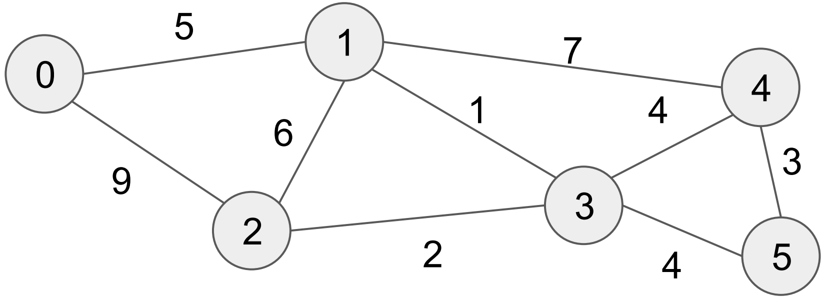
Kruskal's algorithm

- 將所有的邊依照權重由小到大排序
- 從最小開始，選擇不會形成環的邊，直到連接所有節點。



樹的問題 (最小生成樹)

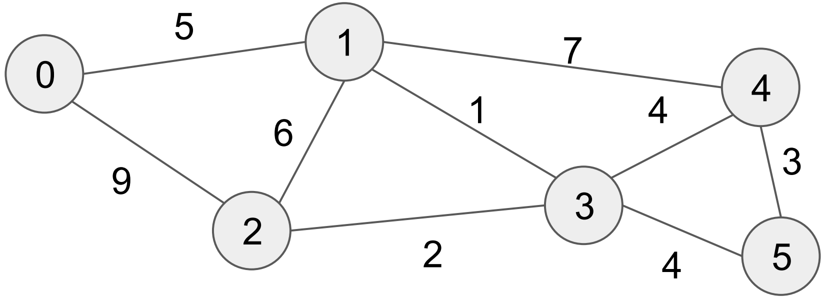
- Kruskal's algorithm
- 用EdgeList方式將所有邊存下來



樹的問題 (最小生成樹)

u	v	w
0	1	5
0	2	9
1	2	6
1	3	1
1	4	7
2	3	2
3	4	4
3	5	4
4	5	3

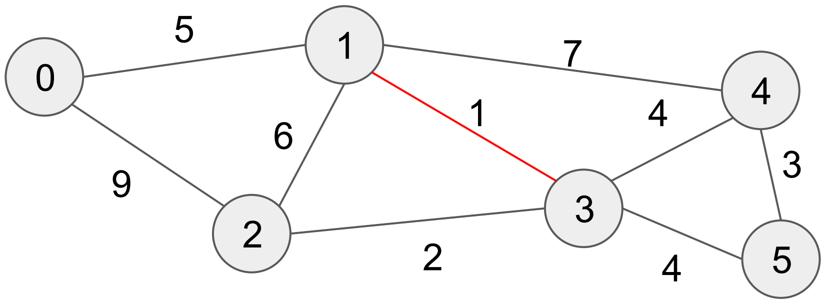
將edgeList 從小到大做 sorting



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

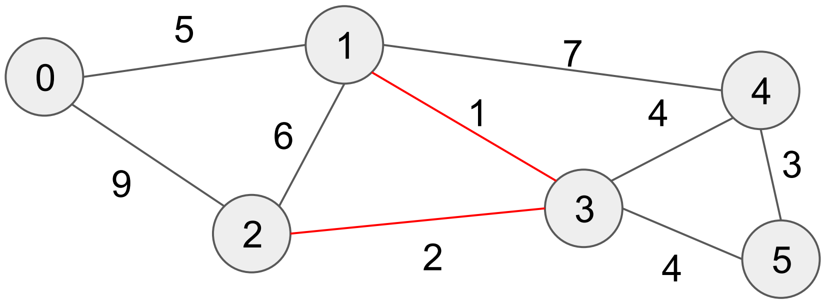
(1, 3) 加入生成樹中不會形成環



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

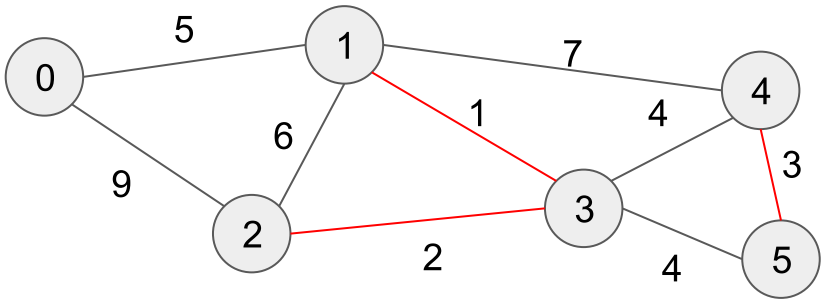
(2, 3) 加入生成樹中不會形成環



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

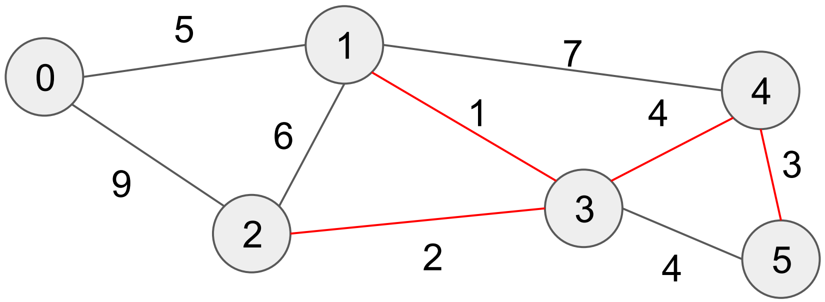
(4, 5) 加入生成樹中不會形成環



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

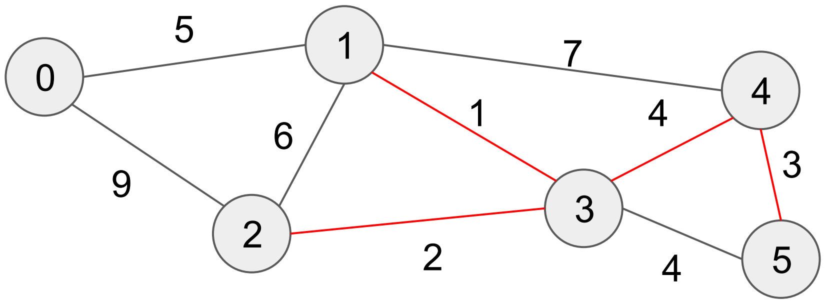
(3, 4) 加入生成樹中不會形成環



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

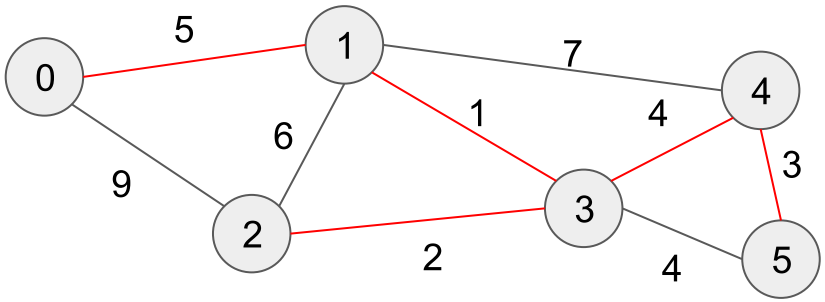
(3, 5) 加入生成樹中會形成環！！



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

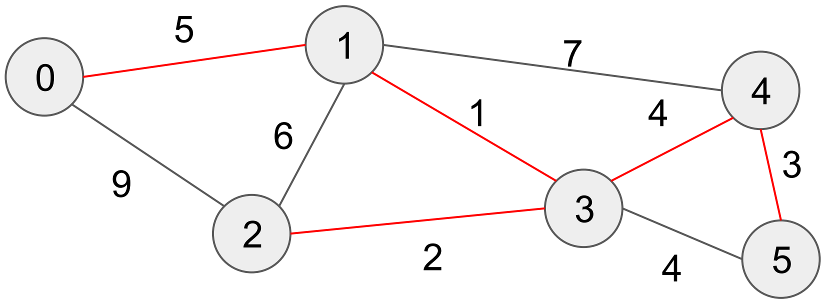
(0, 1) 加入生成樹中不會形成環



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

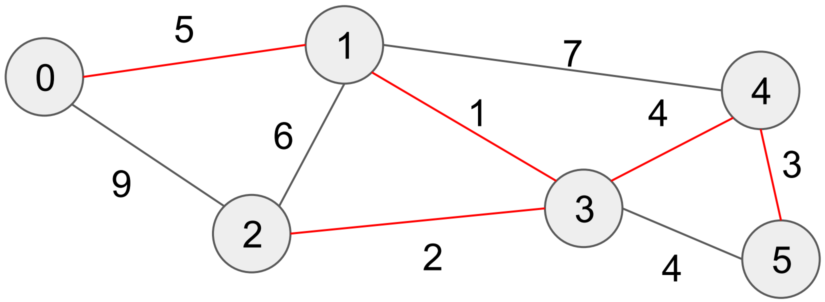
(1, 2) 加入生成樹中會形成環！！



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

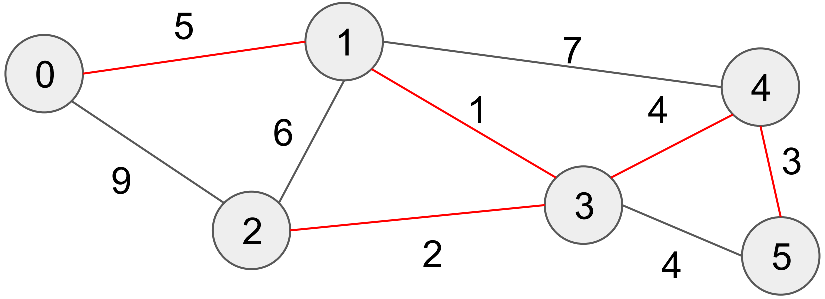
(1, 4) 加入生成樹中會形成環！！



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

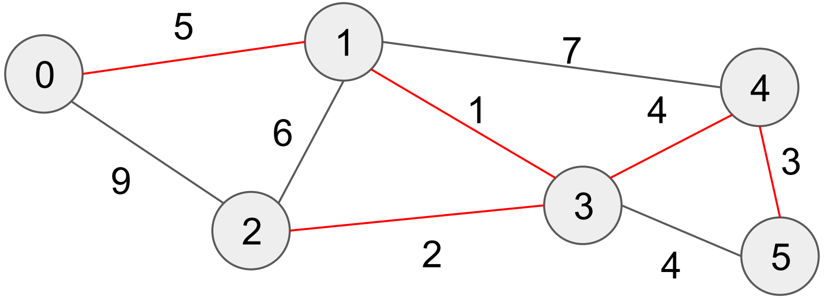
(0, 2) 加入生成樹中會形成環！！



樹的問題 (最小生成樹)

u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

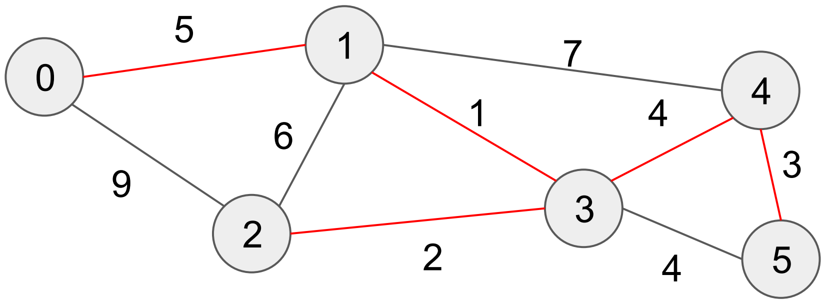
(0, 2) 加入生成樹中會形成環！！



樹的問題 (最小生成樹)

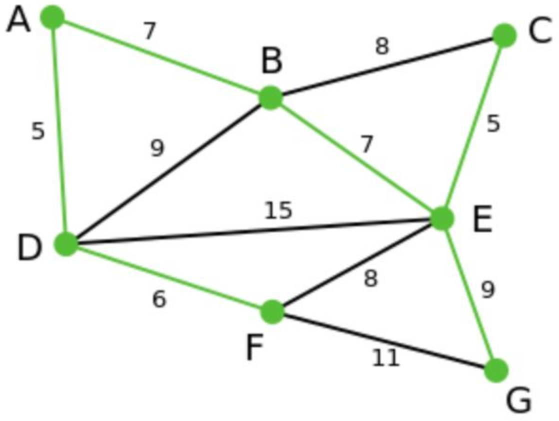
u	v	w
1	3	1
2	3	2
4	5	3
3	4	4
3	5	4
0	1	5
1	2	6
1	4	7
0	2	9

最小生成數的權重為 15 ！



樹的問題 (最小生成樹)

- 方法: Kruskal's algorithm
  - 將所有的邊依照權重由小到大排序
  - 從最小開始，選擇不會形成環的邊，直到連接所有節點。
- 複雜度分析
  - sorting 所以的邊複雜度 O(ElogE)
  - 判斷是否在同一個聯通塊中可以用disjoint set來判斷
  - 總共時間複雜度O(ElogE)



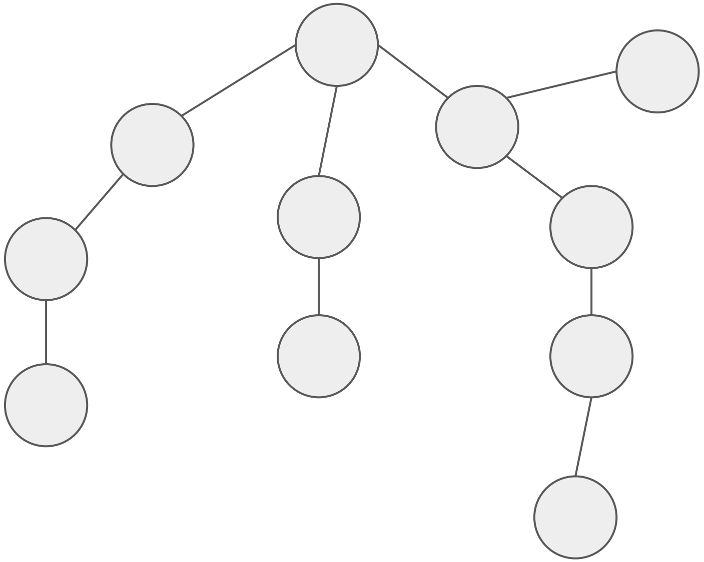
樹的問題 (最小生成數)code

```cpp
struct Edge {
    int u, v, w;
};
bool compare(Edge a, Edge b) {
    return a.w < b.w;
}
vector<Edge> EdgeList;
int kruskal() {
    int weight = 0;
    sort (EdgeList.begin(), EdgeList.end(), compare);
    //sorting EdgeList
    for (auto k : EdgeList) {
        if (merge(k.u, k.v)) { //判斷u v是否在同一個聯通塊中
            weight += k.w;
        }
    }
    return weight;
}

```

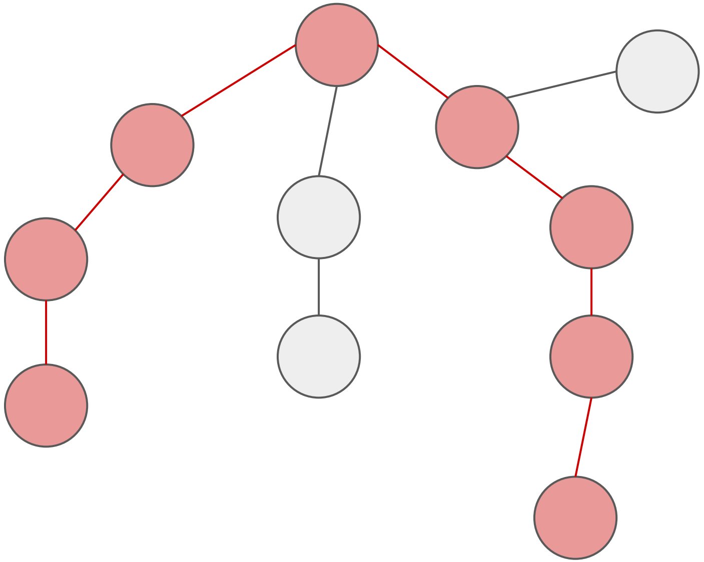
樹的問題 (樹的直徑)

- 找出這棵樹中最長的那條路徑



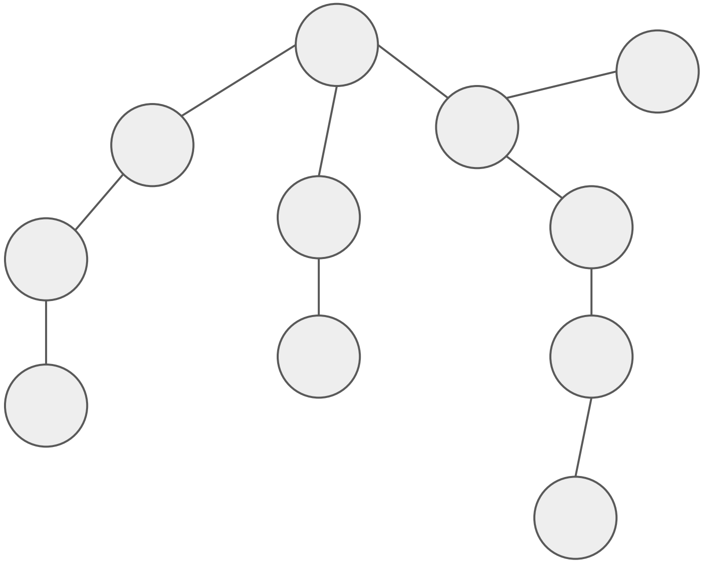
樹的問題 (樹的直徑)

