圖論初階

Yihda Yol

2016 年 9 月 22 日

1 圖論簡介

圖論，顧名思義，是研究圖（Graph）的一門學問。那什麼是「圖」呢？

1.1 圖與圖的種類

> *In the most common sense of the term, a graph is an ordered pair $G = (V, E)$ comprising a set V of vertices, nodes or points together with a set E of edges, arcs or lines, which are 2-element subsets of V (i.e., an edge is associated with two vertices, and the association takes the form of the unordered pair of the vertices). To avoid ambiguity, this type of graph may be described precisely as undirected and simple.*
>
> — *by wiki*

簡單來說，一般提到（也是最基本）的圖，由一些點（vertex, pl. vertices）和一些連接兩相異點的邊（edge）構成。邊是一個包含兩點的集合，通常寫為 $e_i = \{v_i, v_j\}(v_i \neq v_j; v_i, v_j \in V)$，$v_i, v_j$ 分別代表邊的端點。

如此，一個圖 G 便可以表示為 (V, E)，其中 V 是點的集合，E 是邊的集合。

可以看出來，圖常是用來描述物件與物件的二元關係。

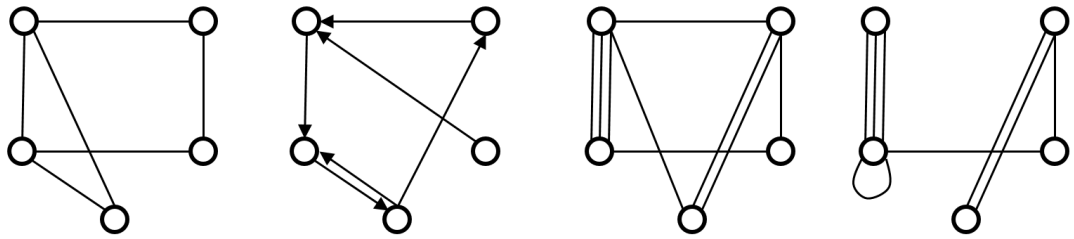
為了避免模稜兩可，精準而言，上述定義的圖要加上兩個形容詞「無向」（undirected）和「簡單」（simple）。這是因為還有很多種變種的圖，列舉如下：

1. 有向圖（directed graph）：有向圖的每一個邊（稱為有向邊）都是一個包含兩點的有序對 (v_i, v_j)（而不是集合），也因此 $(v_i, v_j) \neq (v_j, v_i)$。有向邊 (v_i, v_j) 的兩端點可分為起點 v_i 和終點 v_j。

2. 多重圖（multigraph）：多重圖的邊集並不是一個集合，而是一個多重集。也就是說，可以有很多條相同的邊，稱為重邊（multiple edge）。

3. 偽圖（pseudograph）：偽圖允許邊只連接一個點，稱為自環（loop）。因為在偽圖中邊表示的不一定是兩點間的二元關係，因此才會稱為「偽」。

我們經常將圖畫出來，以方便表達。通常以一個圓圈代表點，一條線代表邊。如果是有向邊，則畫一個箭頭由起點指向終點。以下是幾個例子：



無向的簡單圖　有向的簡單圖　無向的多重圖　無向的偽圖

1.2 圖論術語們

圖雖然看起來就只是一堆點和邊，但卻也是一個充滿大量術語的學問。以下先列舉一些常見的術語：

關於圖的基本性質

1. 點數、邊數：點集和邊集的元素數量。之後寫複雜度時將簡記為 V、E。點數又稱為「階」（order）。

2. 無向圖、有向圖、簡單圖、多重圖、偽圖：上面介紹過了。

點、邊的特性

1. 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的方法。

2. 度（degree）：一個點的度，等於連接這個點的邊數；一個圖的度，等於這個圖中度數最大的點的度。在有向圖中，還可以分為出度和入度（in-degree, out-degree），分別等於將某一點作為起點的邊數、和作為終點的邊數。（特別地，在偽圖當中，自環的度數要算兩次。）

