基礎圖論

baluteshih

2019 年 10 月 23 日

1 圖論簡介

「圖是由若干給定的頂點及連接兩頂點的邊所構成的圖形，這種圖形通常用來描述某些事物之間的某種特定關係。頂點用於代表事物，連接兩頂點的邊則用於表示兩個事物間具有這種關係。」
以上是來自維基百科的說明。在資訊的講法上，一個圖 (Graph)G 通常包含著點 (Vertex)V 跟邊 (Edge)E。一般慣用是會把一個邊 e 表示成 $e = (u, v), u, v \in V$。
初學圖論可能得稍微記好一些專有名詞，畢竟不少東西是高中數學不會接觸到的。

1.1 圖的分類

1. 無向圖 (Undirect graph)：圖中的任何邊 (u, v) 與 (v, u) 等價，意指你可以從 u 走到 v 也能走回來。
2. 有向圖 (Direct graph)：圖中的邊 $(u, v)! = (v, u)$，通常邊 (u, v) 代表這條邊允許從 u 走到 v，但不能反向走。

1.2 圖論術語

1. 點數、邊數：嚴謹一點我們會以 $|V|$ 和 $|E|$ 表示，但為了簡單書寫，我們在寫複雜度時皆會以 V, E 表示。
2. 權重 (weight)：有時候我們會在每一個點和邊附帶一個數稱為「權重」。比較常見的是邊的權重，通常會作為表達長度或花費時間的方法。
3. 度 (degree)：無向圖中，一個點連接的邊數即稱為這個點的度。在有向圖中可細分成為入度 (in-degree) 跟出度 (out-degree)，分別代表以這個點為終點跟起點的邊的數量。
4. 相鄰 (adjacent)：無向圖中，兩個點 u, v 相鄰代表存在一個邊 $e_i = (u, v)$。

5. 指向 (consecutive)：有向圖中，u 指向 v 代表存在一個邊 $e_i = (u, v)$。
6. 路徑 (path)：一個由 u 走到 v 的路徑，可以經由一連串的點由指向 (當然不能反著指) 或相鄰關係構成。
7. 行跡 (trace)：一條路徑中所有的邊兩兩相異，稱為行跡。
8. 簡單路徑 (track)：一條路徑中除了起點終點外，所有的點兩兩相異，稱為簡單路徑。
9. 環 (cycle)：一條簡單路徑的起終點相同，稱為環。
10. 迴路 (circuit)：一條行跡的起終點相同，稱為迴路。
11. 自環 (loop)：一條邊 $e_i = (u, v)$ 滿足 $u = v$，e_i 即稱為自環。
12. 重邊 (multiple edge)：在一張圖中，存在 e_i, e_j 滿足 $i! = j$ 且 $e_i = e_j$，則稱為重邊。
13. 連通 (connected)：無向圖中，若 u 和 v 存在路徑，則 u 和 v 連通。若一群點兩兩連通，則這些點都連通。

1.3 一些特殊的圖

在某些特殊的圖中，可能會有一些性質提供我們更多的解題方向，又或者是會少掉不少麻煩條件，也會使題目變得簡略或漂亮一些。

1. 簡單圖 (Simple graph)：圖中不存在自環及重邊即稱為簡單圖。
2. 連通圖 (Connected graph)：在無向圖上，如果任兩個點都可以經由一些邊互相訪問，那這張圖就是連通圖。
3. 樹 (Tree)：一張沒有環且連通的圖稱為一棵樹。
4. 森林 (Forest)：一張由數棵不連通的樹組成的圖稱為森林。
5. 完全圖 (Complete graph)：無向圖上，對於任何一個點對 (u, v)，都存在邊 $e_i = (u, v)$。一張 n 個點的完全圖通常會以 K_n 簡記之。完全圖在集合上被稱為「團」(clique)。
6. 競賽圖 (Tournament Graph)：有向圖上，對於任何一個點對 (u, v)，都存在邊 $e_i = (u, v)$。
7. 有向無環圖 (Directed acyclic graph, DAG)：沒有環的有向圖稱為一張 DAG。
8. 二分圖 (Bipartite graph)：如果你能把這張圖的點分成兩個集合，且對於任何的邊 $e_i = (u, v)$ 都滿足 (u, v) 分別在不同的集合裡的話，那這張圖就是二分圖。
9. 平面圖 (Planar graph)：可以畫在平面上並且使得不同的邊可以互不交疊的圖。