- 找出這棵樹中最長的那條路徑
- 以右圖來說是這樣



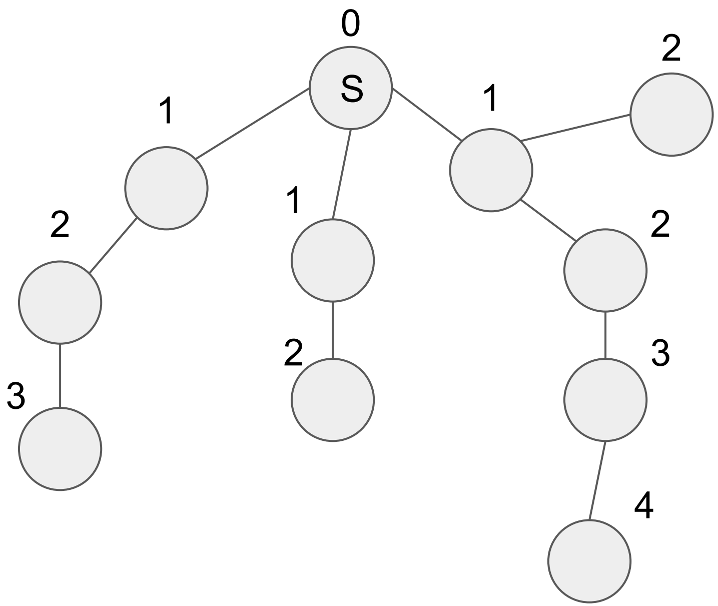
樹的問題 (樹的直徑)

- 方法:兩次BFS



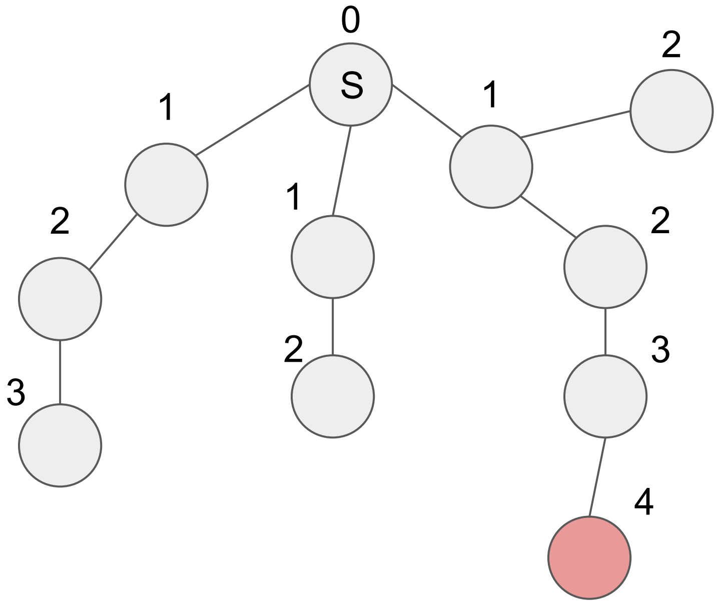
樹的問題 (樹的直徑)

- 方法:兩次BFS
- 隨便對一個點為起點做BFS



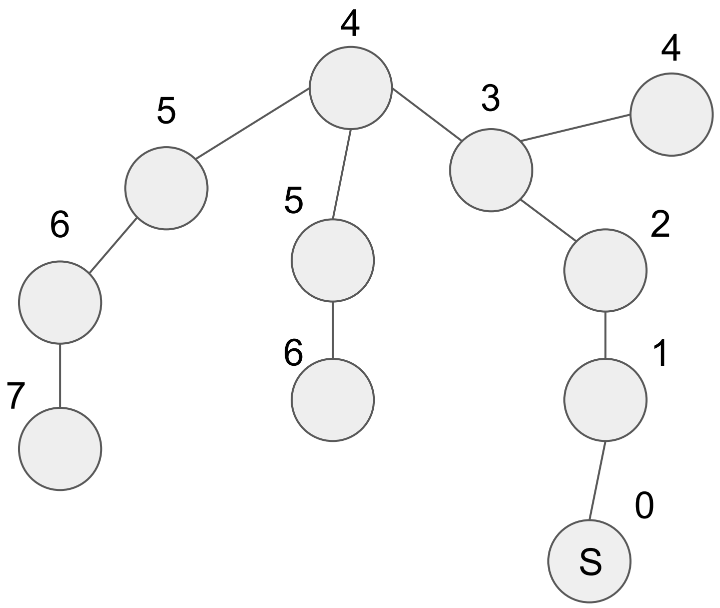
樹的問題 (樹的直徑)

- 方法:兩次BFS
- 隨便對一個點為起點做BFS
- 找出離S最遠的點



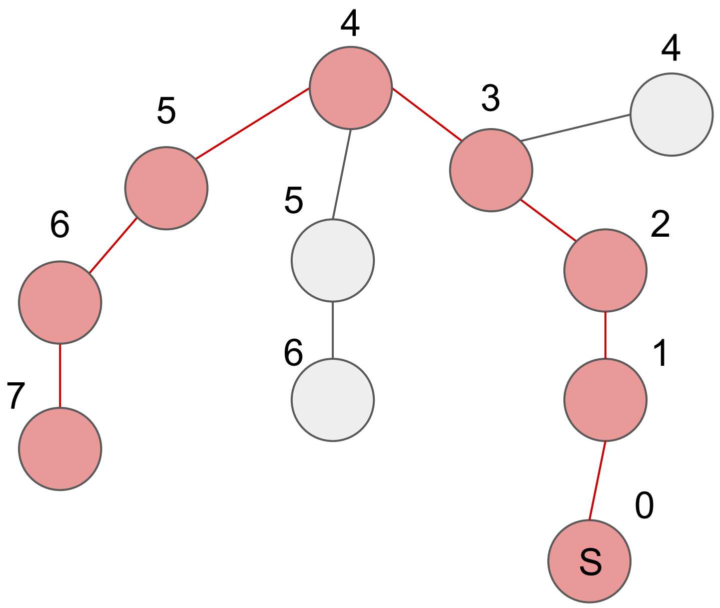
樹的問題 (樹的直徑)

- 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



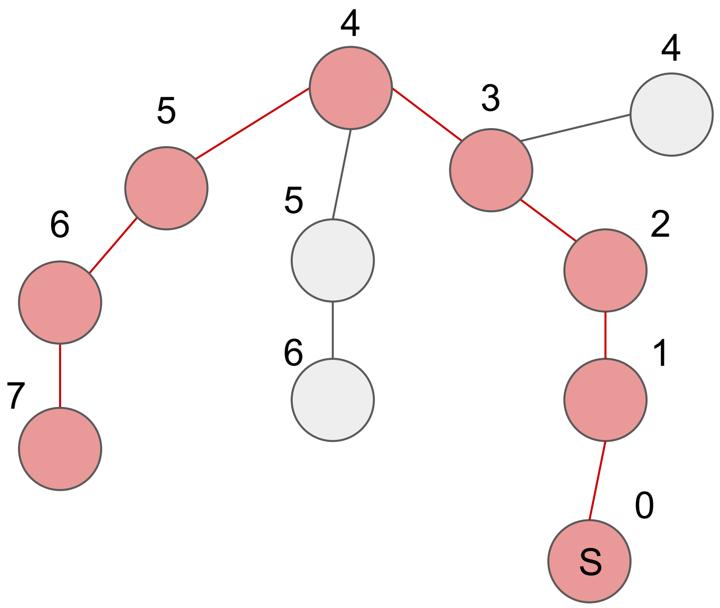
樹的問題 (樹的直徑)

- 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)

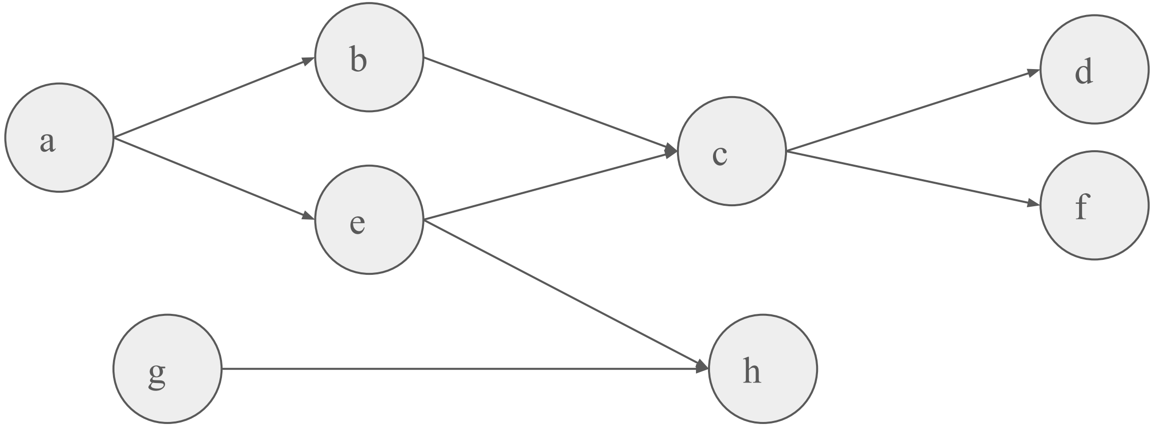


樹的問題 (樹的直徑) Code

```
const int MAXN = 1e3 + 5;
vector<int> Graph[MAXN];
int level[MAXN];
void init();                                          // 銜接bfs的code
void bfs(int s);                                      // 銜接bfs的code
int getDiameter() {
    init(); bfs(0);                                   // 初始化並bfs隨便一個點
    int maxLevel = -1, maxIndex = -1;                 // 找到最大的level值和他的index
    for (int i = 0 ; i < MAXN ; i++) {
        if (maxLevel < level[i]) {
            maxLevel = level[i];
            maxIndex = i;
        }
    }
    init(); bfs(maxIndex);                            // 初始化並從maxIndex開始bfs
    maxLevel = -1, maxIndex = -1;                     // 找到最大的level值 即為直徑
    for (int i = 0 ; i < MAXN ; i++) {
        if (maxLevel < level[i]) {
            maxLevel = level[i];
            maxIndex = i;
        }
    }
    return maxLevel;
}
```

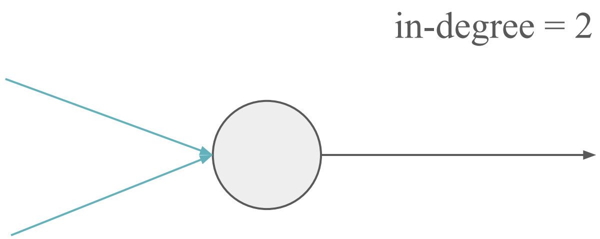
有向無環圖 DAG

- Directed Acyclic Graph



BFS 拔拔樂

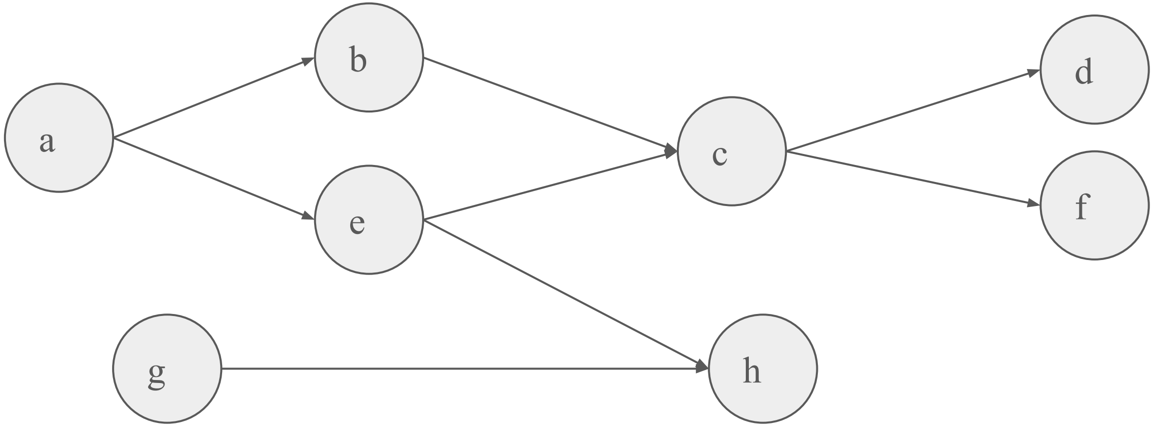
- 每次摘掉 1 個 in-degree 為 0 的起點
- 將它鄰居的 in-degree 都減 1
- 如果遇到減完後 in-degree 變成 0 的就丟進 queue 裡面



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

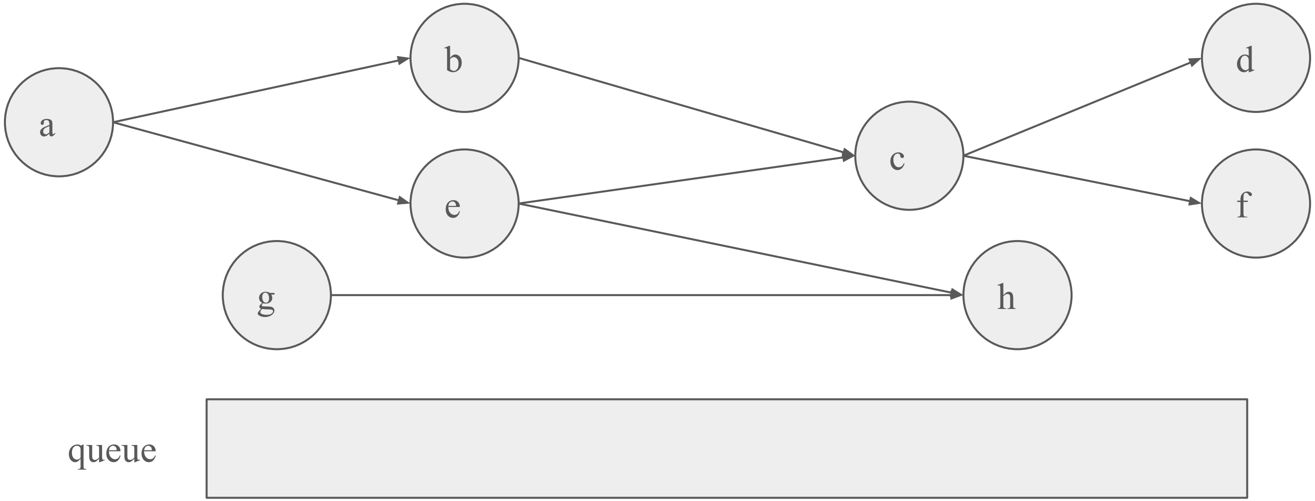
a b e g h c d f



BFS 拔拔樂

topo-sort	

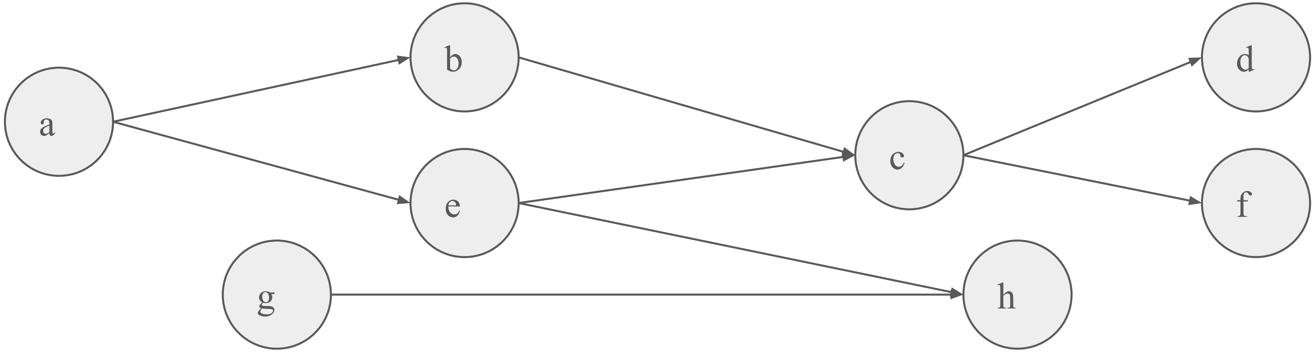
node	a	b	c	d	e	f	g	h
in-degree	0	1	2	1	1	1	0	2