點、邊之間的關係

1. 相鄰（adjacent）：在無向圖當中，若有一條邊連接兩點 $\{v_i, v_j\}$，稱這兩點相鄰。若有一個點被兩個邊 e_i, e_j 連接，我們稱這兩個邊相鄰。

2. 指向（consecutive）：任何一條有向圖的邊都從其起點「指向」終點。另外，如果我們稱點 v_i 指向 v_j，代表存在 (v_i, v_j) 這一條邊。為了方便，這個詞也可用在無向圖上：v_i 指向 v_j 若且唯若 v_i, v_j 相鄰。

3. 路徑（path）：一條由點 A 到點 B 的路徑，記為 $P(A,B)$，指的是一個點邊交錯的序列 $(A=v_0,e_0,v_1,e_1,\cdots,v_{k-1},e_{k-1},v_k=B)$，其中 $e_i=\{v_i,v_{i+1}\}$（或 (v_i,v_{i+1})，如果是有向圖）。也可以看成是一連串經由指向關係連結起來的點。

4. 行跡、迴路（trace, circuit）：如果一條路徑中 $e_0,e_1,\cdots,e_{k-1}$ 兩兩相異，稱這個路徑為行跡；行跡的起終點相同，則稱為迴路。

5. 簡單路徑、環（track, cycle）：如果一條路徑中 $v_0,v_1,\cdots,v_k$，除了 v_0,v_k 以外皆兩兩相異，稱這個路徑為「簡單路徑」。簡單路徑的起終點相同，則稱為環。
（上面五個名詞的英文名稱在文獻中並不統一，有時候會有混淆的情形發生，連帶導致中文也有一些混亂，實際遇到的時候請多加注意。）

6. 連通（connected）：在無向圖中，如果 v_i 到 v_j 的路徑存在（則 v_j 到 v_i 的路徑存在），稱 v_i 和 v_j 連通。如果一群點兩兩連通，則稱這一群點連通。

特殊圖

有一些符合某些條件的圖會有一些值得我們利用的性質，在此稱為特殊圖。以下列舉較為常見的，至於其性質會在之後一一介紹：

1. 連通圖（connected graph）：把所有邊換成無向邊後，任兩點都連通的圖稱為連通圖。

2. 樹（tree）：一個無向、連通且沒有環的簡單圖稱為樹。

3. 森林（forest）：一個圖由數棵互不連通的樹組合而成，稱它是一個森林。

4. 完全圖（complete graph）：任兩點都相鄰的簡單無向圖稱為完全圖。包含 n 個點的完全圖記為 K_n。

5. 有向無環圖（directed acyclic graph, DAG）：顧名思義，沒有環的有向圖稱為有向無環圖。

6. 二分圖（bipartite graph）：若可以把一個無向圖的所有點分成兩個不相交的集合 U 和 V，使得在同一個集合內的任意兩點都不相鄰，稱這個圖為二分圖。

圖之間的關係

有時候，兩張圖會有一定的相關性。以下列舉一些例子：

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）：對於圖 $G=(V,E)$，令集合 K 代表所有 V 的二元子集（或是二元有序對，如果是有向圖），則 G 和 $H=(V,K\setminus E)$ 互為補圖。

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長得一樣。

4. 生成樹（spanning tree）：如果兩張無向圖 G、T 的點集相同，滿足 T 是一棵樹、T 是 G 的子圖，則稱 T 是 G 的生成樹。

5. 元件（component）：又稱為「分量」，如果一張圖中某極大的子圖符合某個條件，我們稱這群點是一個元件。例如「連通元件」即是一個極大的連通子圖。

2 儲存與遍歷

2.1 圖的儲存方法

瞭解了圖的基本術語，還要知道要用什麼樣的資料結構才能儲存一張圖。以下介紹幾種儲存圖的方式：

1. 鄰接矩陣（adjacency matrix）：開一個 $V \times V$ 的資料結構 M（通常是陣列，無邊權時可用 bitset），如果點 a 指向點 b，則 $M_{a,b}$ 為 1，否則為 0。空間複雜度 $O(V^2)$，加、刪邊時間複雜度 $O(1)$。如果有邊權，則相鄰時可記邊權，不相鄰則記一個很大的數。