1.4 圖之間的關係

1. 子圖 (subgraph)：如果兩個圖 $G=(V,E);G'=(V',E')$ 滿足 $V'\subseteq V$ 且 $E'\subseteq E$，則稱 G' 是 G 的子圖。
2. 補圖 (complement graph)：令 $G=(V,E)$ 是一個圖，K 包含所有 V 的二元子集 (2-element subset)。則圖 $H=(V,K\diagdown E)$ 是 G 的補圖。簡單來說就是讓原本的邊消失，原本不存在的邊出現。
3. 同構 (isomorphic)：如果對於兩個圖 G,H 的點集 $V(G),V(H)$，存在一個一一對應的函數 $f:V(G)\to V(H)$，使得對於 G 中的任何兩點 v_i,v_j，v_i 與 v_j 相鄰若且唯若 $f(v_i)$ 與 $f(v_j)$ 相鄰，稱 G 和 H 同構，記為 $G\simeq H$。簡單來說就是這兩張圖可以透過重新編號來變成長得一樣的圖。
4. 生成樹 (spanning tree)：無向圖中，一個樹的子圖稱為一棵生成樹。

2 儲存及遍歷

看完冗長的名詞介紹之後，可以不用急著把他們記起來，常用就會熟了 XD。

2.1 儲存

對於一張指定的圖，我們需要某些方式來把他儲存起來。以下提到的兩種方式都有各自的用途所在，還請讀者要能在不同的情況下運用學會兩者。

1. 鄰接矩陣 (adjacency matrix)：開一個 $V\times V$ 的資料結構 M(通常會用二維陣列)，$M_{a,b}$ 代表的是點 a 至 b 的邊數或權重。空間複雜度 $O(V^2)$。加、刪邊時間複雜度 $O(1)$。
2. 鄰接串列 (adjacency list)：開 V 個可變長度的資料結構 (通常在 C++ 用 vector、在 C 用 linked list)，第 i 個裡面放所有第 i 個點指向的點的編號 (和邊權或其他邊的資訊)。空間複雜度 $O(V+E)$，加邊時間複雜度 $O(1)$、刪邊時間複雜度 $O(V)$。

鄰接串列的好處在於，遍歷圖的時候可以保證不會讀到不必要的資訊，複雜度通常都是 $O(V+E)$，空間也相對優秀。而其餘的情況皆使用鄰接矩陣並不會比較差。

2.2 遍歷

把圖存好之後，為了獲取一些圖中的資訊，我們勢必需要一些讀取資訊的方法，這些方法即稱為「遍歷」。

以下的程式碼我們皆會假設圖是以鄰接串列 vector<int> G[] 來存取的。

2.2.1 深度優先搜索 (Depth-First Search, DFS)

同時也是最直觀的遍歷方式，如人在探索路徑一般，找到新的一條路就先走下去，當走到沒有新的路時，便沿著原路返回。要注意的是，一張圖不一定會是連通的，所以必須要額外自己開一個迴圈，跑過所有點，如果沒被遍歷過，就得再呼叫一次函式。
做法通常會以遞迴實現。

Algorithm 1: DFS

```
1 void dfs(int u){
2     vis[u]=1;
3     for(int i:G[u])
4         if(!vis[i]) dfs(i);
5 }
```

時間複雜度為 $O(V+E)$，額外空間複雜度為 $O(V)$。
dfs 也可以使用 stack 實做，不過大概在一些會卡 stack overflow 的 Judge 才會派上用場，實做方法也就只是自己拿 stack 模擬遞迴而已 (如果你是在本機遇到這個問題，可以在編譯器連結器參數加上-Wl,–stack,536870912 來防止 overflow)。

2.2.2 廣度優先搜索 (Breadth-First Search, BFS)

一口氣把看到的路都找出來，然後加入待辦項目中，並且只跟著加入的順序遍歷。這種做法我們可以很快地想到要使用 queue 來維護，並且可以察覺到這相當於每次都是把相鄰起點一條邊的跑完，再跑兩條的...，所以 BFS 其實是可以解決邊權相同時的最短路徑問題的。

Algorithm 2: BFS

```
1 void bfs(int u){
2     queue<int> q;q.push(u),vis[u]=1;
3     while(!q.empty()){
4         int v=q.front();q.pop();
5         for(int i:G[v])
6             if(!vis[i]) q.push(i),vis[i]=1;
7     }
8 }
```

時間複雜度一樣為 $O(V+E)$，額外空間複雜度也為 $O(V)$。

2.2.3 格子圖小技巧

有時候圖會是一張格子圖，有可能會叫你上下左右移動，也可能會叫你馬步的移動，總之各種奇怪的移動方式都是有可能出現的。這時候可以用以下一點小技巧來加快編程速度。以下是以相鄰八格為例。

Algorithm 3: moving tirck

```
int dx[8]={1,-1,0,0,1,1,-1,-1},dy[8]={0,0,1,-1,1,-1,1,-1};
bool check(int x,int y){}//用來檢查(x,y)是否超範圍或不可走
//假設遍歷到(x,y)
for(int i=0;i<8;++i)
    if(check(x+dx[i],y+dy[i]))
        //遍歷(x+dx[i],y+dy[i])
```