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

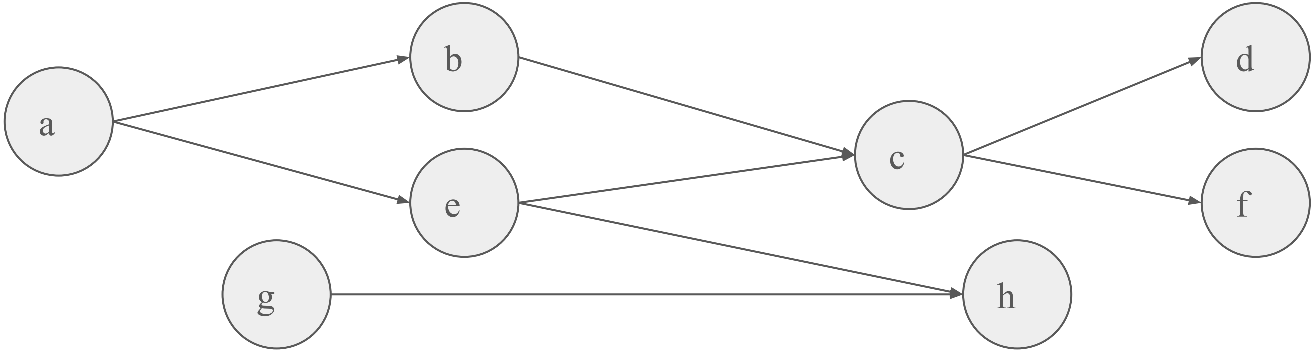


queue	a

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

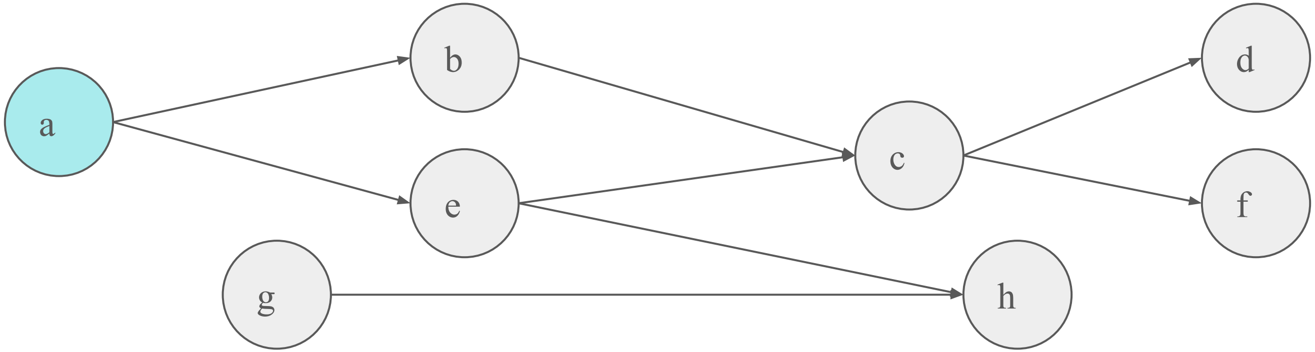


queue	a	g	

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

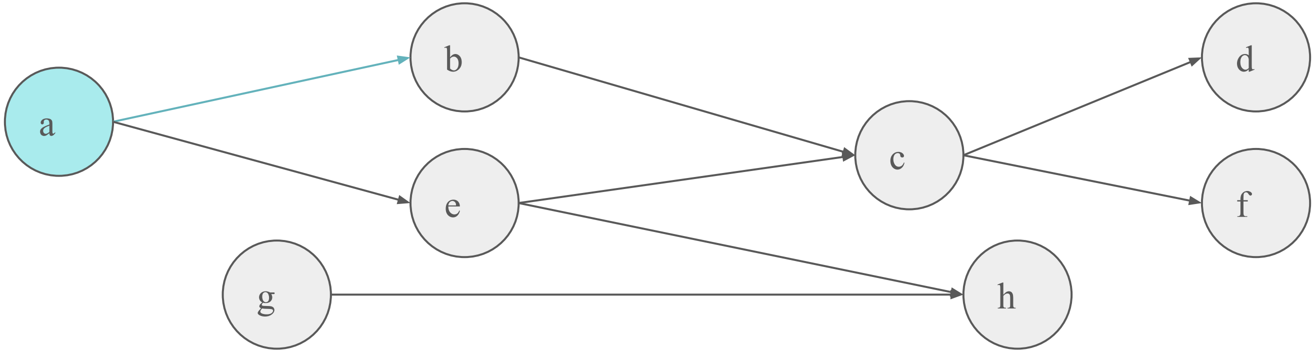


queue	a	g

BFS 拔拔樂

topo-sort	a

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



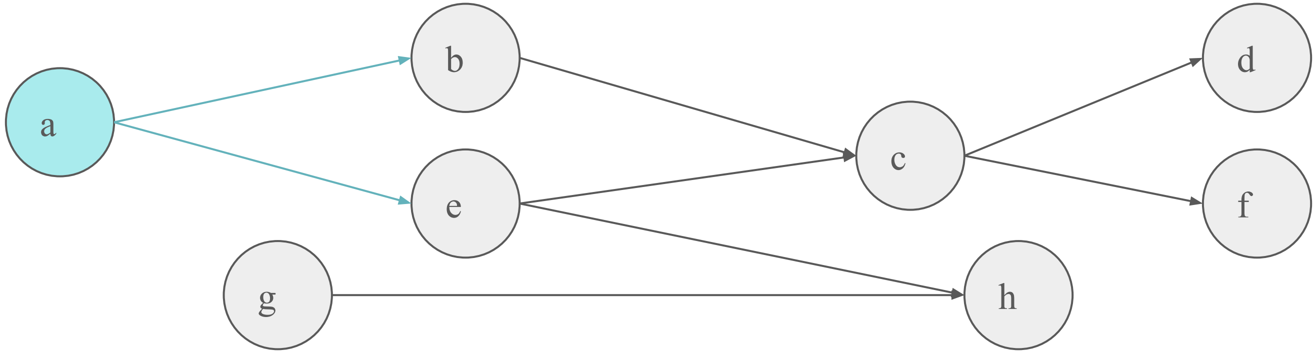
queue

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

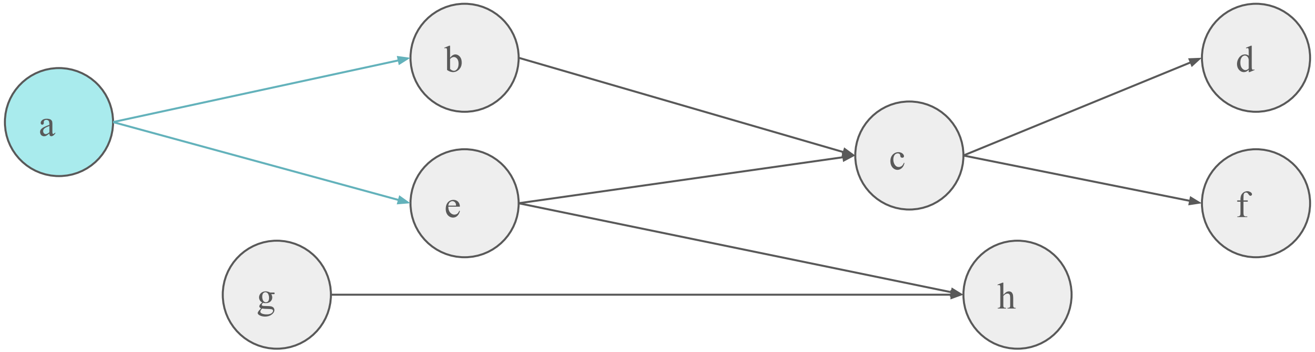


queue	g	b

BFS 拔拔樂

topo-sort	a

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

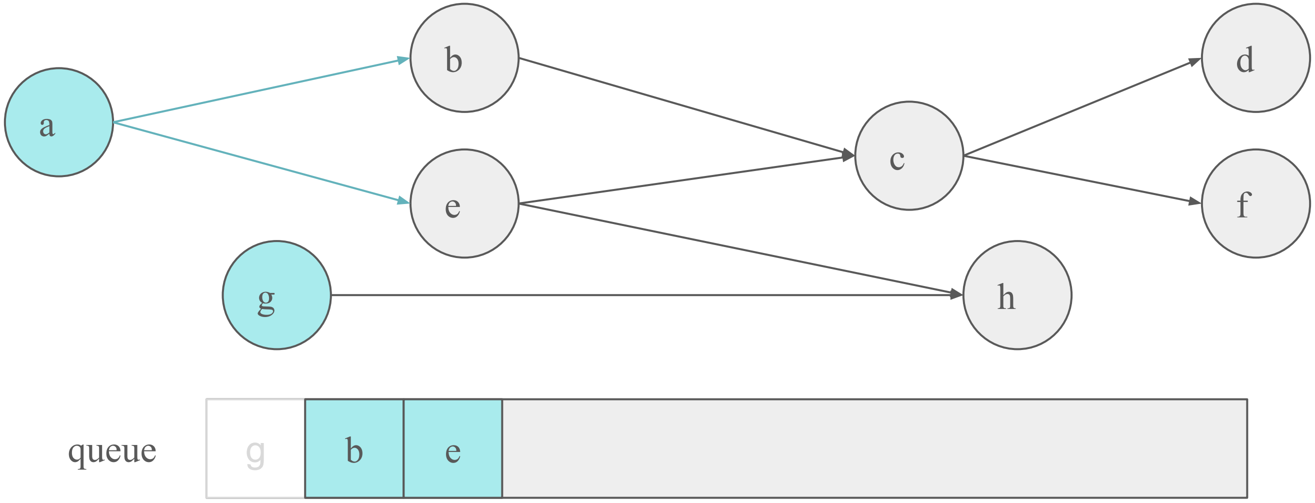


queue	g	b	e	

BFS 拔拔樂

topo-sort	a g

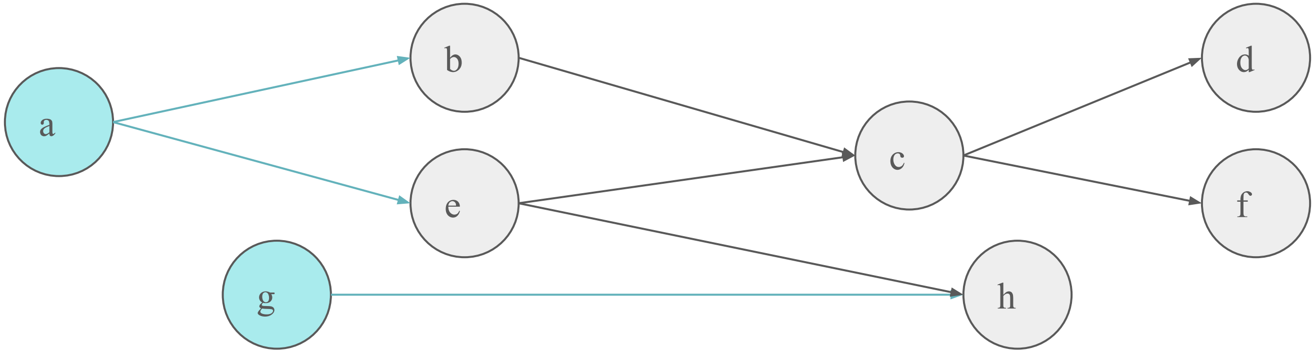
node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	0	1	0	2



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	2

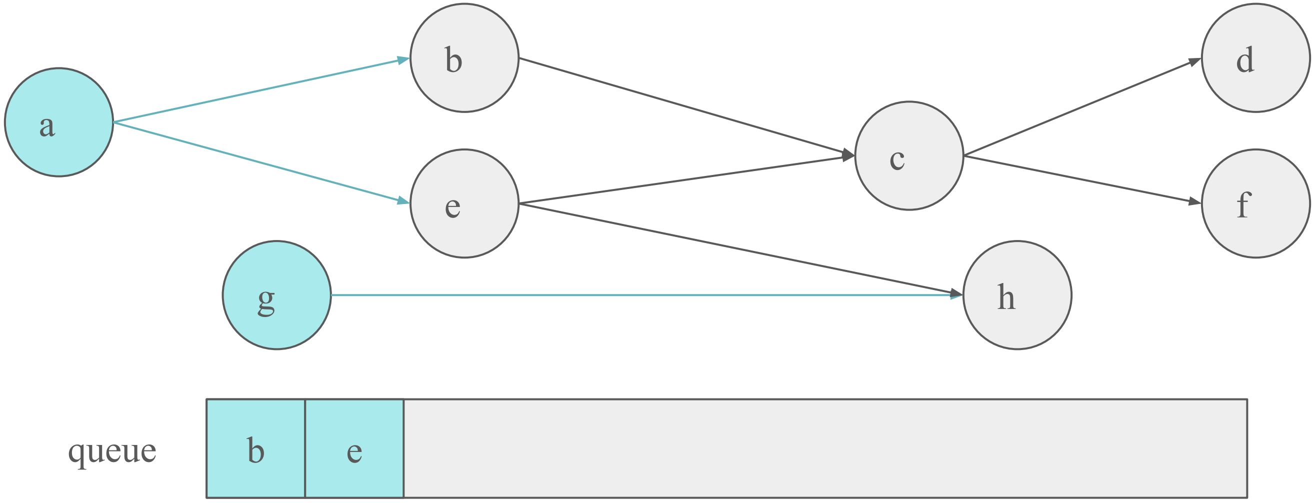


queue	b	e

BFS 拔拔樂

topo-sort	a g

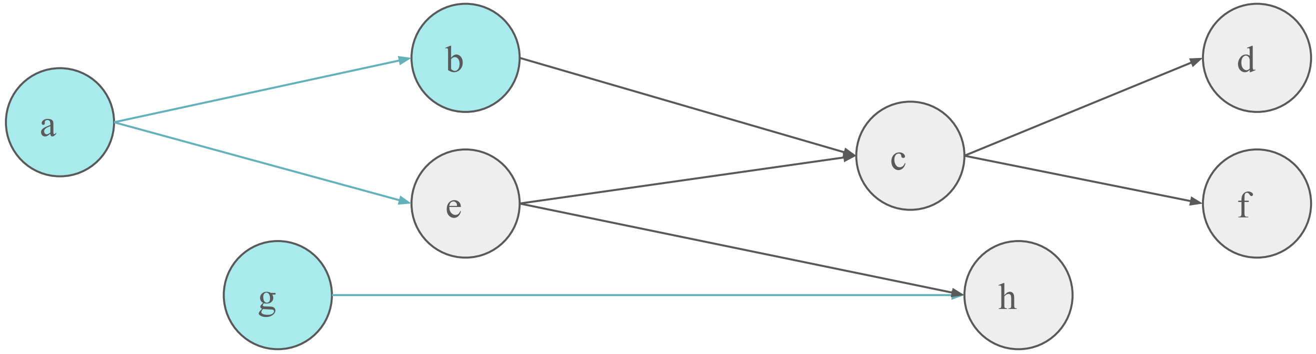
node	a	b	c	d	e	f	g	h
in-degree	0	0	2	1	0	1	0	1