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 $O(V)$。

3. 前向星（forward star）：把鄰接串列裡面的 V 個資料結構全部頭尾相接，變成一個陣列，然後另外開一個長度為 V 的資料結構記錄每個點的資訊從陣列的第幾個元素開始。空間複雜度 $O(V + E)$（但是常數比鄰接串列小），加、刪邊時間複雜度 $O(V + E)$。

通常我們會在稀疏圖用鄰接串列，在稠密圖用鄰接矩陣。至於不用前向星的原因，是在建圖的時候，如果沒有按照順序輸入，一種方法是先建好鄰接串列再換過去（多此一舉），一種是先用 $O(E \log E)$ 把邊排序（浪費時間）。

2.2 圖的遍歷方法

當我們得到一張圖，當然要從這張圖裡面得到一些資訊。要怎麼把圖中的所包含的資訊完整的看過一遍（稱為遍歷）呢？於是有以下兩種最常見的方法：

深度優先搜索（Depth-First Search, DFS）

這是最直觀的一種遍歷圖的方法，是建立在指向的關係之上，使用遞迴實作。從一個點開始（稱為「造訪」），把這個點標記為已經造訪過（開一個陣列之類的），然後看這個

點指向哪些點，並立刻遞迴造訪這些點；如果已經把所有指向的點都走完了，就結束。

因為建立於指向的關係，DFS 過程經過的所有邊會形成一棵生成樹，稱為 DFS Tree。因為此演算法優先拓展 DFS Tree 的深度，因而得名。

另外，如果原圖並不連通，或者是有向圖而不強連通（之後會提到），一次 DFS 不一定能遍歷完整個圖，這時候就需要再從還沒造訪過的點再次 DFS，直到所有點都走過為止。以下是虛擬碼：

Algorithm 1: Depth-First Search

```
1 procedure DFS(vertex v)
2     label v as visited
3     for each vertex w adjacent / consecutive to v
4         if w is not labeled as visited
5             DFS(w)
6         end if
7     end for
8 end procedure
```

時間複雜度和存圖的複雜度一樣，額外空間複雜度 $O(V)$。因為 DFS 很簡單，而且有很多很好的性質，所以常常被我們使用。之後我們會一一提到。

順帶一提，由於 DFS 是以遞迴實作的，在自己的電腦上跑的時候，如果遞迴太深會產生堆疊溢位（stack overflow）的情況，需要對 linker 動一點手腳才能避免（在連結器參數加上-Wl,--stack,536870912）。不過如果是在比賽或者 online judge 之類的話就不用擔心會發生這種事。

另外附帶一個 C++11 小技巧：如果要把 STL 容器中的每個元素讀過一遍，for 迴圈可以直接寫 for(auto& a : v) {……}（假設容器名稱為 v）。如果用 vector 實作鄰接串列的話，這個技巧非常有用。

廣度優先搜索（Breadth-First Search, BFS）

這種遍歷方法也是建立於指向的關係。它先把由原點經由一條邊可以到的點遍歷完，接著遍歷原點經由兩條邊可以到的點，以此類推，直到遍歷完所有的點。

至於實作方法，則是每造訪一個點，就把所有相鄰的點丟進 queue 裡，造訪完畢後，接著造訪從 queue 裡面拿出來的點。容易看出它造訪的順序符合上列敘述。

由其遍歷順序可知，在邊權都相等且為正的情況，BFS 能找出一個點到其它所有點的最短路徑。這也是 BFS 通常的用途，除此之外 BFS 很少用到。

時空複雜度都等於 DFS，小心實作的時候千萬不要讓空間複雜度變成 $O(V+E)$。

2.3 習題

1. (Codeforces 676D) 有一個矩形的移動迷宮，由 $n \times m$ 個正方型房間組成。每一個房間都有 0~4 個門，朝向前、後、左、右之中的幾個方向。你現在在 (x_S, y_S)（第 x_S 排第 y_S 列的房間），每一分鐘你可以按下一個機關使整個移動迷宮每一個房間都順時鐘旋轉 90 度（門連帶跟著朝向不同的方向），或者移動到相鄰的房間（前提是這兩個房間都要有相對應方向的門）。求你最短需要花多少時間才能移動到 (x_E, y_E)，或者根本不可能到達。$n, m \leq 1000$。