可以透過改變 dx 和 dy 來改出不同的移動方法。

2.3 習題

1. (ZJ a290) 給你一張有向圖，問你 A 是否存在路徑能到達 B。
2. (TIOJ 1209) 請你判斷一張圖是否為二分圖。
3. (ZJ c124) 三維迷宮最短路徑。
4. (TIOJ 1143) 馬步最短路徑。
5. (ZJ a597) 某種連通塊判斷。
6. (ZJ d537) 題敘略。
7. (ZJ c402) 給你一張圖，請你根據一些學生兩兩互不可以分在同一組的規則，將學生拆成兩組，並使兩組學生的權重和之差最小，有可能無解。

3 樹

在圖論中，樹是一種性質相對來說多很多的特殊圖，在樹上可以有極多的新算法和問題被發掘出來。

3.1 判斷

對於以下任何一種無向圖的敘述皆是在表示樹，這些敘述也可以當做是判斷一棵樹的方式。

1. 連通且 $V = E + 1$。
2. 任意兩個點之間存在唯一的簡單路徑。
3. 連通，但去掉任意一條邊就不連通。
4. 沒有環，但加上任意一條邊就有環。

5. 若節點編號存在一個順序，除了第一個節點，每個節點都伸出一條邊連到順序比自己前面的節點。

這些性質甚至有時候會被當做是解題的關鍵，可以嘗試理解他們。

3.2 名詞

既然有新的問題被發掘，自然就有新的名詞得記了。

1. 根 (root)：樹上一個具代表性的節點，若題目有給定根節點，則稱這棵樹為「有根樹」，通常會被當成遍歷起點，所以無根樹基本上也得自己亂找一個根給他。
2. 葉子 (leaf)：在樹上度數只有一的節點稱為葉子，通常如果有根樹根節點度數是一的話會獨立不將根視為葉節點，視題目所需。
3. 父節點 (parent)：有根樹中，兩個相鄰的節點，離根較近的節點是另一個節點的父節點，反之為其子節點 (child)。
4. 祖先 (ancestor)：有根樹中，一個節點的祖先包括了他的父節點和父節點的祖先。而反過來說，對於一個節點的祖先，這個節點是他的子孫 (descendant)。有時候自己也可以是自己的祖先。
5. 距離 (distance)：兩點間的距離是它們之間路徑的邊數 (畢竟路徑唯一)，或者是路徑上邊權的加總。
6. 深度 (depth)：有根樹中，一個點的深度是它到根的距離稱為深度。
7. 高度 (height)：有根樹中，一個點到與它距離最大的葉節點的距離稱為高度。根的高度稱為這整顆樹的高度。
8. 子樹 (subtree)：移除一個點之後，原樹會被拆成很多棵樹，稱為子樹。有根樹中，一個節點的子樹通常泛指他的所有子孫 (常會包括自己本身)。
9. 二元樹 (binary tree)：每個節點都至多只有兩個子節點的樹。

如果只是要求深度的話，可以用簡單的 dfs 就能實做，只要多傳一個參數給子節點，計算遞迴深度即可；高度也是大同小異，只是變成要是子節點回傳參數回來，並計算高度最大的子節點再 $+1$。
其他的關係會在接下來的算法中提到。

3.3 二元樹

二元樹比樹又多了更多性質。

3.3.1 遍歷

一棵二元樹的遍歷分為「前序、中序、後序」三種，三者的差別在於造訪根節點的時機。不過實作上我們依然得照前序遍歷 (也就是好正常的 dfs 順序)，若要得到後兩者的造訪順序，只要更改輸出時機就能仿造。

Algorithm 4: Binary Tree's Order

```
1 int lc[],rc[];//左子節點lc，右子節點rc，0表示不存在
2 void dfs(int x){
3     //cout<<x<<" "; 前序遍歷法(Preorder)
4     if(lc[x]) dfs(lc[x]);
5     //cout<<x<<" "; 中序遍歷法(Inorder)
6     if(rc[x]) dfs(rc[x]);
7     //cout<<x<<" "; 後序遍歷法(Postorder)
8 }
```

當你有一棵二元樹的中序和前或後序遍歷順序的話，你就可以還原出原本的二元樹的長相。還原方法通常是採用分治法，主要是利用中序「根節點位置的左右恰為左右子樹」的性質來拆，實作並不困難，複雜度會是 $O(N)$。

3.3.2 二元搜尋樹 (Binary Search Tree)

二元搜尋樹是二元樹的資料結構應用，其應用廣泛，定義如下。

1. 根節點的鍵值必大於左子節點的鍵值，必小於右子節點的鍵值。
2. 根節點的左右子樹都是一棵二元搜尋樹。

對於任何一個序列，你只要依照上面兩條規則就能夠蓋出對應的二元搜尋樹，而讀者應該不難發現，當序列已經排序完成，二元搜尋樹會是一條鏈，蓋樹的時間複雜度會退化至 $O(N^2)$，樹的深度變成 $O(N)$。所以其實二元搜尋樹本身並不是很實用，真正實用的是他的推廣結構，但由於其推廣複雜，這裡先不提，至少先讓讀者知道他的存在。
一點小性質，由於二元搜尋樹本身的定義，他的中序遍歷會剛好是排序好的序列。