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

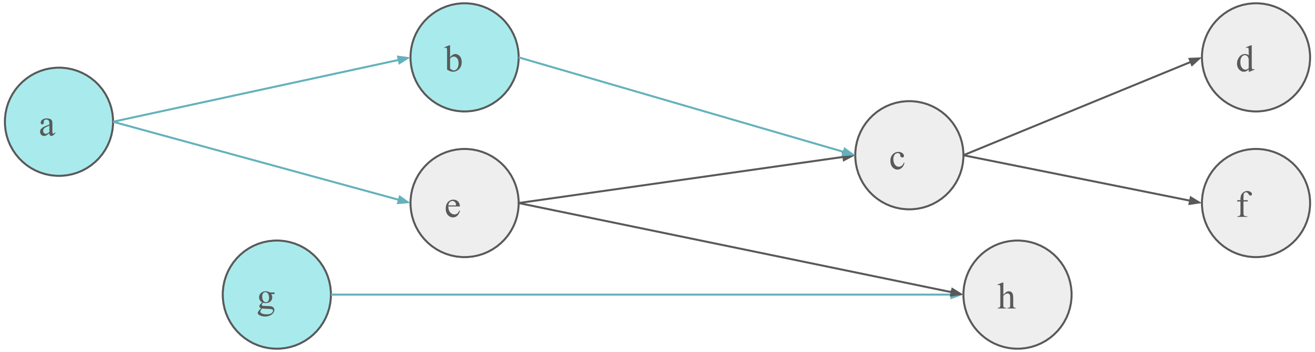


queue	b	e

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

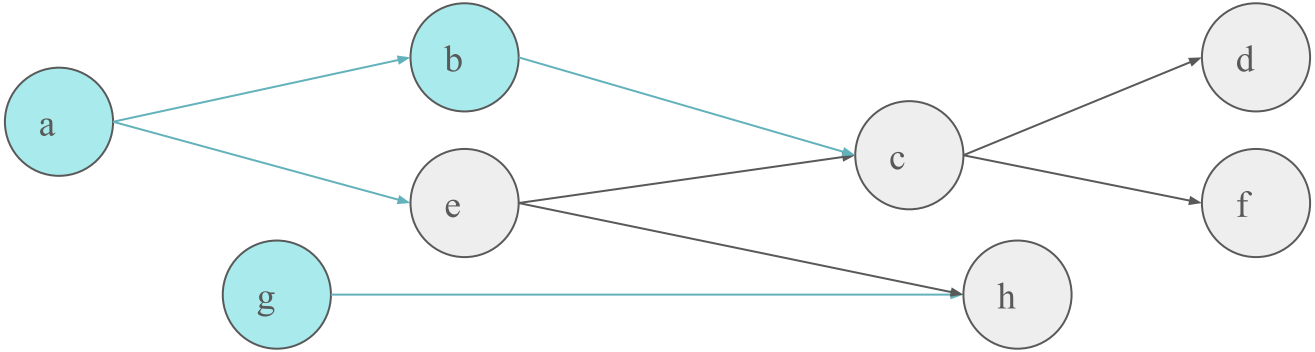


queue

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

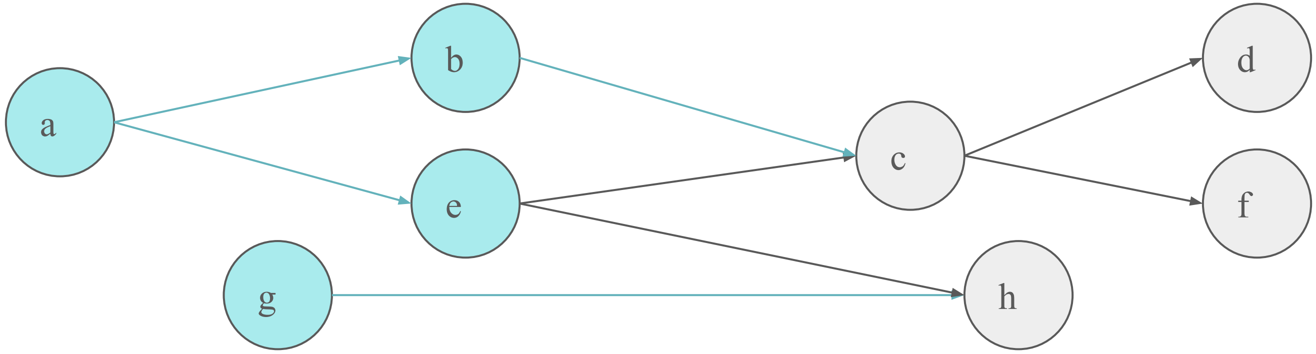


queue

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



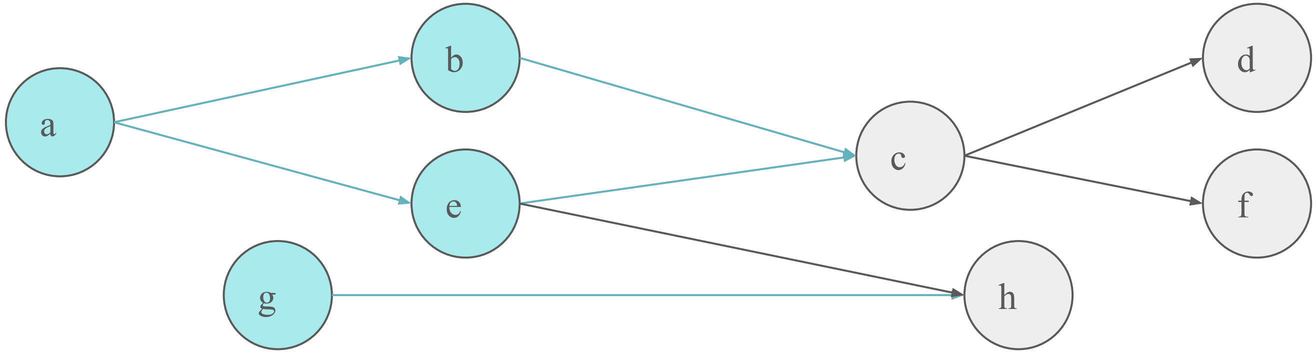
queue

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

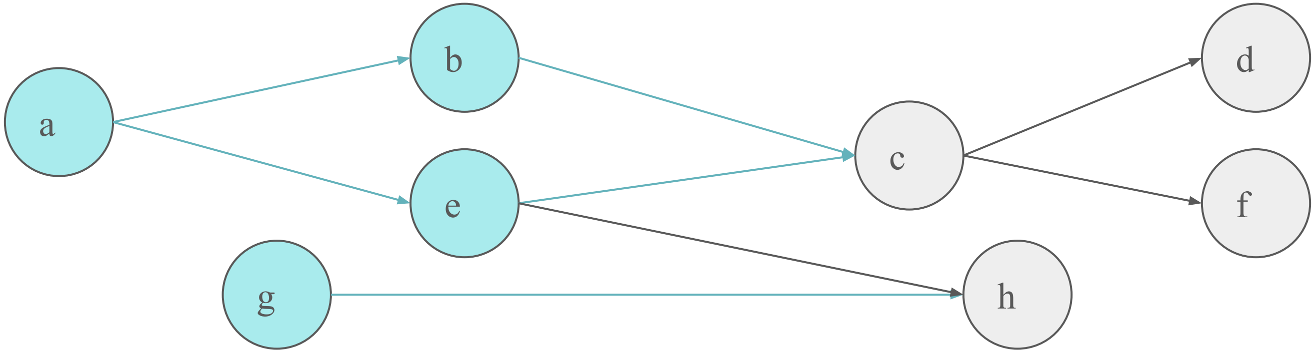


queue

BFS 拔拔樂

topo-sort	a g b e

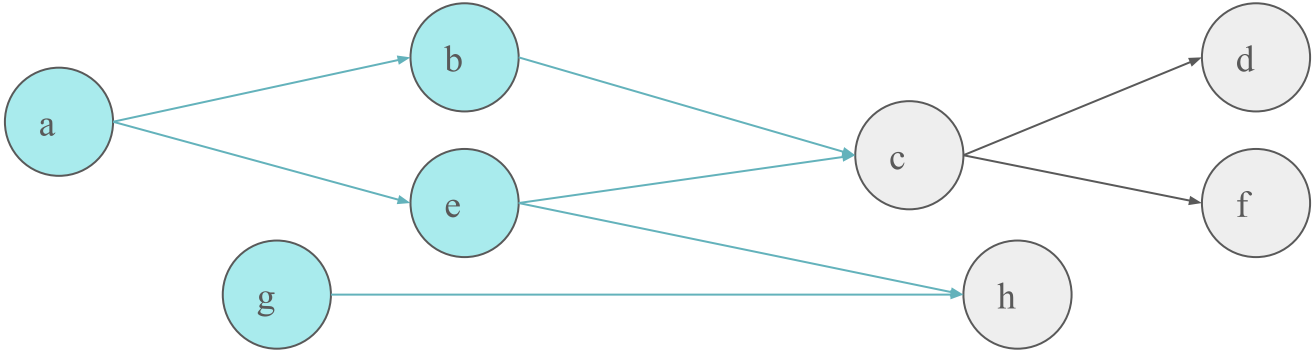
node	a	b	c	d	e	f	g	h
in-degree	0	0	0	1	0	1	0	1



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



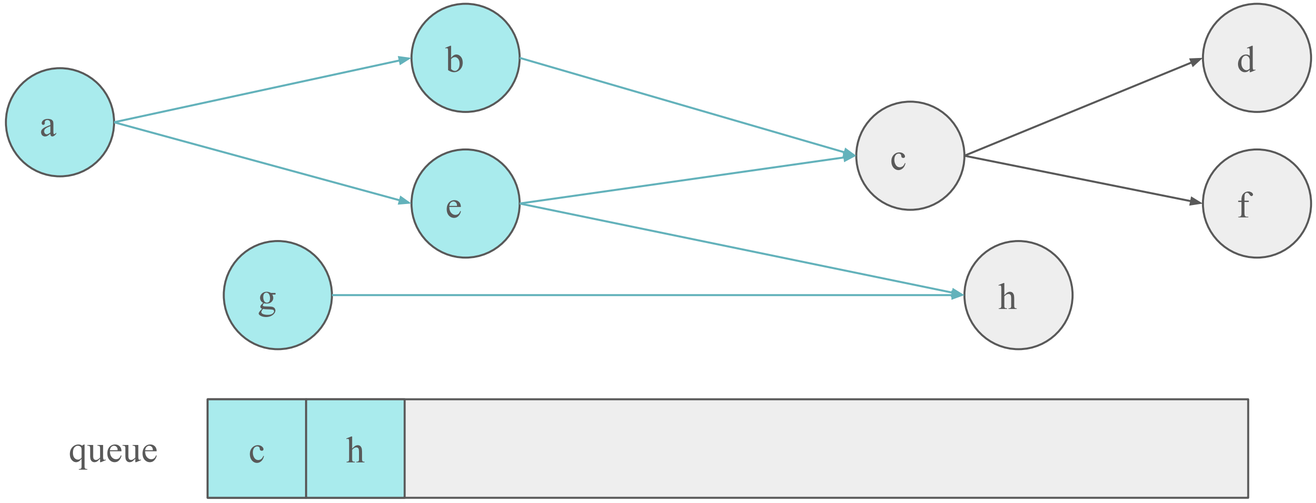
queue

c	

BFS 拔拔樂

topo-sort	a g b e

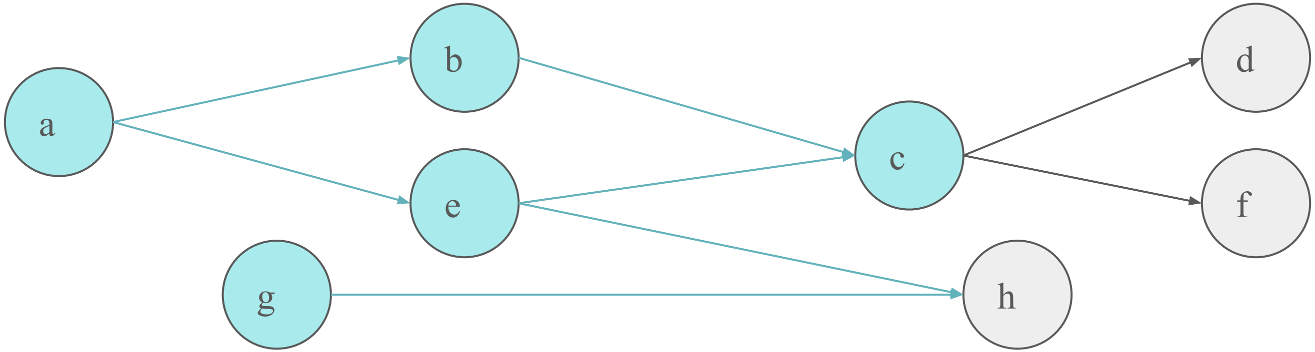
node	a	b	c	d	e	f	g	h
in-degree	0	0	0	1	0	1	0	0



BFS 拔拔樂

topo-sort	a g b e c

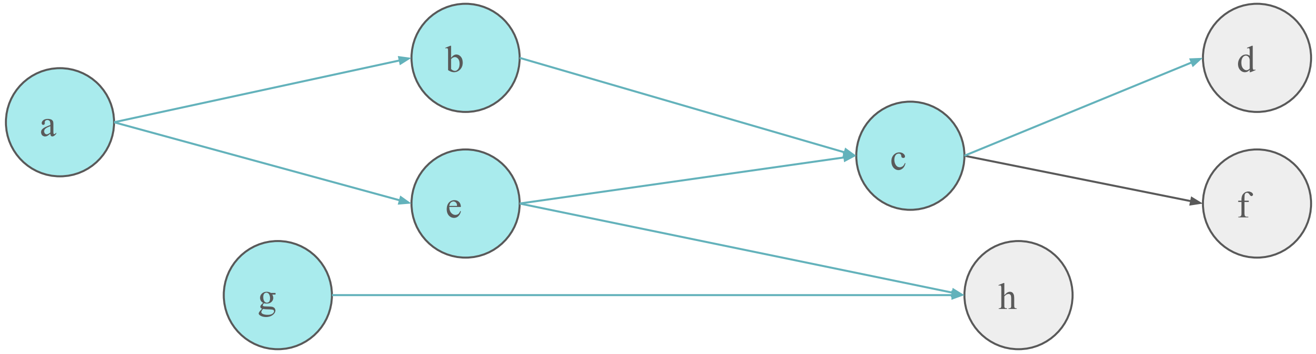
node	a	b	c	d	e	f	g	h
in-degree	0	0	0	1	0	1	0	0