2. (TIOJ 1481) 有 $n \leq 2000$ 個城市和 $k \leq 20000$ 條雙向航線。求一種把所有航線從 1 編號到 k 的方法，使得對於任意一個至少連接兩個航線的城市，它連接的所有航線編號的最大公因數是 1，或是不存在這種方法。（其實可以到 $n, k \leq 10^6$。）

3 最小生成樹

最小生成樹（minimum spanning tree）是一個非常經典的圖論題，幾乎每次筆試題都會出現，上機題中偶爾也會見到它的蹤影。大意就是對於一個有邊權的連通無向圖，求一個邊權總和最小的生成樹。

這個問題有兩個經典的演算法。

3.1 Kruskal's algorithm

我們將原圖中連接一部分點的生成樹，稱為部分生成樹。首先觀察幾個特性：

1. 一個單獨的點，它自己就是一個最小部分生成樹。

2. 兩個最小部分生成樹，要加一邊合併成一個部分生成樹時，加上連接這兩個生成樹的邊中邊權最小的那個邊會最好。

3. 三個部分最小生成樹要合併，先合併要加的邊最小的那兩個最小生成樹會最好。

綜合上述三點，我們發現一個貪婪準則：從邊權最小的邊開始加，如果這個邊連接同一個部分生成樹就不要選。如此便能逐步建構出整個圖的最小生成樹。

而從邊權最小的邊開始加，可以簡單的將邊按照權重排序後一個一個取，複雜度 $O(E \log E) = O(E \log V)$。但是要怎麼判斷這個邊連接的是不是同一個部分生成樹呢？這裡插播一個很常使用且簡單易寫的資料結構：

3.2 並查集（Disjoint-set）

假如現在有 n 個元素，每一個元素都恰屬於一個集合（稱這些集合為「互斥集」）。一開始每個集合都只有一個元素。我們希望它提供兩個功能：

1. 合併（Union）：把兩個集合合併成一個。

2. 查詢（Find）：查詢某個元素現在在哪個集合裡。

要如何實作呢？很直觀的方法是：對每個集合編號，並且記錄每一個元素在哪一個集合裡。集合 A 和 B 合併的時候，找到所有 B 的元素，把它們全部換到 A 裡面。如此，合併複雜度是 $O(n)$、查詢複雜度是 $O(1)$。

但是合併的複雜度看起來很糟糕，如果進行 $n-1$ 次合併，總複雜度會變成 $O(n^2)$。

要改進合併的複雜度，可以換一個方法：每一個集合 S 選定一個「代表元素 $R(S)$」，記錄它的「上級」為自己。集合 A 和 B 合併的時候，就把 $R(B)$ 的上級記錄為 $R(A)$，相當於把 B 併進 A。查詢時，一直追溯到最上級就知道它屬於哪個集合了。這個結構相當於一個森林。如此，合併複雜度是 $O(1)$，但查詢複雜度卻變成了 $O(n)$。

但仔細看一下這個方法，其實存在兩種簡單的優化。一個是對每個集合記錄它的元素個數，合併時永遠把小集合併進大集合。另一個是，在查詢的時候，把每一個經過的元素的上級都直接設為代表元素。

這兩個改進方法分別稱為按秩合併（union by rank）和路徑壓縮（path compression）。容易證明，只套用前者時，查詢複雜度是 $O(\log n)$；另外可以證明，只套用後者時，查詢複雜度是 $O(\log_{2+f/n} n)$，其中 f 是查詢的次數。