3.4 直徑和圓心

一棵樹的直徑定義為樹當中最遠的兩點間的簡單路徑；圓心是能使樹高度最小的根。
要找出直徑可以先隨便 dfs 一遍，由於可以證明，距離任何一個點 u 最遠的點肯定可以做為最遠點對之一，所以可以找到距離 u 最遠的點 v 後再 dfs 一遍，並找到距離 v 最遠的點，即為最遠點對，兩者的簡單路徑就是直徑。
圓心也被證明一定會在直徑上，所以也可以用類似的方法找到圓心，兩者的證明都不難證，讀者有興趣可以自己證證看。

3.5 重心

一棵樹的重心定義為，以這個點為根，那麼所有的子樹 (不算整個樹自身) 的大小都不超過整個樹大小的一半。
同樣可以證明樹重心存在以下兩個性質：

1. 把兩個樹通過一條邊相連得到一個新的樹，那麼新的樹的重心在連接原來兩個樹的重心的路徑上。
2. 把一個樹添加或刪除一個葉子，那麼它的重心最多只移動一條邊的距離。

不過尋找重心可以不用到這兩個性質，只要隨便選取一個根 dfs 一遍，紀錄每個節點有根樹上的子樹大小，再去分別計算每個節點最大的子樹，並找到最大子樹最小的節點就是重心了。
重心存在一些神奇的用途，但這裡先不提。

3.6 時間戳記

紀錄一個有根樹遍歷時，進入這個節點的時間 $in[u]$ 和離開的時間 $out[u]$，可以有很多性質。

Algorithm 5: Tree Timestamp

```
1 int t;
2 void dfs(int x,int f){
3     in[x]=++t;
4     for(int i:G[x])
5         if(i!=f) dfs(i,x);
6     out[x]=t;
7 }
```

1. u 是 v 的祖先若且唯若 $in[u] \leq in[v]$ and $out[v] \leq out[u]$，反之亦然。
2. 新開一個陣列 $arr[]$，滿足 $arr[in[u]] = u$，稱為樹壓平 (或樹序列化)。這個動作可以將樹視為一個序列做事，在往後的算法會再被提到。
3. 一個有根樹節點 u 的子樹正好會是樹壓平的區間 $[in[u], out[u]]$。

於是時間戳記基本上可以先被認為是判斷祖先後代關係的工具。

3.7 最低共同祖先 (Lowest Common Ancestor, LCA)

在一棵有根樹上的兩個節點 u, v，其 $LCA(u, v)$ 即為兩個節點的祖先交集中，深度最深的祖先。
暴力搜尋 LCA 的複雜度是 $O(N)$，這是我們不太樂見的。

3.7.1 倍增法 (Prefix Doubling)

我們可以事先存好每個節點往上一層、二層、四層、... 的祖先。以下令 $L = \log_2 N$

Algorithm 6: Doubling Ancestor

```
int ac[L][N]; //ac[i][j]表示節點j第2^i層的祖先
//先蓋好ac[0][j]，可以用dfs處理好，並令ac[0][root]=root
for(int i=1;i<=L;++i)
    for(int j=1;j<=N;++j)
        ac[i][j]=ac[i-1][ac[i-1][j]];
    //第2^i層的祖先是「第2^i-1層的祖先的2^i-1層的祖先」
```

蓋表複雜度相當於某種 DP，$O(N \log N)$。

蓋好這個表之後我們就可以開始二分搜了，原因是因為我們如果要求 $LCA(u, v)$ 的話，對於 u 的所有祖先都存在著「深度小於等於 $LCA(u, v)$ 的話，就會同時是 v，反之則不是」的單調性，而如果我們限制查詢的左右界是 $[1, 2^{L+1}]$ 的話，就可以在每次查詢中間值的時候落點都在倍增法的表上了。

寫法異常精簡。

Algorithm 7: LCA

```
//記得寫好bool ancestor(int a,int b)，即祖先的判斷
int LCA(int a,int b){
    if(ancestor(a,b))return a; //如果自己是對方的祖先就先預判掉
    for(int i=L;i>=0;--i)
        if(!ancestor(ac[i][a],b)) a=ac[i][a]; //提高下界，a往上跳
    return ac[0][A]; //有了預判，可以保證a不會是b的祖先
}
```

可以很明顯看出查詢一次的複雜度是 $O(\log N)$。

有了 LCA 後，不難證明對於任兩個節點 u, v，若節點深度是 $deep(x)$，那麼 $distance(u, v) = deep(u) + deep(v) - 2 \times deep(LCA(u, v))$。