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

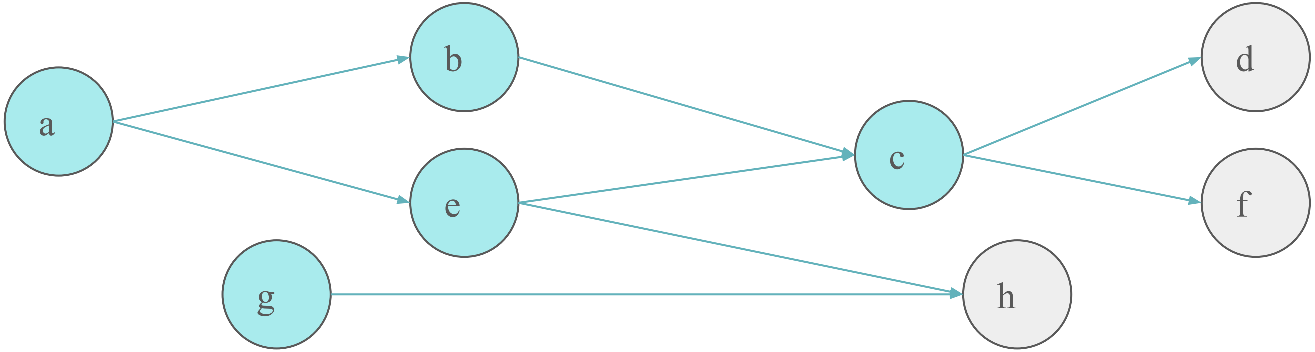


queue	h	d	

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



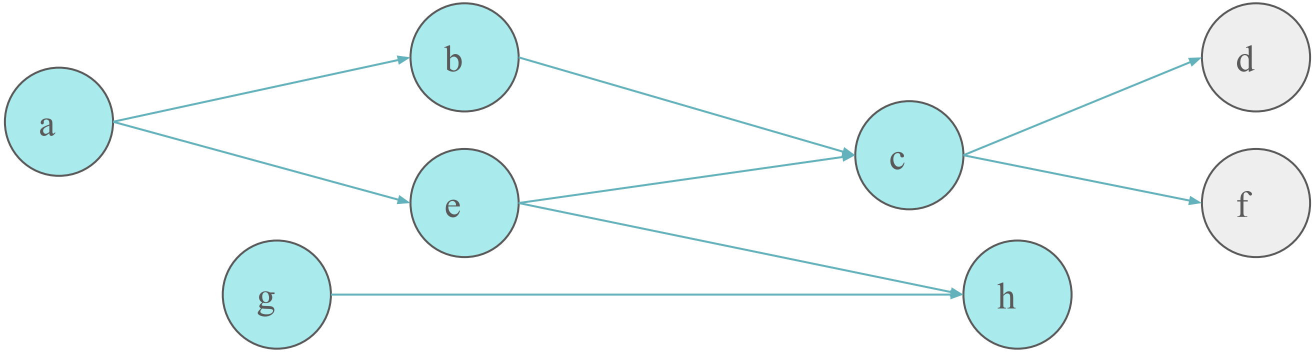
queue

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

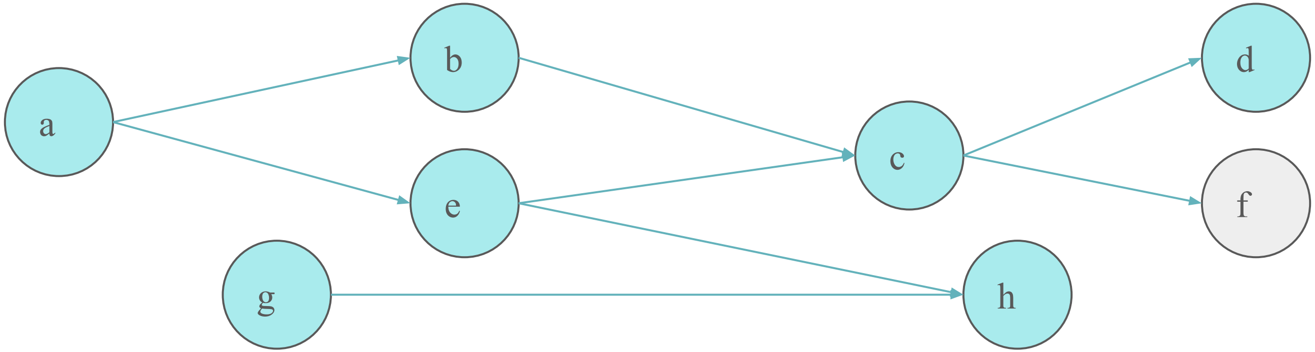


queue	h	d	f	

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

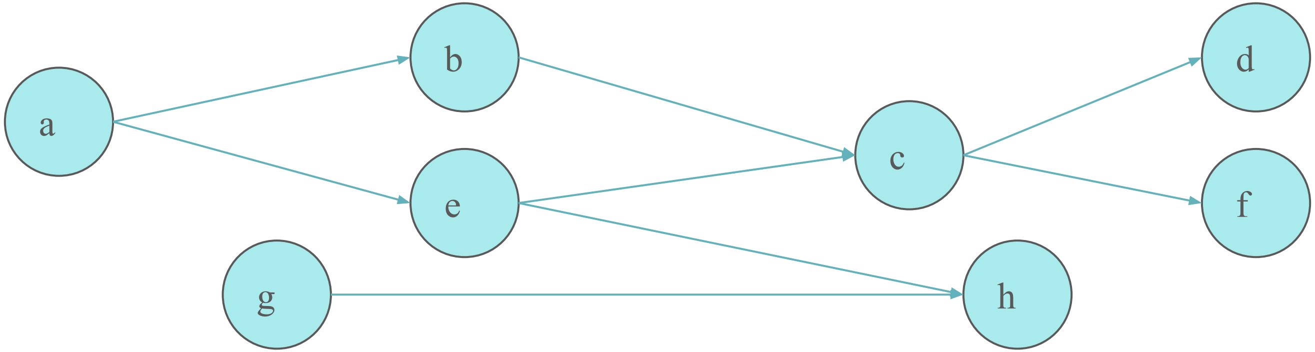


queue	d	f

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

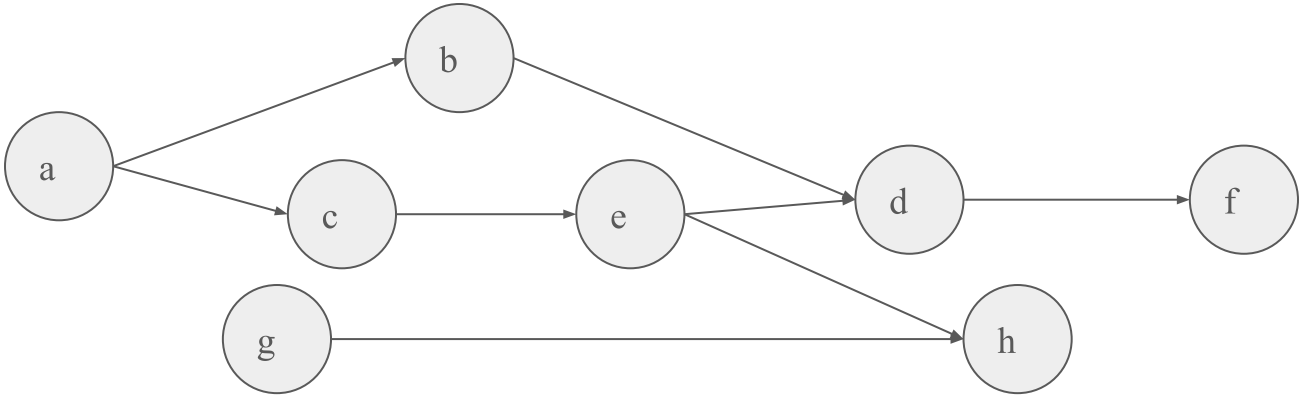
```cpp
const int MAXN = 1e3 + 5;
int n;                                  // 有幾個節點
vector<int> G[MAXN];
int indegree[MAXN];                     // 存放indegree
void init() {
    memset(indegree, 0, sizeof(indegree));
}
void addEdge(int u, int v) {
    G[u].push_back(v);
    indegree[v]++;              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

BFS 拔拔樂 Code

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

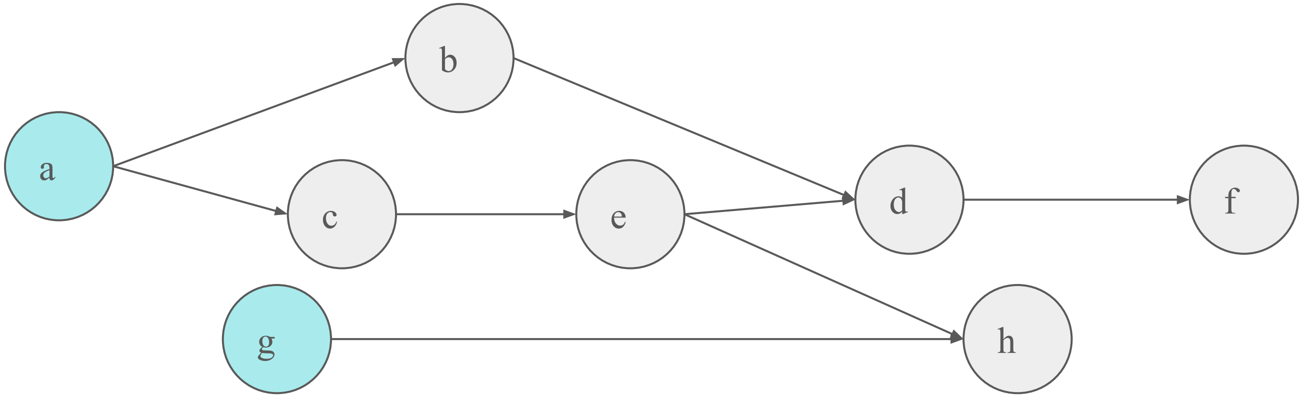
DAG 最長路

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



DAG 最長路

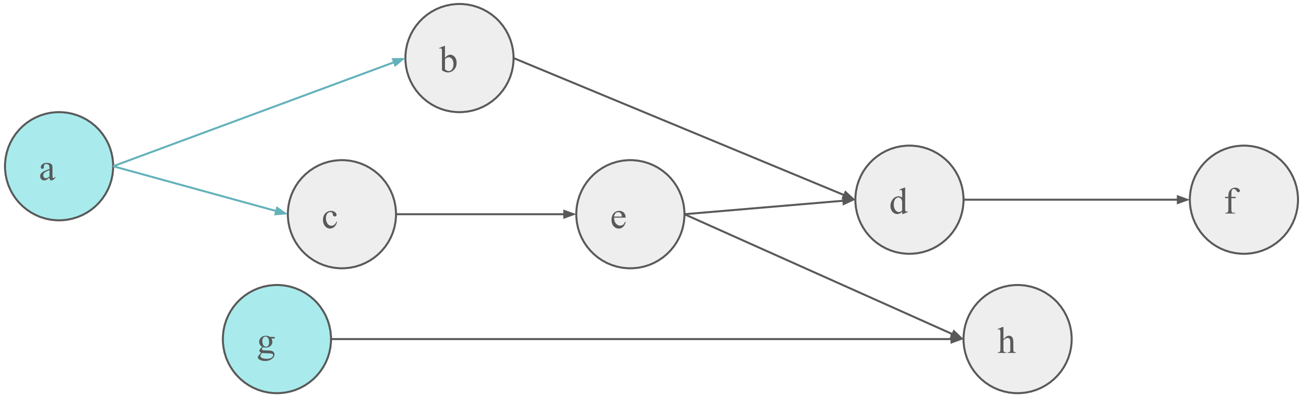
node	a	b	c	d	e	f	g	h
dist	0	-1	-1	-1	-1	-1	0	-1



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

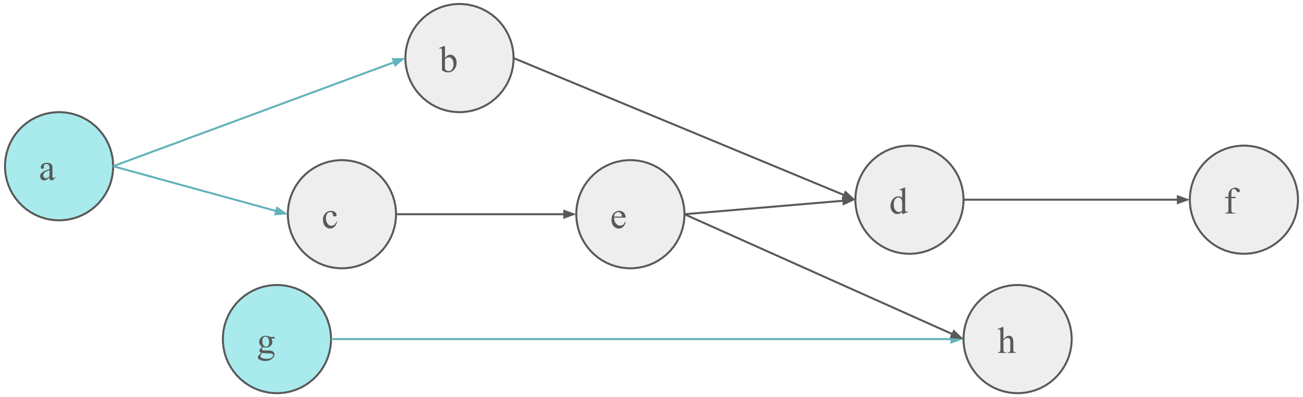


DAG 最長路

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

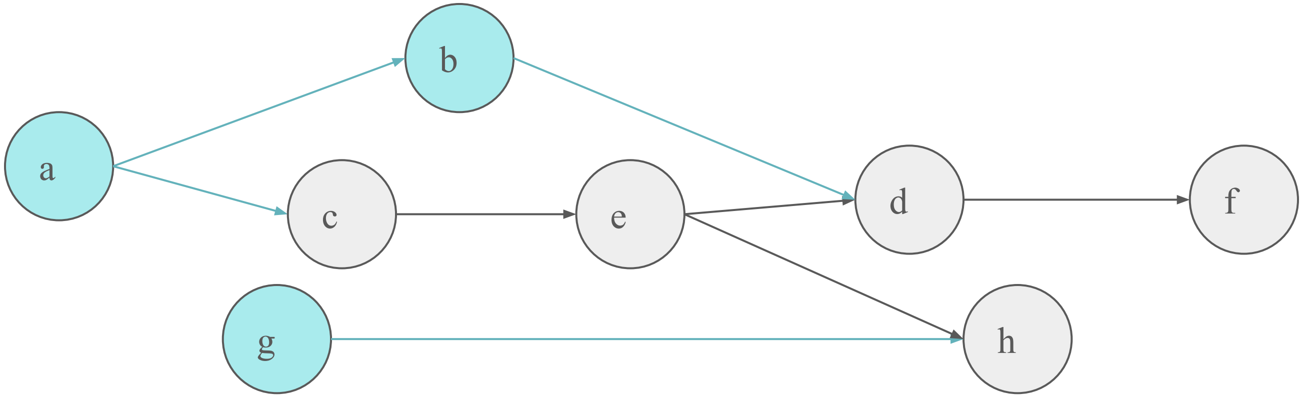


DAG 最長路

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

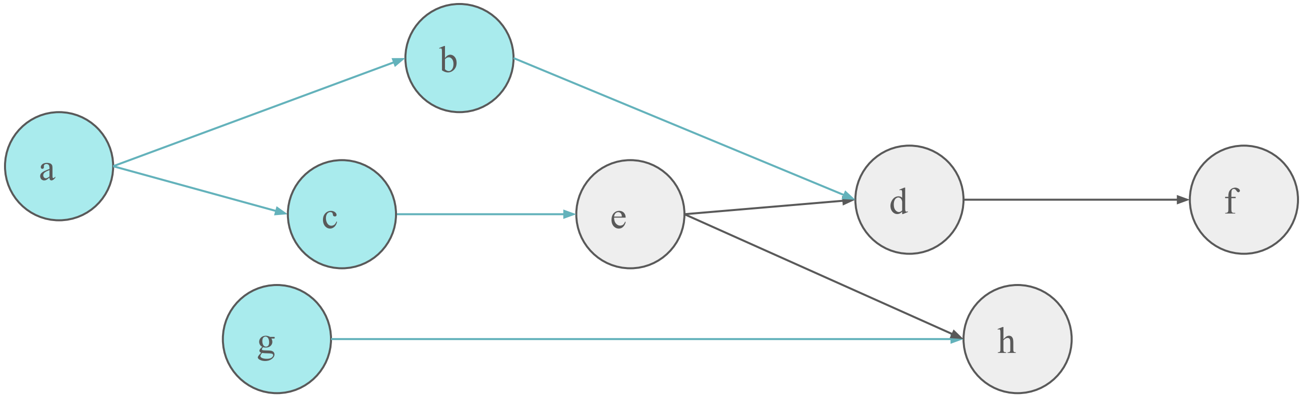


DAG 最長路

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

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

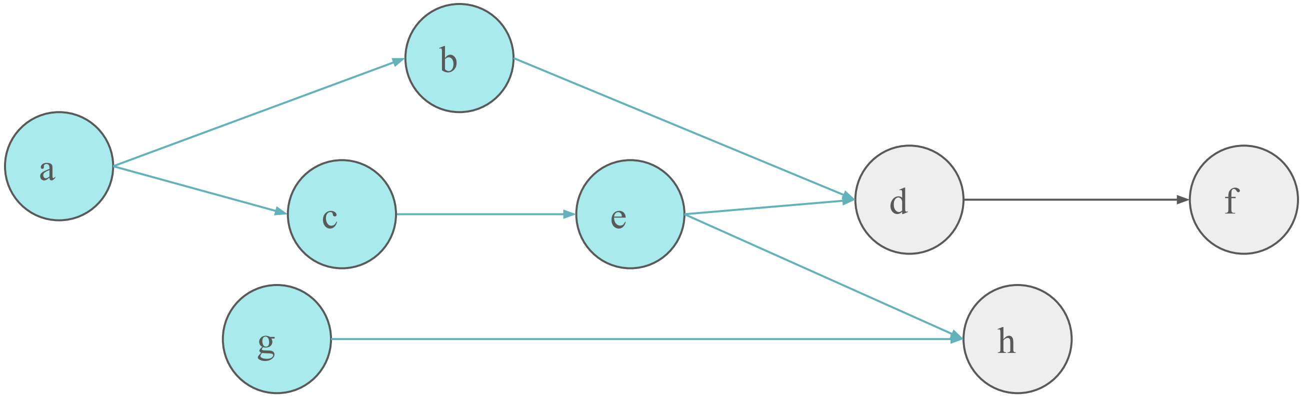
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	-1	0	1



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

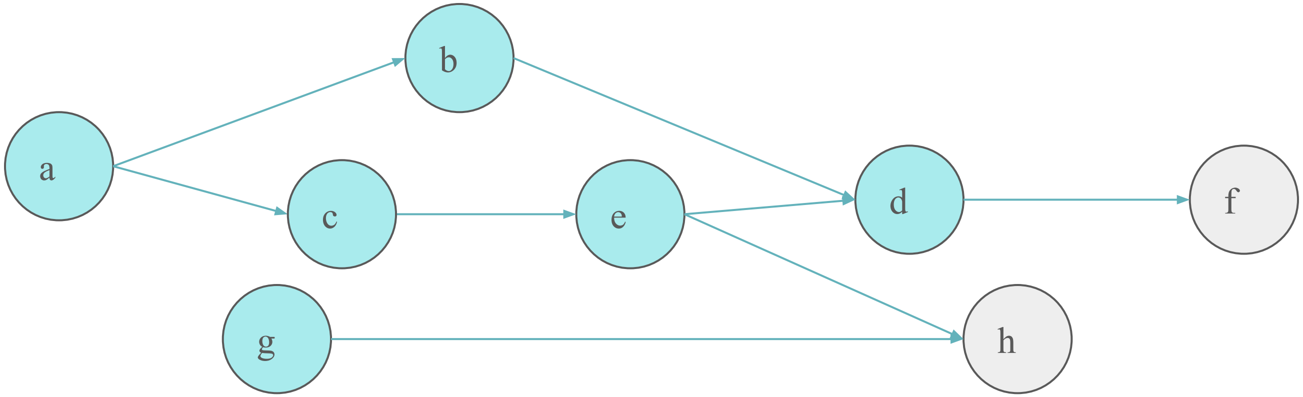


DAG 最長路

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

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

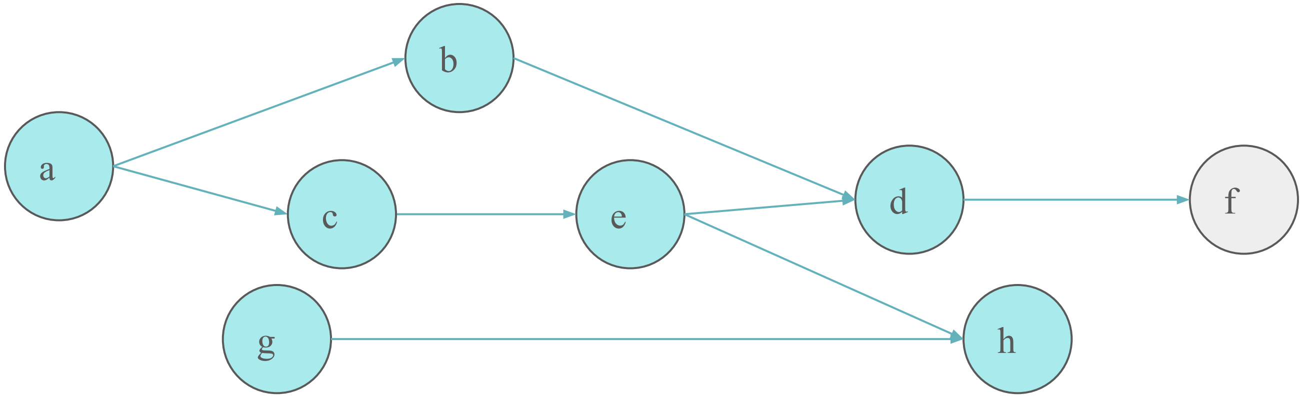
node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	4	0	3



DAG 最長路

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

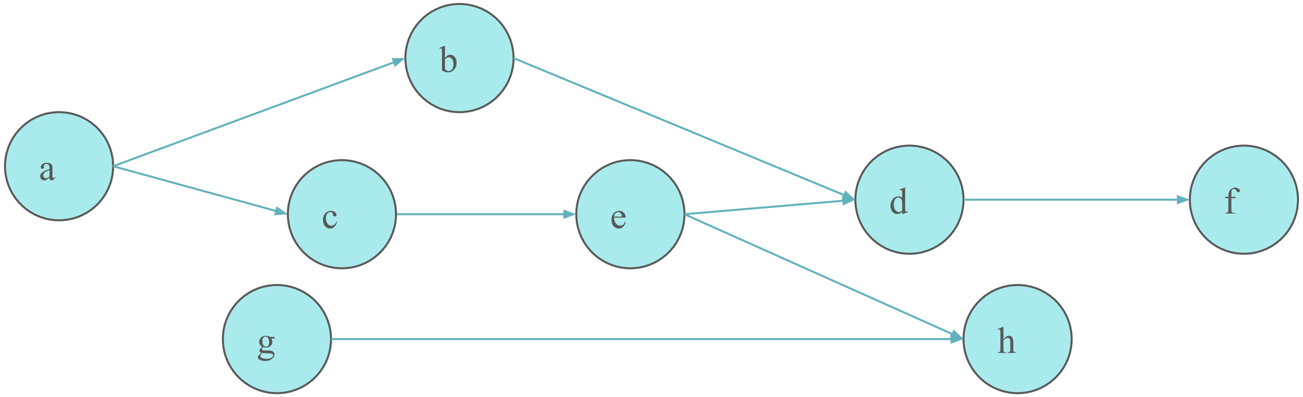
node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	4	0	3



DAG 最長路

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

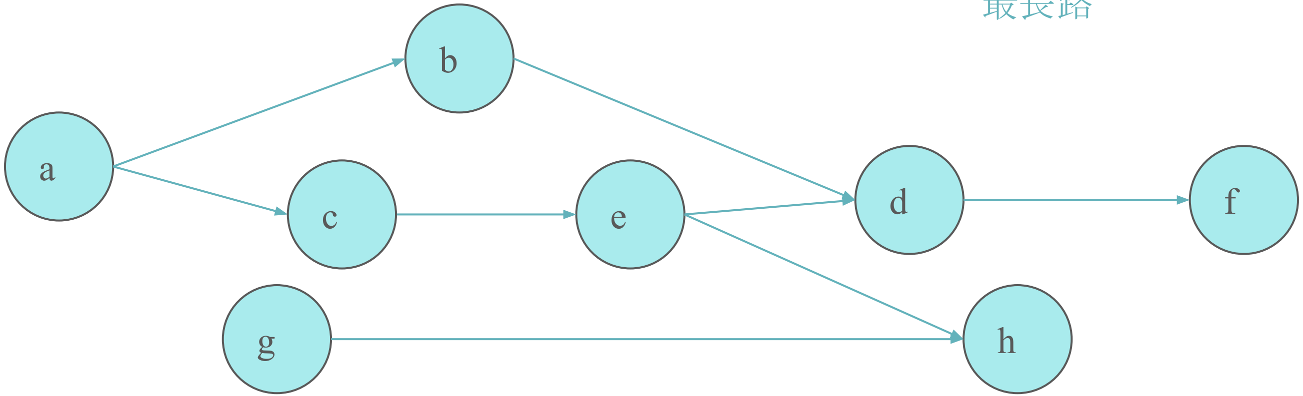
node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	4	0	3



DAG 最長路

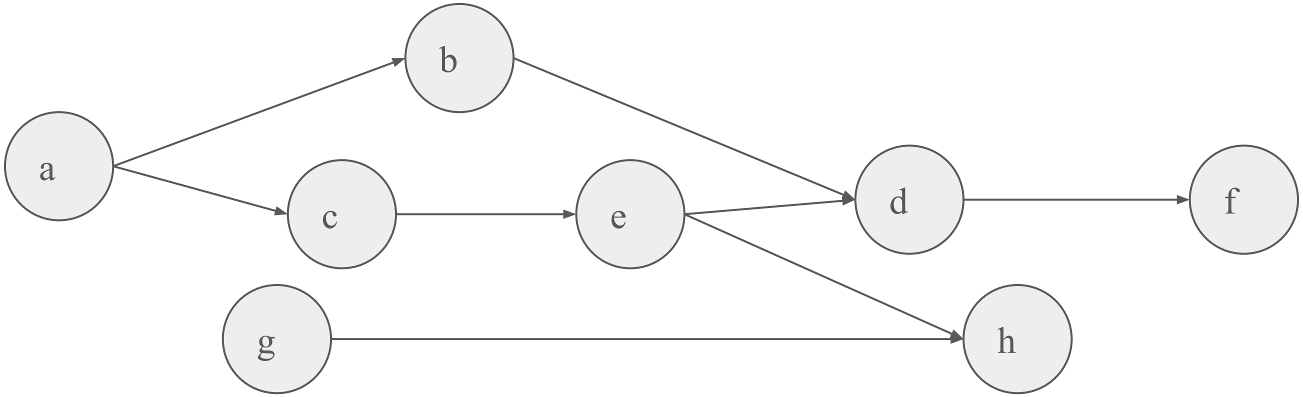
node	a	b	c	d	e	f	g	h
dist	0	1	1	3	2	4	0	3

最長路



DAG 最短路

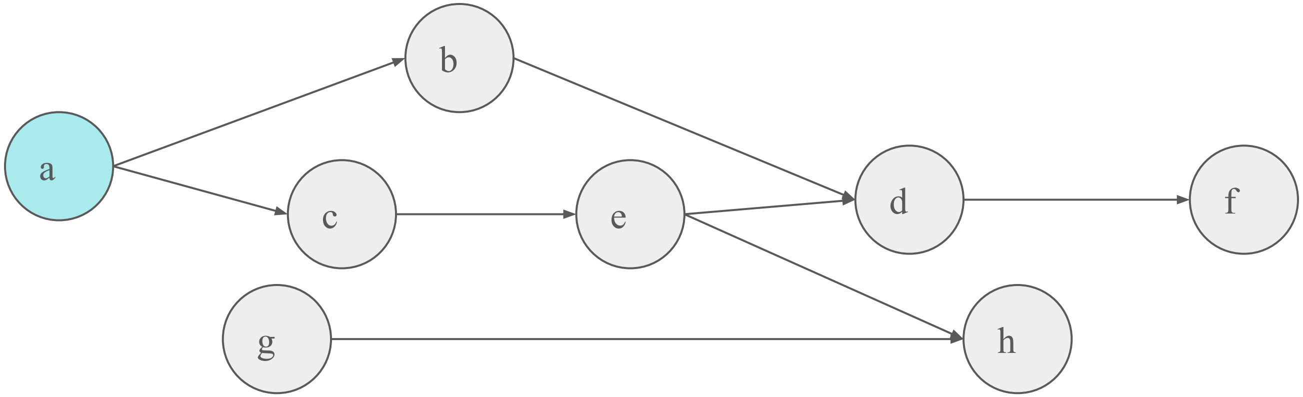
node	a	b	c	d	e	f	g	h
dist	INF	INF	INF	INF	INF	INF	INF	INF



DAG 最短路

選定一個起點 s
dist[s] = 0
dist[a] = 0

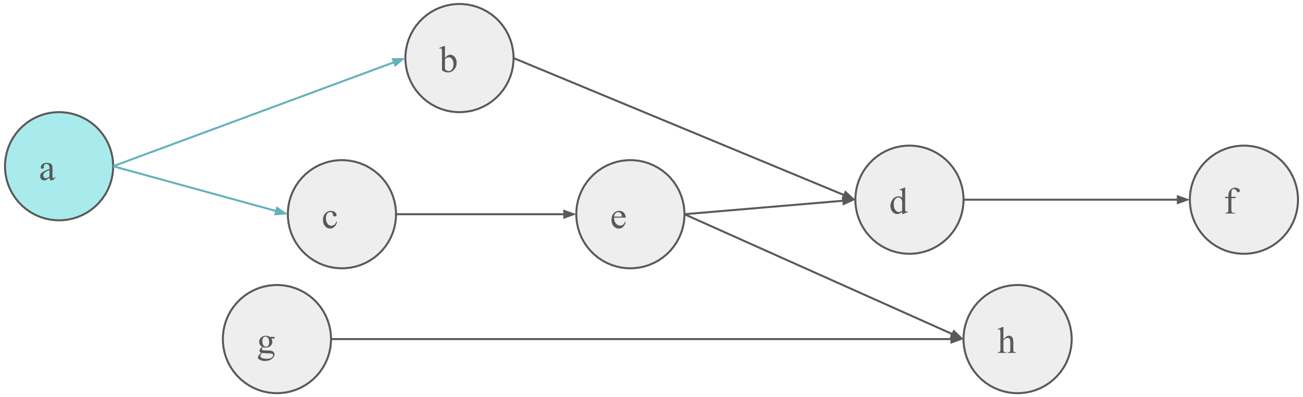
node	a	b	c	d	e	f	g	h
dist	0	INF	INF	INF	INF	INF	INF	INF



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

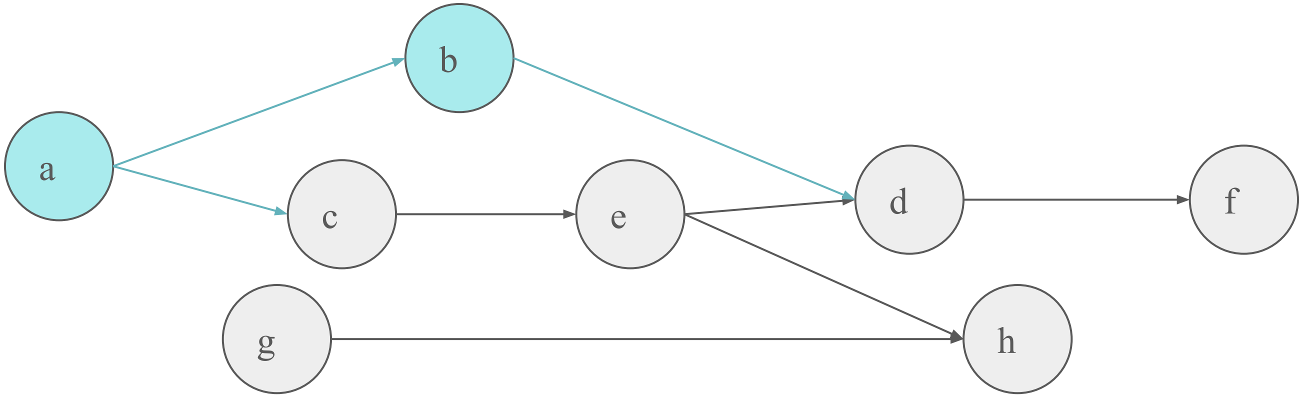


DAG 最短路

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

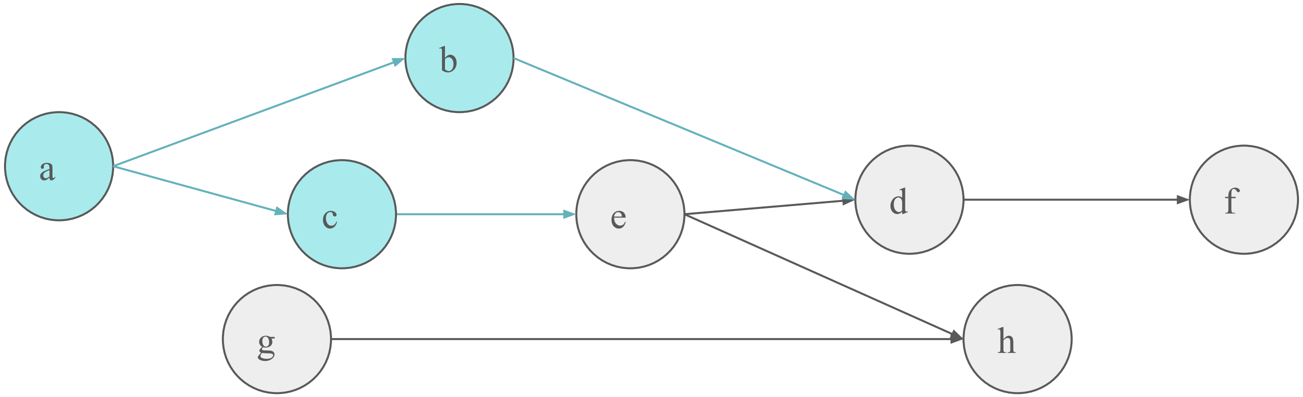