重要的是，當兩者一起套用時，查詢複雜度是 $O(\alpha(n))$，$\alpha(x)$ 指的是阿克曼函數 $A(x,x)$ 的反函數。這個函數在所有實用範圍的數字（$\leq 2^{2^{2^{16}}} - 3$）內都不大於 4，所以基本上都把它當作常數。

回到 Kruskal's algorithm，將每一個部分最小生成樹當成一個集合，則每次看一條邊時，只要查詢並查集，就可以知道它們屬不屬於同一個部分最小生成樹，加邊時則把兩個集合合併。於是，總複雜度 $O(E \log V)$。

3.3 Prim's algorithm

仔細觀察另一個特性：如果原圖點集為 V，一個部分最小生成樹的點集為 S，則加入了連接 S 和 $V \setminus S$ 的邊中權重最小的那一個邊之後，依然是個部分最小生成樹。

如此，我們得到另一種貪婪演算法：隨便挑一個起始點，將其「最短距離」設為 0，其它所有點則設為 ∞。每次找不在當前部分最小生成樹的點中最短距離最小的那個點 u（以及連接它的邊）加入，並且把所有與 u 相鄰的點的最短距離更新。重覆直到所有點都被加入。

這個演算法需要 E 次減少集合中某元素的值、V 次把最小的元素從集合中拿掉。用 priority_queue 可以達到 $O((V+E)\log E) = O(E\log V)$ 的複雜度；改用費波那契堆（fibonacci heap）可以達到 $O(E + V\log V)$。但是因為它常數比較大，而費波那契堆幾乎不可能手刻出來，所以平常用 Kruskal's 就可以了。

3.4 習題

這裡也會順便附帶一些並查集的習題。

1. (TIOJ 1211) 裸最小生成樹。

2. (TIOJ 1220) 有 $n \leq 10^5$ 個人，其中 $m \leq 2 \times 10^6$ 對人能互相連絡。求最多可以把這些人分成幾群，使得對於任兩個不在同一群的人都能互相連絡。

3. (TIOJ 1192) 有 $n \leq 1000$ 個門鎖，你想要做幾把長度為 $L \leq 5 \times 10^5$ 的鑰匙，每把鑰匙每單位的長度都可以選擇是凹或凸。每個門鎖都有 $q_i \leq 1000$ 個限制，每個限制要求鑰匙上某一個位置必須是凹的或凸的。一把鑰匙必須滿足一個門鎖要求的所有條件才能打開那個門鎖。求你能不能設計出兩把不同的鑰匙，使得每個門鎖至少可以被其中一把打開。

4. (TIOJ 1795) 有一個邊權非 0 即 1，$n \leq 10^5$ 個點、$m \leq 3 \times 10^5$ 條邊的無向簡單圖，問是否存在一個總邊權為 k 的生成樹。

4 樹

樹沒有環，所以相較於有環的圖簡單很多。同時，也多出了很多性質可以利用。樹的幾個性質列舉如下：

1. 連通且邊數等於點數減一。

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。

以上四個性質是等價的。這也是判斷一個圖是不是樹的方法。

4.1 基本性質

在此先介紹幾個基本的、專屬於樹的術語（樹語？）：

1. 根（root）：在一個樹上可以選定一個點做為「根」。有限定根的樹稱為「有根樹」。

2. 葉（leaf）：無根樹中，度數為 1 的點稱為「葉」；有根樹中，度數為 1 且非根的點稱為「葉」，但只有一個點的情況，根也是葉。

3. 子樹（subtree）：移除一個點之後，原樹會被切割成很多棵樹，稱為子樹。在有根樹中，子樹通常指遠離根的那些樹。

4. 父、子關係（parent, child）：有根樹中，如果 v_i, v_j 相鄰，且 v_i 比較接近根，稱 v_i 是 v_j 的父親、v_j 是 v_i 的兒子。