我們也可以用類似的方法查詢任兩點路徑上的最小邊權，只要多蓋一張表，並在查 LCA 往上跳時取極值即可。

3.8 習題

1. (ZJ c131)：給你一張圖，判斷是否為樹。
2. (TIOJ 1106)：模擬蓋樹。
3. (ZJ c126)：給你二元樹的前序中序，請你輸出後序。
4. (ZJ d526)：模擬蓋二元搜尋樹。
5. (TIOJ 1609) 輸出用一個序列蓋出來的二元搜尋樹的中序遍歷順序。

6. (ZJ b967/TIOJ 1213)：求直徑長度。
7. (ZJ d767)：求任兩點的距離和 LCA。
8. (CS Academy, Triplet Min Sum)：給你一棵樹，問你 Q 次 A, B, C，請你找到一個點 D 使得 D 到 A, B, C 的距離和最小。
9. (TIOJ 1687)：給你一棵樹，問你 Q 次 S, T, K，請你輸出從 S 往 T 走 K 步抵達的點，走過頭請輸出 -1。
10. (2015 TOI 三模 pB/TIOJ 1909) 題敘略。

4 最小生成樹 (Minimum Spanning Tree, MST)

無向圖中，一個樹的子圖稱為一棵生成樹。生成樹有許多長相，好比我們 dfs 或 bfs 的順序其實都可以構成一棵生成樹。
而在所有的生成樹中，邊權總和最小的生成樹，我們就稱他為最小生成樹。

4.1 並查集 (Disjoint Set)

先來提提一個看起來不甚相關的東西，並查集是一種資料結構，他可以支援以下操作。

1. 詢問元素隸屬的集合。
2. 合併兩個集合。

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

4.1.1 理念

當我們想詢問元素隸屬的集合時，我們勢必得回傳這個集合被賦予的編號，而並查集可說是幾乎省略了這個動作，他直接對每個集合賦予一個「老大」，每個元素只要找到自己的老大就可以知道自己在哪個集合上了。
實作上，由於一開始每個元素都尚未合併，都是自己為一個集合，所以我們會令 $boss[i] = i$，代表自己集合的老大是自己，合併的時候只要隨便找一個集合的老大建一條指向另一個集合老大的邊，每次查詢用遞迴的往上走找到老大，而老大的老大就會剛好是自己，以這個為判斷基準就可以了。不過這樣一次查詢的複雜度有可能退化成 $O(N)$，顯然不太優秀。

4.1.2 啟發式合併

建立每個老大的 $rank[i]$ 代表集合任一元素的最大遞迴深度，由於並查集可以視為一棵以眾多老大為有根樹的森林，所以 $rank[i]$ 就相當於樹的高度。可以先看看程式碼。

Algorithm 8: Disjoint Set, Union by rank

```
int boss[],rank[];
void init(){//reset
    for(int i=0;i<n;++i) boss[i]=i,rank[i]=1;
}
int query(int x){
    if(boss[x]==x) return x;
    return query(boss[x]);
}
void Union(int a,int b){
    a=query(a),b=query(b);//先找到自己集合的老大
    if(a==b) return;//一樣就不合併了
    if(rank[a]<rank[b]) boss[a]=b;
    else if(rank[a]>rank[b]) boss[b]=a;
    else boss[a]=b,++rank[a];//rank一樣，高度肯定變高
}
```

容易發現，最後對老大建指向邊的動作只有在 $rank$ 一樣時會讓 $rank$ 提升，其餘兩種情況 $rank$ 都不變。如此一來需要讓最大 $rank$ 提升至 x 就必須要有 2^x 個元素，易知由此蓋出來的樹高不超過 $\log N$，查詢複雜度即為 $O(\log N)$，合併包含著查詢也是 $O(\log N)$。

4.1.3 路徑壓縮

再觀察一下就會發現，除了老大身上以外的 $rank$ 好像都沒有用，那乾脆就試試看能不能直接把子元素的老大指到定位就好啦。於是我們只要修改 $query$ 函數如下。

Algorithm 9: Disjoint Set, path compression

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

在遞迴結束前特別把每個路徑上的元素老大都指向真正的老大，這樣便能獲得優秀的複雜度，其一次查詢的複雜度被證明為 $O(\alpha(N))$，其中 $\alpha(x)$ 是阿克曼函數 $A(x,x)$ 的反函數，$A(x,x)$ 在 $x=4$ 時約為 $2^{2^{10^{19729}}}$，所以 $\alpha(x)$ 幾乎可以說只是一個常數而已。
題外話，如果只寫路徑壓縮不寫啟發式合併的話，複雜度會降為 $O(log_{2+\frac{f}{N}}N)$，其中 f 為查詢次數。