DAG 最短路

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

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

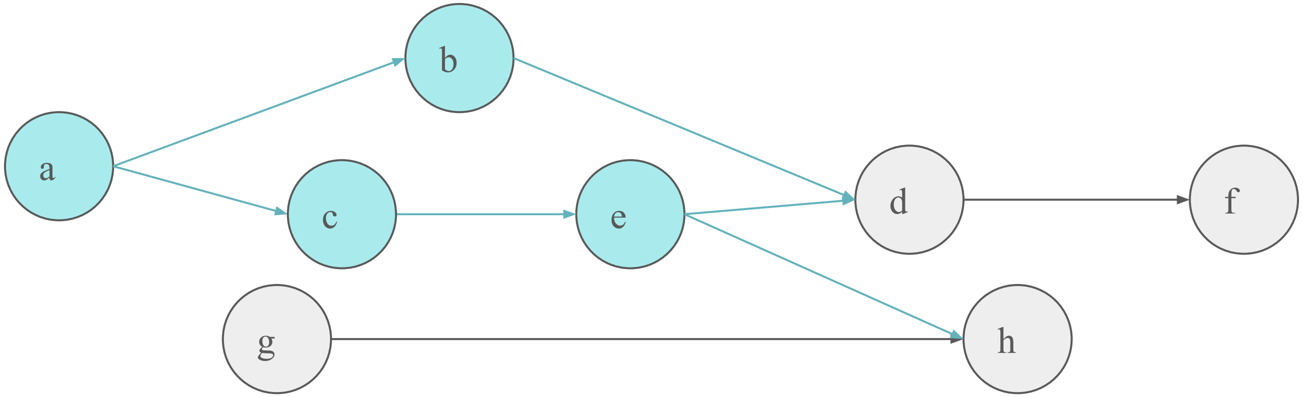
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	INF	INF	INF



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

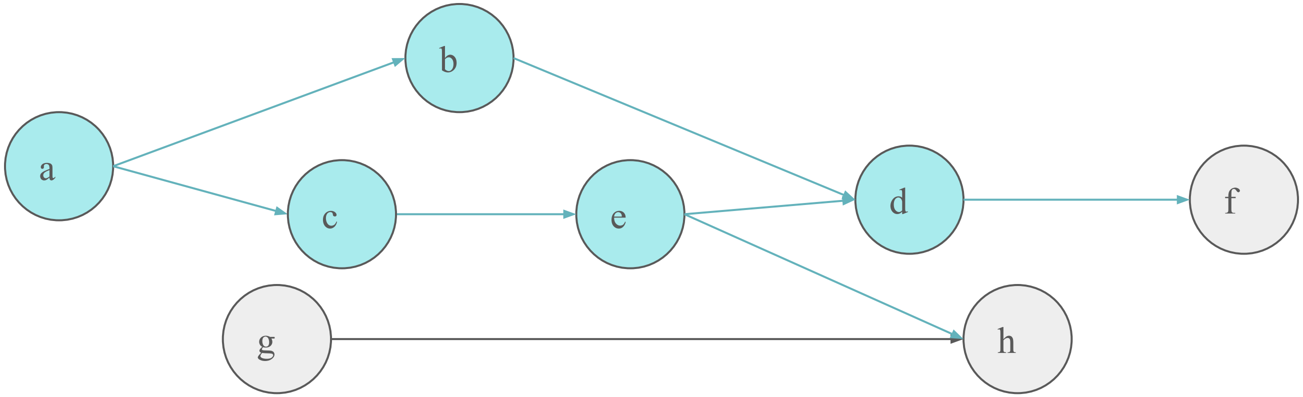


DAG 最短路

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

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

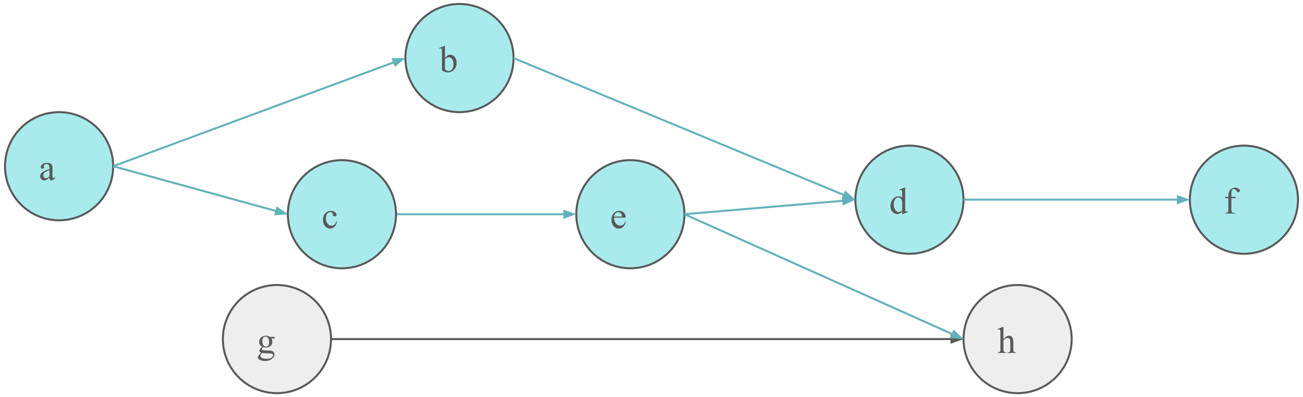
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	3	INF	3



DAG 最短路

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

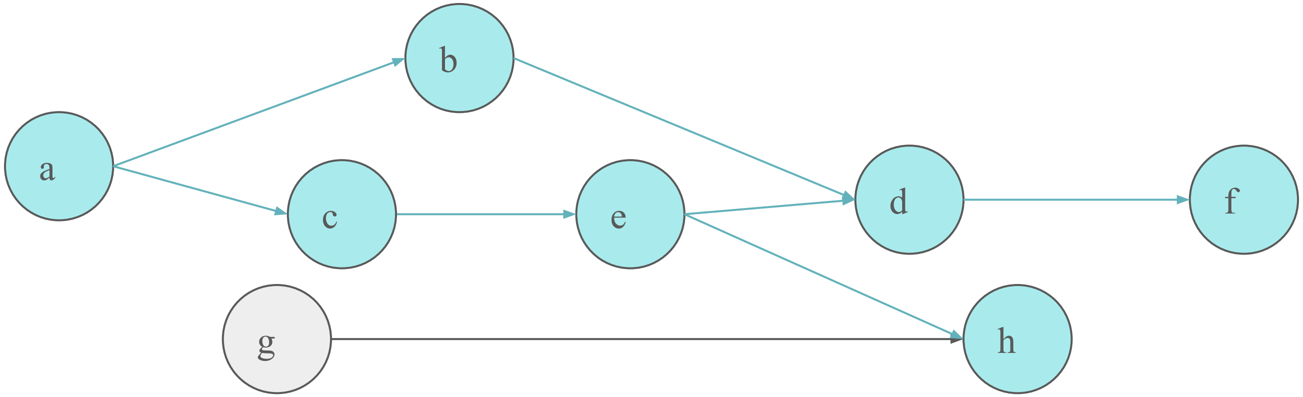
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	3	INF	3



DAG 最短路

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

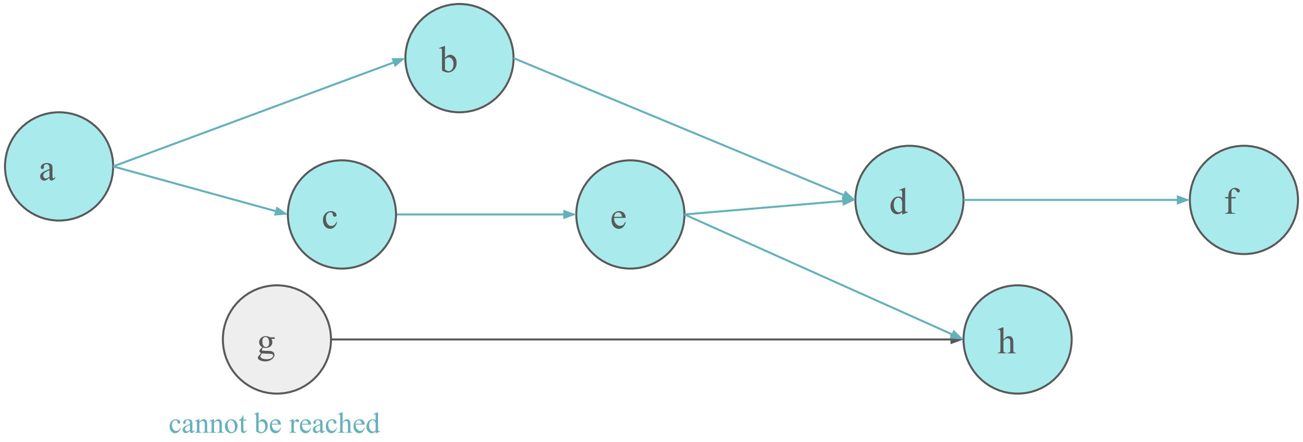
node	a	b	c	d	e	f	g	h
dist	0	1	1	2	2	3	INF	3



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



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

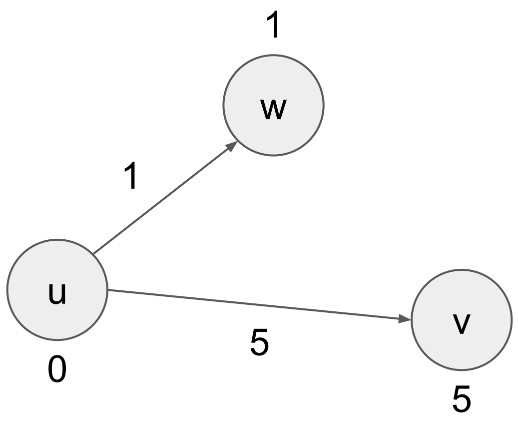
- 一般圖多出了環，還可以一次拔拔樂搞定嗎？
    - 如果拔到剩環，就不會有 in degree是0的點可以拔，所以沒辦法

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

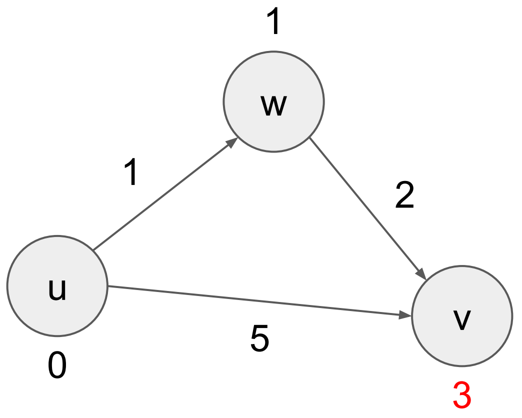
一般圖正權重的最短路

- 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



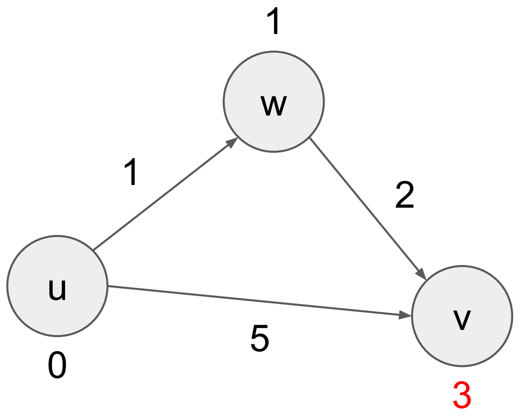
一般圖正權重的最短路

- 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

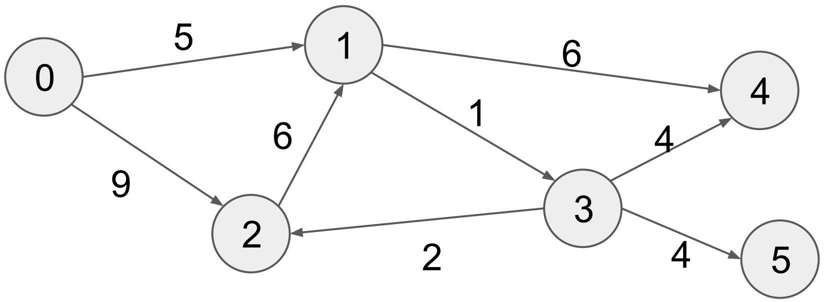
- 一般圖多出了環，還可以一次拔拔樂搞定嗎？
  - 如果拔到剩環，就不會有 in degree是0的點可以拔，所以沒辦法
- relaxation操作
  - 原本已知的最短路為右圖
  - 找到一條更短的路，讓當前最短路更短了
    - dis[i]為從原點到 i 的當前最短路徑
    - dis[v] = min(dis[v], dis[u] + cost(w, v))
- Djikstra演算法
  - 貪心性質，如果該節點距離是尚未被選取的點中最小的，那他就是最短路徑



Dijkstra

初始化dis陣列 (dis[i] := 起點到 i 的最短路徑)。

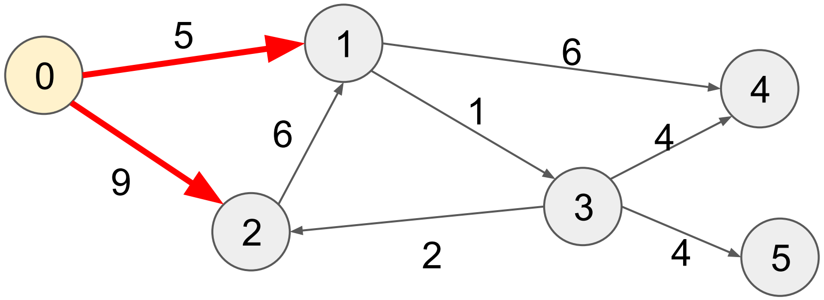
i	0	1	2	3	4	5
dis[i]	0	INF	INF	INF	INF	INF



Dijkstra

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

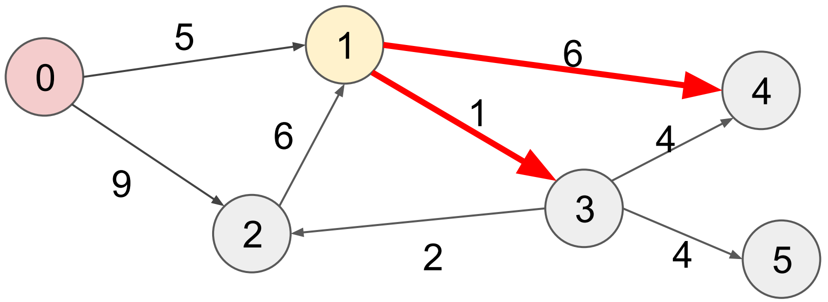
i	0	1	2	3	4	5
dis[i]	0	5	9	INF	INF	INF



Dijkstra

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

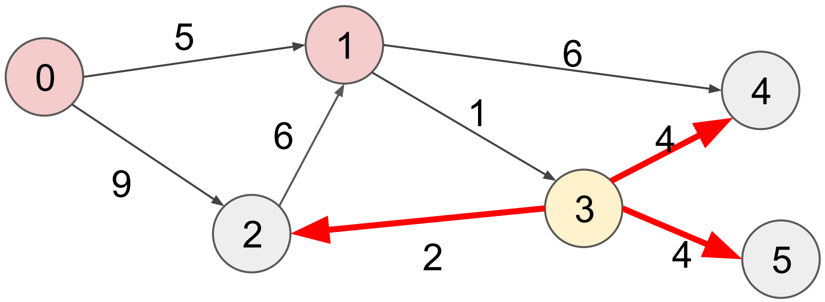
i	0	1	2	3	4	5
dis[i]	0	5	9	6	11	INF



Dijkstra

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

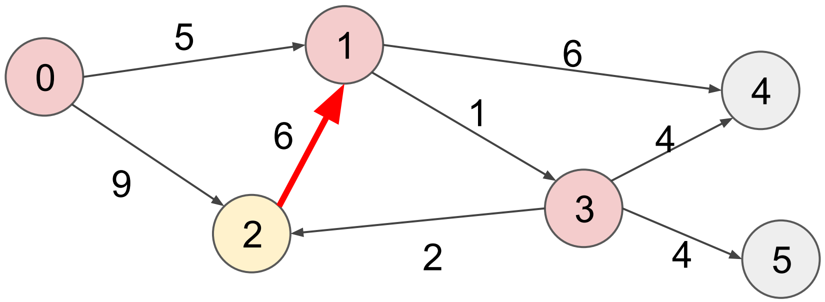
i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Dijkstra

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

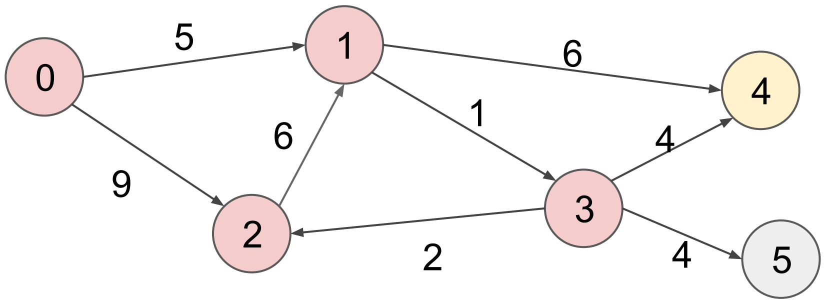
i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Dijkstra

選出最小的 dis[i] (節點4), 沒有鄰居不動作

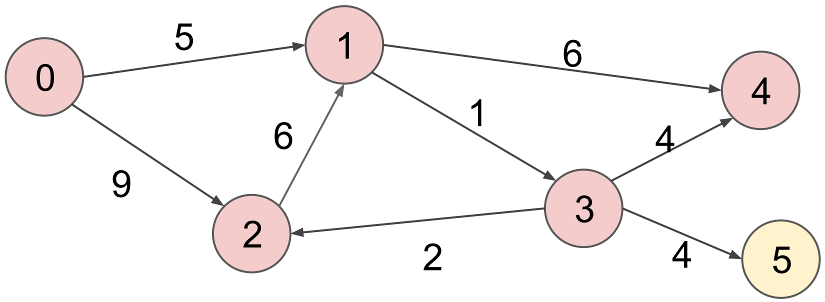
i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



Dijkstra

選出最小的 dis[i] (節點5), 沒有鄰居不動作

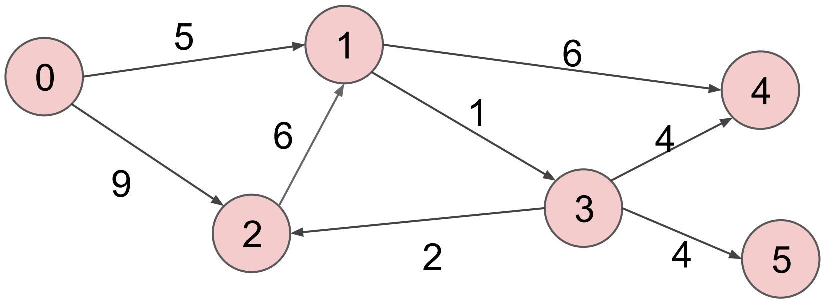
i	0	1	2	3	4	5
dis[i]	0	5	8	6	10	10



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

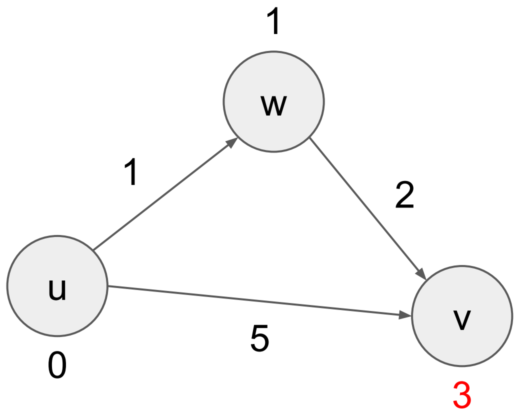
- 每次都挑選當前dis陣列中最小的節點
    - 需要一個可以支援插入新東西並排好序的
    - priority_queue
- 複雜度分析
    - 最壞的情況每個點每個邊都需要被丟進去 priority_queue
        - O((E+V)lgV)

Dijkstra Code

```
struct Edge {
    int v, w;
    bool operator < (const Edge &cmp) const {
        return cmp.w < w; //定義edge 的排序方式
    }
};
vector<Edge> Graph[maxn];
int dis[maxn];
void dijkstra(int s) {
    memset(dis, -1, sizeof(dis)); // 初始化dis陣列將值設成-1
    priority_queue<Edge> pq;
    pq.push({s, 0});
    while (pq.size()) {   //當pq 還沒有是空的話就繼續做
        auto node = pq.front(); pq.pop();// 將node 從pq pop掉
        if (dis[node.v] != -1) continue; //node.v在這之前就已經更新過了
        dis[node.v] = node.w;            //更新dis[node.v]
        for (auto k : Graph[node.v]) {       //列舉所有相鄰的邊
            if (dis[k.v] == -1) {        //relaxation 操作
                pq.push({k.v, node.w + k.w});// 將新的狀態push在pq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

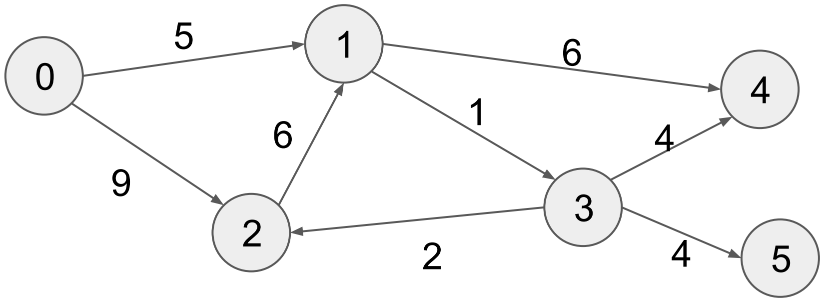


一般圖正權重的最短路

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

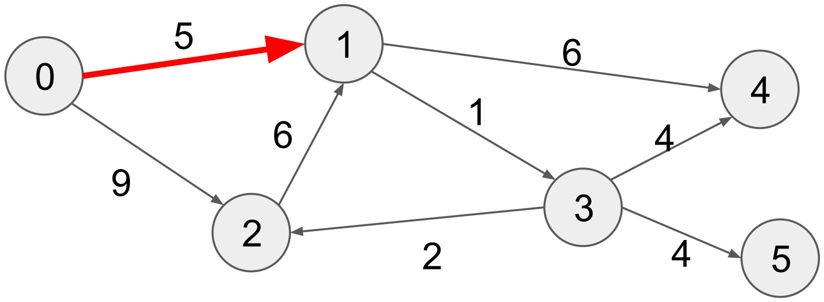


一般圖正權重的最短路

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

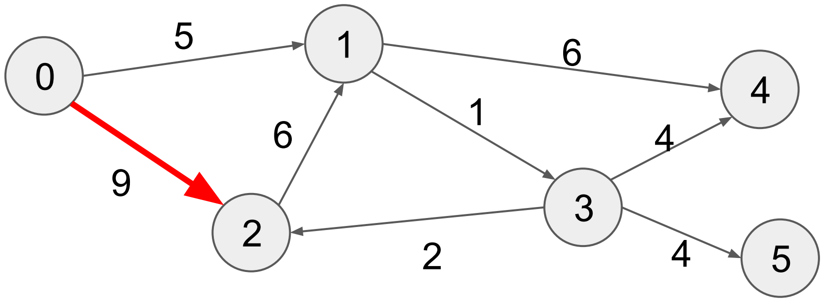


一般圖正權重的最短路

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

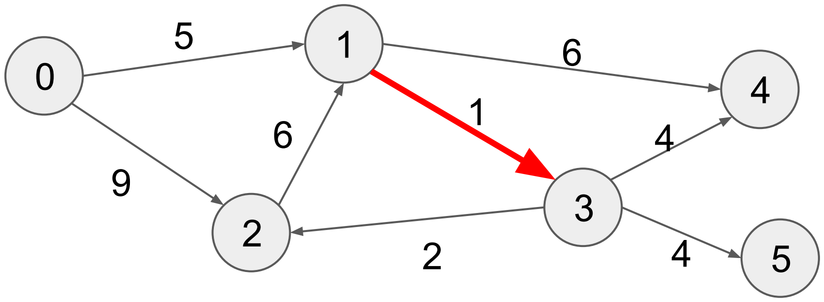


一般圖正權重的最短路

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

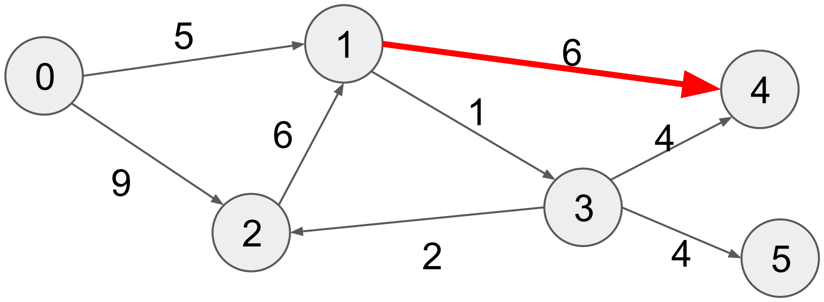


一般圖正權重的最短路

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

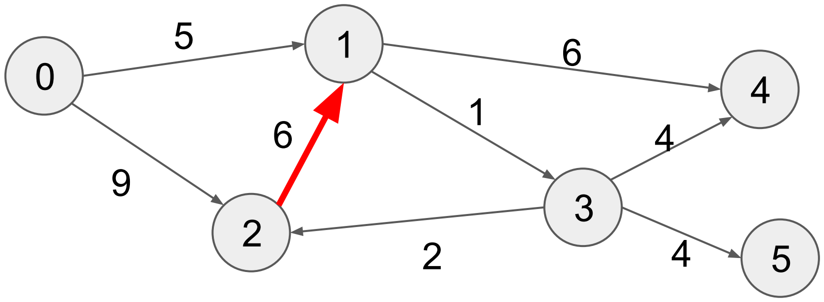


一般圖正權重的最短路

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

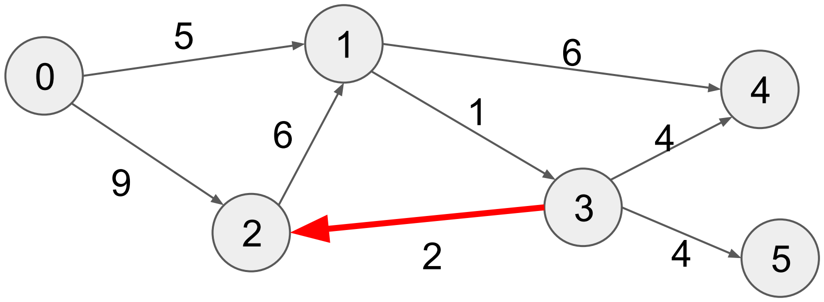


一般圖正權重的最短路

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

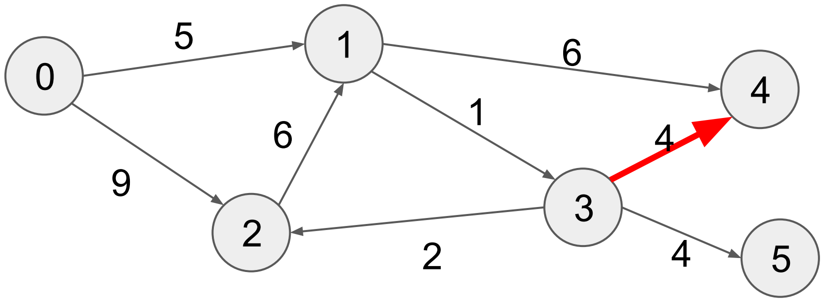


一般圖正權重的最短路

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

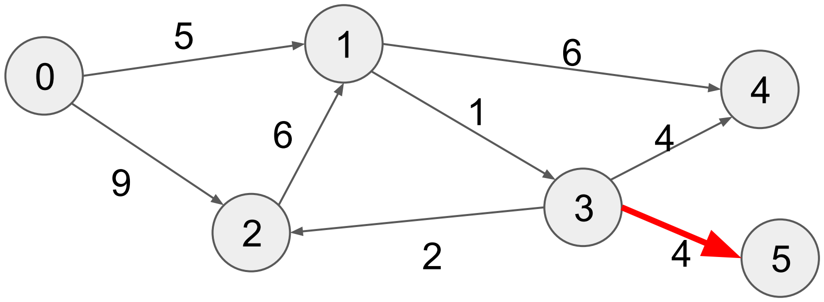


一般圖正權重的最短路

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

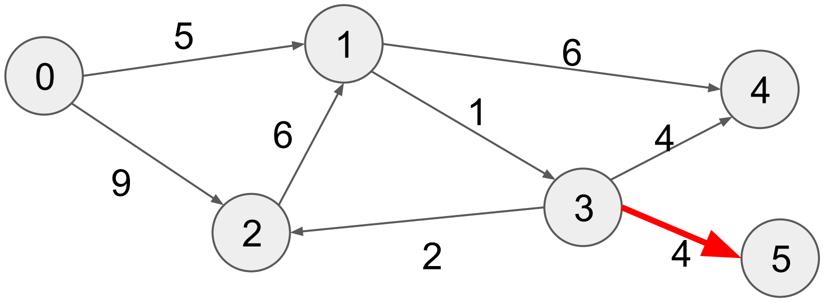


一般圖正權重的最短路

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 (code)

```
const long long INF = 1e18;
int n, dist[maxn];
vector<Edge> edgeList; //邊列表
void BellmanFord(int s) {
    for (int i = 0; i < maxn; i++) dist[i] = INF; //初始化dist
    dist[s] = 0;
    for (int i = 0; i < n - 1; i++) {  //做n - 1次
        for (auto e : edgeList) {       //每次跑整個邊
            dist[e.v] = min(dist[e.v], dist[e.u] + e.w); // relaxation 操作
        }
    }
}
```