並查集的用途很多，不過接下來我們將會介紹他在最小生成樹上的應用。

4.2 Kruskal's algorithm

回到最小生成樹，Kruskal 觀察到如果我們將原圖中連接一部分點的生成樹，稱為最小生成子樹，可以發現一些特性：

1. 一個單獨的點，可以視作一棵最小生成子樹。
2. 兩棵最小生成子樹連結成一棵最小生成子樹，以兩棵最小生成子樹之間權重最小的邊進行連結，顯然是最好的。
3. 三棵最小生成子樹連結成一棵最小生成子樹，先連結其中兩棵連結權重最小的最小生成子樹，才連結第三棵，總是比較好。

所以我們可以 greedy 地，從最小權重的邊開始選起，在選取一條邊時，將邊連到的兩個點合併成連通，如果下一個選的邊連結到的兩個點已連通，則不選 ── 正好可以使用並查集！
先對邊排序完之後，我們發現這個動作可以跑完一次邊就完成，複雜度 $O(E(\log E + \alpha(V)))$，常數很理想，實作也不難。

4.3 Prim's algorithm

可以從特性中觀察出另一個性質，從一棵 MST 連出去的邊中，找到離樹最近的點並合併，重複執行後一定會是最好的。實作上我們可以任取一個起點當成初始 MST，每次掃過所有相鄰點後把邊權當成距離加入資料結構內，每次從資料結構內拿出一個最小距離的點 (如果此點已被合併就繼續拿)，再進行合併 (這裡不用使用並查集)。而維護最小距離的資料結構我們便能很輕易的直接想到 priority_queue，故因每個點只會被合併一次，每條邊都會遍歷一次，複雜度為 $O((V+E)\log E)$。如果使用一種叫費波那契堆 (Fibonacci heap) 的資料結構的話可以達到 $O(E + V\log V)$，但常數不甚理想，也不好寫，並不是很實用。

4.4 Borůvka's algorithm

上面提到可以找到離樹最近的點並合併，那不如我們讓所有的最小生成子樹都做一次同樣的動作如何？實作上可以每次掃一遍每棵最小生成子樹連出去權重最小的邊，然後硬是將這些邊合併，若合併完還不是一棵完整的 MST 的話，就重新掃一邊重新合併，一樣會利用並查集維護連通性。
最差的 case 不外乎是，每個最小生成子樹每次都兩兩成對，最多會執行 $\log V$ 回合，總複雜度 $O((V+E)\log V)$。期望複雜度可以達到 $O((V+E))$，非常優秀。(小常識：因為每次並查集都會被合併＋查詢，所以 α 可以完全省略。)

4.5 次小生成樹

枚舉非生成樹的邊，然後擺進去最小生成樹拔掉一條邊形成新的生成樹，取最小值，即為次小生成樹。理論上複雜度會是 $O(EV)$，可以再做優化。

觀察：擺一條新的、連結 A, B 的邊進去一棵樹，一定會在 A, B 之間形成一個環，所以只拆掉這條環上權重最大的邊 (除了擺進去的邊)，就能形成新的、對於這條邊次小的樹。而環上權重最大的邊 $= A, B$ 路徑最大值，故我們能使用 LCA 來查詢這些東西。

蓋 MST$O((V+E)\log V) \to$ 蓋 LCA 表 $O(V \log V) \to$ 枚舉剩下的邊，每次用 LCA 找到路徑最大值，運算後取最小值後即為答案 $O(E \log V)$。總複雜度還是 $O((V+E)\log V)$，而且只多了一倍左右的常數。

4.6 最小比率生成樹

若現在邊權改成一個 $pair(a, b)$，且權重相加的時候是獨立各自相加的，能否求出最小的生成樹滿足權重相加後的 $(f(v), g(v))$，有 $\frac{f(v)}{g(v)}$ 最小？這類題目不只會出現在生成樹，不過生成樹是一個比較常見的題型。

我們可以假設最佳解為 t，則對於所有的 $\frac{f(v)}{g(v)}$ 有 $\frac{f(v)}{g(v)} \geq t$，做點移項之後得 $f(v) - g(v) \times t \geq 0$。也就是說，對於任何一棵生成樹，$f(v) - g(v) \times t \geq 0$ 永遠成立。

於是若我們任意猜測一個 t，就可以令所有的邊權重 (a, b) 的新權重為 $a - b \times t$，並蓋出一個最小生成樹，赫然發現若 t 太小，$f(v) - g(v) \times t > 0$；反之 $f(v) - g(v) \times t < 0$，所以我們能對 t 二分搜來得到答案。