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

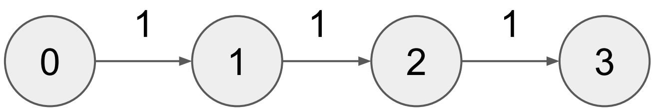
Bellman Ford

- 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

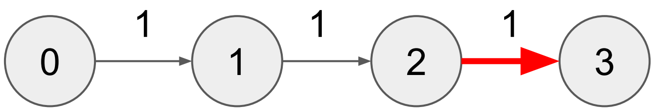


i	0	1	2	3
dis[i]	0	INF	INF	INF

Bellman Ford

對第一條邊做relaxation

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

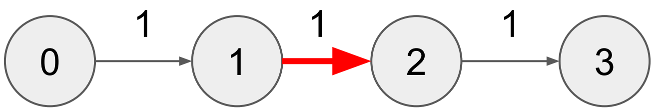


i	0	1	2	3
dis[i]	0	INF	INF	INF

Bellman Ford

對第二條邊做relaxation

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

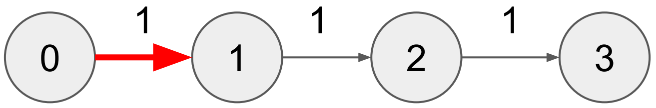


i	0	1	2	3
dis[i]	0	INF	INF	INF

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

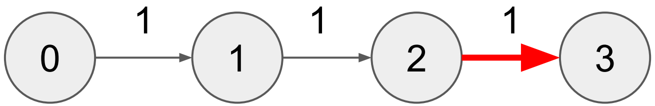


i	0	1	2	3
dis[i]	0	1	INF	INF

Bellman Ford

對第一條邊做relaxation

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

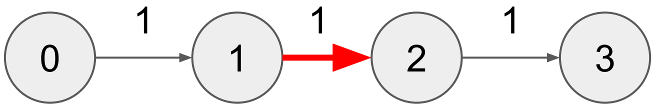


i	0	1	2	3
dis[i]	0	1	INF	INF

Bellman Ford

對第二條邊做relaxation

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

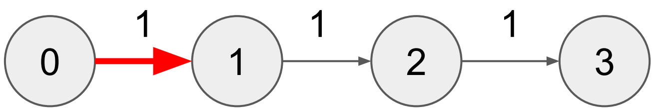


i	0	1	2	3
dis[i]	0	1	2	INF

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

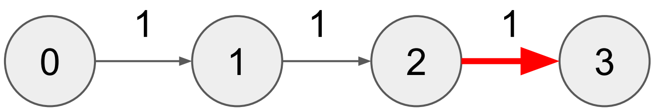


i	0	1	2	3
dis[i]	0	1	2	INF

Bellman Ford

對第一條邊做relaxation

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

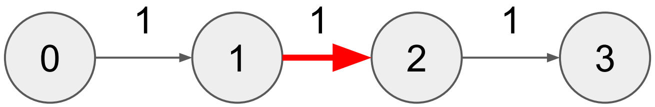


i	0	1	2	3
dis[i]	0	1	2	3

Bellman Ford

對第二條邊做relaxation

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

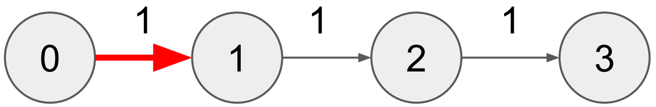


i	0	1	2	3
dis[i]	0	1	2	3

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

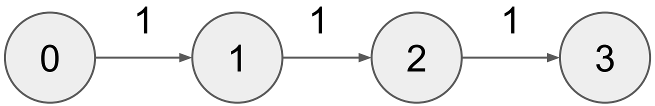


i	0	1	2	3
dis[i]	0	1	2	3

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

- 複雜度分析

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

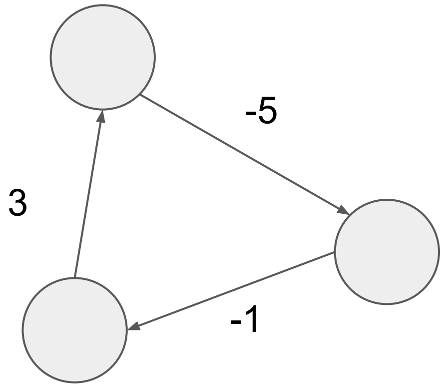
- 權重出現負權重, BellmanFord 和 Dijkstra還可以運作嗎？

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

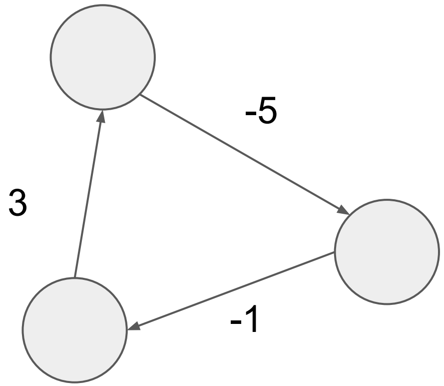
一般圖負權邊的最短路

- 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



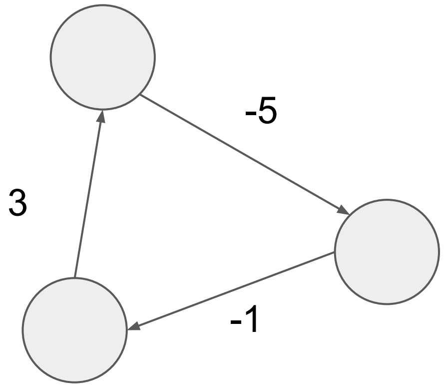
一般圖負權邊的最短路

- 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

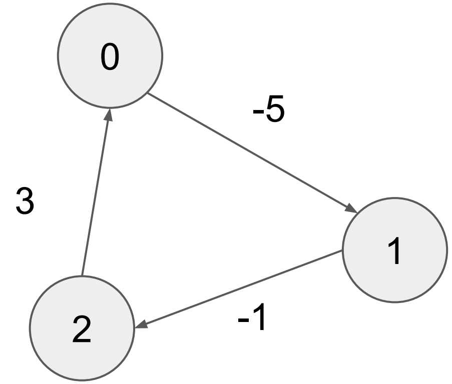
- 權重出現負權重，BellmanFord 和 Dijkstra還可以運作嗎？
  - Dijkstra 不行 (可以想想看為什麼 Hint:貪心性質還會不會成立)
  - BellmanFord 不一定
- 負環
  - 找到一個環，他的總和是負數
  - 在有負環的圖中，多繞幾圈他會形成更短的路徑
    - 所以不存在最短路徑
- 若圖沒有出現可以到達的負環，
  BellmanFord依然可以照常運作。



Bellman Ford遇到負環

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

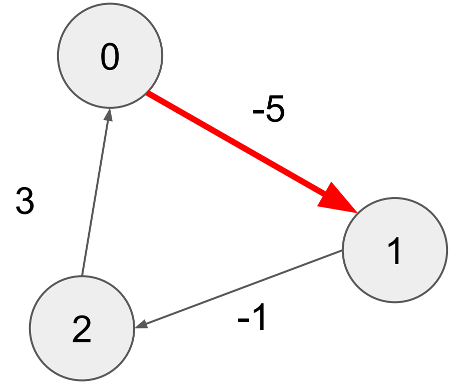
i	0	1	2
dis[i]	0	INF	INF



Bellman Ford遇到負環

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

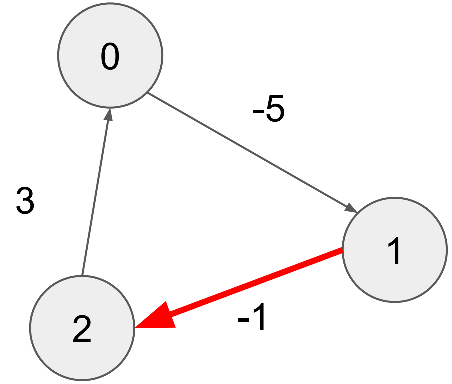
u	v	w
0	1	-5
1	2	-1
2	0	3



Bellman Ford遇到負環

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

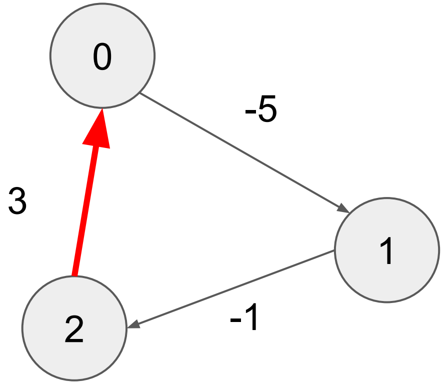
i	0	1	2
dis[i]	0	-5	-6



Bellman Ford遇到負環

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

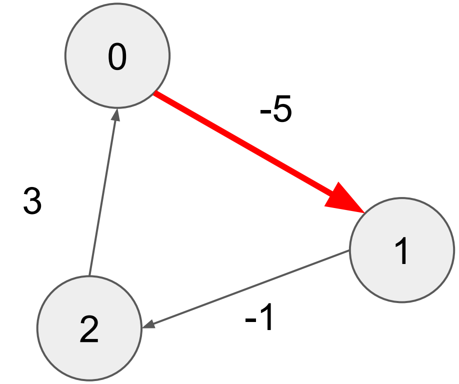
i	0	1	2
dis[i]	-3	-5	-6



Bellman Ford遇到負環

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

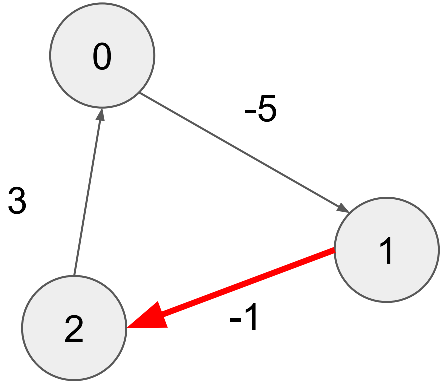
i	0	1	2
dis[i]	-3	-8	-6



Bellman Ford遇到負環

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

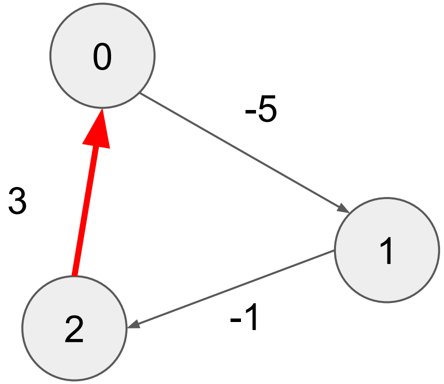
i	0	1	2
dis[i]	-3	-8	-9



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

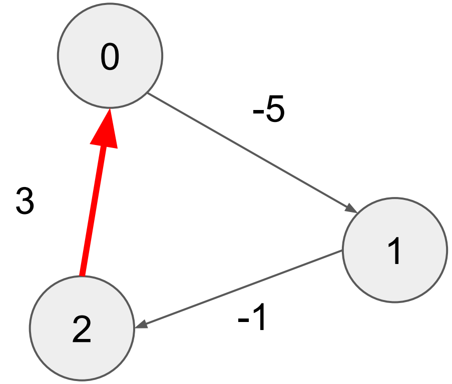


Bellman Ford遇到負環

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

```
const long long INF = 1e18;
int n, dist[maxn];
vector<Edge> edgeList; //邊列表
bool negative_cycle() {
    for (int i = 0; i < maxn; i++) dist[i] = 0; //初始化dist
    for (int i = 0; i < n; i++) {  //做n - 1次
        for (auto e : edgeList) {      //每次跑整個邊
            if (dist[e.v] > dist[e.u] + e.w) {
                dist[e.v] = dist[e.u] + e.w; // 第n次還有relaxation的話代表有負環
                if (i == n - 1) return true;
            }
        }
    }
    return true;
}
```

一般圖全點對最短路徑

- 如果圖為正值權, 那就做V次dijstra複雜度O(V(E+V)lgV)
- 如果圖為負值權, 那就做V次bellman Ford複雜度O(EV^2)

一般圖全點對最短路徑

- 如果圖為正值權, 那就做V次dijstra複雜度O(V(E+V)lgV)
- 如果圖為負值權, 那就做V次bellman Ford複雜度O(EV^2)
- 但但但！！！！！！

一般圖全點對最短路徑

- 如果圖為正值權，那就做V次dijstra複雜度O(V(E+V)lgV)
- 如果圖為負值權，那就做V次bellman Ford複雜度O(EV^2)
- 但但但！！！！！！
- 如果圖為完全圖 E = V^2
- 正值權複雜度O(V^3lgV)
- 負值權為O(V^4)
-

一般圖全點對最短路徑

- 再看一下relaxation的式子dis[v] = min(dis[v], dis[k] + dist(k, v))

一般圖全點對最短路徑

- 再看一下relaxation的式子dis[v] = min(dis[v], dis[k] + dist(k, v))
- 我們改寫一下式子讓dis[u][v]為u到v的最短距離

一般圖全點對最短路徑

- 再看一下relaxation的式子dis[v] = min(dis[v], dis[k] + dist(k, v))
- 我們改寫一下式子讓dis[u][v]為u到v的最短距離
- dis[u][v] = min(dis[u][v], dis[u][k] + dis[k][v])

一般圖全點對最短路徑

- 再看一下relaxation的式子dis[v] = min(dis[v], dis[k] + dist(k, v))
- 我們改寫一下式子讓dis[u][v]為u到v的最短距離
- dis[u][v] = min(dis[u][v], dis[u][k] + dis[k][v])
- 有沒有發現這個很像dp式！

一般圖全點對最短路徑

- 再看一下relaxation的式子dis[v] = min(dis[v], dis[k] + dist(k, v))
- 我們改寫一下式子讓dis[u][v]為u到v的最短距離
- dis[u][v] = min(dis[u][v], dis[u][k] + dist(k, v))
- 有沒有發現這個很像dp式！
- dp(k, u, v) 為利用前k個節點relaxation後的結果

一般圖全點對最短路徑

- 再看一下relaxation的式子dis[v] = min(dis[v], dis[k] + dist(k, v))
- 我們改寫一下式子讓dis[u][v]為u到v的最短距離
- dis[u][v] = min(dis[u][v], dis[u][k] + dist(k, v))
- 有沒有發現這個很像dp式！
- dp(k, u, v) 為利用前k個節點relaxation後的結果
- dp(k + 1, u, v) = min{dp(k, u, v), dp(k, u, k + 1) + dp(k, k + 1, v)}

一般圖全點對最短路徑

- 其實討論 k + 1的時候只需要用到 k 的部分所以可以重複使用 dis 陣列！！
- floyd warshall 演算法

floyd warshall 演算法(code)

```
int Vertex = 100;
int dist[Vertex][Vertex];
for (int k = 0; k < Vertex; k++) {
    for (int i = 0; i < Vertex; i++) {
        for (int j = 0; j < Vertex; j++) {
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
        }
    }
}
```

一般圖全點對最短路徑

- 其實討論 k + 1的時候只需要用到 k 的部分所以可以重複使用 dis 陣列！！
- floyd warshall 演算法
- 複雜度分析 O(V^3)

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
  - 一筆劃問題
  - ……

練習題目

- 到Formosa OJ 上Join 第36個 group

33	Intractable Problems, 2021 Spring	Closed	Apply
34	荊宇泰教授演算法 2021 Spring	Public	Join
35	葉宗泰教授離散數學 2021 Spring	Public	Join
36	Introduction to Algorithms 2021 Fall, Kai-Chiang Wu	Public	Join
37	李毅郎教授圖形理論導論 2021 Fall	Public	Join

練習題目

- [Formula 1]()
- 題目

  求出最小生成樹並且在這個生成樹中求出全點對路徑和

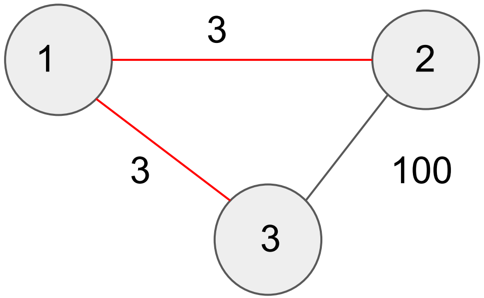
  (保證最小生成樹只有一種)

練習題目

- [Formula 1]()
- 範例測資

全點路徑和為24

u	v	w
1	2	3
1	3	3
2	1	3
2	3	6
3	1	3
3	2	6

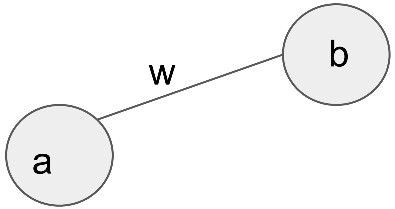


練習題目

- 方向一 最小生成樹可以用kruskal方式找出來
- 那該如何求出全點對路徑和呢？

練習題目

- 方向一 最小生成樹可以用kruskal方式找出來
- 那該如何求出全點對路徑和呢？



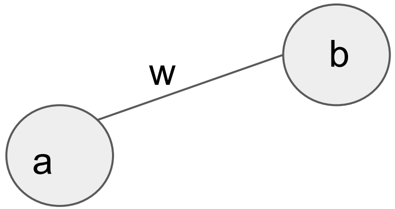
會有 a * b * 2 個點對經過 w 這條邊！

練習題目

會有 a * b * 2 個點對經過 w 這條邊！

所以在dfs時候順便維護子樹的sz !!!!

最後將所有的答案加起來



練習題目

- Mad mobile phone gamer !
- 裸的全點對最短路給你們練習

練習題目

- Deducting Weights
- 如何減少路徑使得從 1 到 n 的最短路徑會經過此路徑

練習題目

- Deducting Weights
- 如何減少路徑使得從 1 到 n 的最短路徑會經過此路徑，減少的量要最少
- 假設(x, y) 之間存在一條，那麼我的路徑要變成 dist(1, n) - dist(1, x) - dist(y, n)

練習題目

- Deducting Weights
- 如何減少路徑使得從 1 到 n 的最短路徑會經過此路徑，減少的量要最少
- 假設(x, y) 之間存在一條，那麼我的路徑要變成 dist(1, n) - dist(1, x) - dist(y, n)
- 該如何求出 任何點到 n 的最短距離呢？

練習題目

- Deducting Weights
- 如何減少路徑使得從 1 到 n 的最短路徑會經過此路徑，減少的量要最少
- 假設(x, y) 之間存在一條，那麼我的路徑要變成 dist(1, n) - dist(1, x) - dist(y, n)
- 該如何求出 任何點到 n 的最短距離呢？
- 提示：如果將邊反轉呢？
- 這樣會變成什麼

課後練習題目

- Drug Dealer
- Building Highways
- Time Machine Network
- TIOJ 1509 . 地道問題
- Problem - 938D - Codeforces
- Problem - 1463E - Codeforces