4.7 習題

1. (ZJ a129/TIOJ 1211)：最小生成樹。
2. (ZJ d808)：給你兩兩之間隸屬同一部落的關係，請你找到至多有幾個部落和最大的部落大小。
3. (ZJ b570)：給你一張圖，給你拔邊的順序，問你分別拔掉幾條邊之後，圖被分成了幾塊。
4. (TIOJ 1795)：給你一張 n 個點 m 條邊的無向簡單圖 G，每條邊的邊權非 0 即 1。問 G 是否存在一個生成樹其邊權總和為 k。
5. (106 全國賽 pE/TIOJ 2038)：題敘略。
6. (ZJ c495/TIOJ 1445)：次小生成樹。
7. (POJ 2728)：最小比率生成樹。

5 有向無環圖 (Directed Acyclic Graph, DAG)

一張有向圖，沒有環，就可以衍伸出一些性質。對於一個 DP 的狀態圖，我們也可以說他是一張 DAG。

5.1 拓樸排序 (Topological Sort)

訂定出一個點的排序 $\{v_1, v_2, \cdots, v_n\}$，滿足 $\forall i \leq j$ 皆不存在 v_i 到 v_j 的路徑，就是拓樸排序。一個好的 buttom up DP 順序，也是一種拓樸排序。
拓樸排序有以下兩種方法：

1. Greedy：把入度為 0 的點先丟進資料結構內，不斷從資料結構內拿出點並把他指向的點入度減一，若有新的點入度歸零就也把他丟進資料結構內，直到資料結構是空的為止。如果還有點沒被遍歷到，表示這張圖不存在拓樸排序；否則拿出點的順序即為一種拓樸排序。
   這裡的資料結構通常會用 stack 或 queue，若使用 priority_queue，可以得到極值字典序。

2. DFS：對整張圖 DFS，紀錄各點的離開時間。如果遍歷到一點 u 時，存在某一條邊 (u, v) 滿足 v 已進入但尚未離開，代表圖不存在拓樸排序；否則將點按照離開時間後到前排序即為一種拓樸排序。

DAG 肯定存在拓樸排序，所以我們也可以用拓樸排序來判斷有向圖是否為 DAG。
有些 DP 題會直接叫你在 DAG 上 DP，不用看到圖就害怕，其實也只不過是轉移來源不太一樣的 DP 而已，好好拓樸排序就不會出事了。

5.2 習題

1. (CF 510C)：有 n 個字串。請你找出一種英文字母的大小關係，使得輸入是依照這個關係由小到大字典序排序的。

2. (ZJ a454)：DAG 最長路徑問題。

3. (ZJ b058)：找出會影響 DAG 最長路徑的點數量。

6 最短路徑

最短路徑上不允許存在負環，如果有，則為 NP-Complete 問題。

6.1 單點源最短路徑

求出一個點到所有點的最短路徑，其實就相當於是建出以起點為根的一棵最短路徑樹。

6.1.1 鬆弛 (Relaxation)

對於任意兩個點 u, v，如果起點 s 到他們的距離 d_u, d_v 滿足 $d_u + w_{u,v} < d_v$(其中 $w_{u,v}$ 為邊 (u, v) 的權重)，那我們就可以把 d_v 更新為 $d_u + w_{u,v}$，使 s 到 v 的距離變短，這個動作就叫做鬆弛。

可以想像，如果在圖上不斷尋找可鬆弛的點鬆弛，當再也找不到點的時候，肯定所有點都能得到自己的最短路徑。如果要蓋出最短路徑樹，那只要紀錄每個點最後一次被鬆弛的點是誰，以其為父節點就好。

6.1.2 Bellman-Ford Algorithm

秉持著不斷尋找可鬆弛點的理念，把起點的最短路徑設為 0，其他點先設為無限大，不斷 $O(E)$ 枚舉邊來鬆弛，直到不需要更新就可以停止了。而我們可以發現，最短路徑不會經過同一個點，所以最多重複執行 $V - 1$ 次，複雜度為 $O(VE)$。如果執行第 V 次的時候還有點能被鬆弛，表示存在負環，所以 Bellman-Ford 也可以當作是判斷負環的一種方法。

這個演算法有一個被稱為 Shortest Path Faster Algorithm(SPFA) 的優化方法，每次枚舉邊都只枚舉起點是被鬆弛過的點的邊，鬆弛過別人的點沒有被重新鬆弛也都不要用的話，期望複雜度可以達到 $O(V + E)$，不過 Worst Case 還是 $O(VE)$。

6.1.3 Dijkstra's Algorithm

如果邊權都是非負實數，會存在一個性質：用完成一部分的最短路徑樹上的點去鬆弛其他不在樹上的點，所得到的離根最近的點肯定也在最短路徑樹上。有了這個性質我們就能用類似 Prim's Algorithm 的做法來快速的求出最短路徑了，其理念在每次加一個點進來就用他鬆弛別的點一遍，並把鬆弛過的結果丟進 priority_queue 內，每次把路徑最短的點拿出來檢查是否在最短路徑樹上就好了。以下假設圖是用 vector< pair<int,int> > G[] 儲存的。

Algorithm 10: Dijkstra

```
1 typedef pair<int,int> pii;
2 #define MP make_pair
3 #define F first
4 #define S second
```

```
5  void Dijkstra(int s,int *dis){
6      priority_queue<pii,vector<pii>,greater<pii>> pq;
7      for(int i=1;i<=n;++i) dis[i]=INF;
8      pq.push(MP(0,s)),dis[s]=0;
9      while(!pq.empty()){
10         auto tmp=pq.top();pq.pop();
11         if(dis[tmp.S]!=tmp.F) continue;//表示點已在最短路徑樹上
12         for(auto i:G[tmp.S])
13             if(tmp.F+i.F<dis[i.S])//鬆弛
14                 dis[i.S]=tmp.F+i.F,pq.push(MP(tmp.F+i.F,i.S));
15     }
16 }
```

最慘每個邊都會需要鬆弛一次，複雜度 $O(E\log E)$，使用費波那契堆，可以優化至 $O(E+V\log V)$。

6.2 多點源最短路徑

一口氣求出點對的最短路徑，可以使用 V 次單點源最短路徑來完成這件事。

6.2.1 Floyd-Warshall Algorithm

我們來用點老方法，也就是動態規劃。
令 $dp[k][i][j]$ 代表，若只以點 $1\sim k$ 當中繼點的話，則 $dp[k][i][j]$ 為 i 到 j 的最短路徑長。
基底為 $dp[0][i][j]=w[i][j]$，其中 $w[i][j]$ 為圖上一開始就有的邊權 (重邊記得取最小值)，若無邊權則預設為無限大。
則當加點進去之後考慮以其為中繼點，存在轉移式：

$$dp[k][i][j]=min(dp[k-1][i][k]+dp[k-1][k][j],dp[k-1][i][j])$$

時間複雜度為 $O(V^3)$，空間也為 $O(V^3)$。但由於可以滾動，以下附的 code 將為空間 $O(V^2)$ 的做法，其程式碼異常好寫，常數也非常小。

Algorithm 11: Floyd-Warshall

```
1  int dp[][],w[][];
2  for(int i=1;i<=n;dp[i][i]=0,++i)//根據題目需求，有可能要把自己到自己設為0
3      for(int j=1;j<=n;++j)
4          dp[i][j]=w[i][j];
5  for(int k=1;k<=n;++k)
6      for(int i=1;i<=n;++i)
7          for(int j=1;j<=n;++j)
8              dp[i][j]=min(dp[i][k]+dp[k][j],dp[i][j]);
```

若執行完畢後存在 $dp[i][i]<0$，表示存在負環，所以 Floyd-Warshall 也是一個判斷負環的方法。

6.2.2 Johnson's Algorithm

應該不少讀者能發現，當 E 夠小的時候，跑 V 次 Dijkstra 可能會比 Floyd-Warshall 優，於是為了讓 V 次 Dijkstra 在負權上也能運作就出現了這樣一個演算法，由於需要使用 Bellman-Ford 預處理，所以在單點源上並不會比較好。
作法有些複雜也不常需要使用，有興趣的讀者可以自己查看看。

6.3 差分約束系統

當一個多變數 $x_1 \sim x_N$ 的線性規劃問題可以用 $x_i - x_j \leq d$ 的形式表示，我們就可以用最短路徑演算法求解。
建圖：若有限制 $x_j - x_i \leq d \Rightarrow x_i + d \geq x_j$，則建邊 (i, j) 且邊權為 d。
可以發現，當固定了起點，若不斷出現 $x_i + d < x_j$ 能鬆弛的話，就相當於不斷在把變數推至符合限制的情況。所以若不存在負環 (不會不斷地鬆弛)，則肯定存在一組解滿足不等式，而且肯定為最佳解。而若一個點的最短距離是無限大，則表示這個點的變數範圍無限大。

6.4 習題

1. (ZJ d793)：方格上的最短路徑。
2. (ZJ a674)：給你一張圖，問你 Q 次任兩點間所有路徑上的最大值的最小值。
3. (TIOJ 2058)：次短路徑。
4. (TIOJ 1706)：給你一張圖，每條邊的權重都是日期 T 的線性函數，試問在所有日子中，最短路徑長為多少。
5. (CF 938D)：給你一張圖，對於每個點 i 請你找到 $min\{2d(i, j) + a_j\}$，其中 $d(i, j)$ 是 i, j 的最短路徑，a_j 是給定的權重。
6. (POJ 3169)：有 N 隻牛。有些牛跟牛之間是不願相離超過某單位距離；有些牛跟牛之間不願靠得比某單位距離近。且 $i < j$ 時，牛 i 必須在牛 j 的位置左邊。請問是否可以滿足所有限制，若可以的話輸出牛 1 和牛 N 間的最大距離，又或者是判斷最大距離是否為無限大。