5. 祖先、後代關係（ancestor, descendant）：一個點的小孩、小孩的小孩……等等皆稱為其「後代」；一個點的父親、父親的父親……等等皆稱為其「祖先」。

6. 距離（distance）：兩點間的距離是它們之間路徑的邊數，或者是路徑上邊權的加總。

7. 深度（depth）：有根樹中，一個點的深度是它到根的距離稱為深度。

8. 高度（height）：有根樹中，一個點到與它距離最大的葉的距離稱為高度。根的高度稱為這整顆樹的高度。

判斷父子關係、祖先後代關係可以用一次的 DFS 預處理，即可達到 $O(1)$ 查詢。父子關係只要對每個點記錄它的父親是誰即可；祖先後代關係則可以在 DFS 時記錄進入該點和離開該點的時間，當點 A 的時間區間被點 B 完全覆蓋，則 A 是 B 的後代。

距離、深度的部分，可用簡單的 DFS 求出一點到所有點的距離。高度的話，因為葉子的高度是 0，而一個點的高度等於所有子樹中高度加連向子樹的邊權的最大值，所以照樣可以用 DFS 解決。

順帶一提，因為樹沒有環，所以在 DFS 時不需要記錄一個點有沒有被造訪過，只要知道父親是哪個點，不要往回走即可。

4.2 直徑、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。

要如何找出一棵樹的直徑呢？有一個很簡單的方法：先從任意一個點 DFS 到深度最深的點 v，再從 v 點 DFS 到最遠的點 u，則 v、u 的簡單路徑即為直徑。這個方法的正確性不難證，可以自己嘗試看看。

另外也可以證明圓心必定在直徑上面，因此找圓心也可以直接透過 DFS 解決。

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

顧名思義，有根樹中兩點的「最低共同祖先」，即是兩點共同的所有祖先中，深度最深的那個點。以下介紹兩種計算 LCA 的方法：

Tarjan's LCA Algorithm

這個演算法目的是求出樹中數個指定點對的最低共同祖先。

想法是利用並查集，在 DFS 過程中，將「已造訪完所有孩子」的點和其父親合併起來。並且記錄每一個集合的根（因為每個集合會形成一棵樹）。仔細觀察就會發現，一個點 u 剛造訪完所有孩子，尚未和父親合併時，它和另一個已造訪完所有孩子的點 v 的 LCA，會等於 v 所在集合的根。

Algorithm 2: Tarjan's LCA Algorithm

```
disjoint_set djs //包含所有點
procedure LCA(vertex u)
    Root[u] = u
    for each vertex v in u.children
        LCA(v)
        djs.Union(u, v)
        Root[djs.Find(u)] = u
    end for
    label u as completed
    for each vertex v such that {u, v} in queries
        if v is labeled as completed
            result[u][v] = result[v][u] = Root[djs.Find(v)]
        end if
    end for
end procedure
```

複雜度是 $O((V+Q)\alpha(V))$，其中 Q 是欲查詢的點對數量。這個演算法不常用，看看就好。

二分搜

觀察一下，容易發現求出 LCA 等價於求出令「A 的上 k 輩祖先是 B 的祖先」成立的最小 k 值，而這件事具有單調性，因此可以套用二分搜。但是要二分搜的話，就需要知道能快速知道 A 的上 k 輩祖先是誰，以及查詢祖先關係。

因此，在 DFS 預處理的時候，除了紀錄每個點進入和離開的時刻（以利查詢祖先關係），另外對每一個點紀錄它的上 1 輩、上 2 輩、上 4 輩、上 8 輩……的祖先是誰（這個技巧我們稱作「倍增法」）；二分搜的範圍則取 $[1, 2^{\lceil \log_2 V \rceil}]$。於是在二分搜的時候，如果發現 $k > mid$，則將 A 換成 A 的上 mid 輩，由於一開始範圍是 2 的冪次的關係，每次的判斷可以直接查表，達到 $O(1)$ 的判斷時間。

這樣一來，預處理的複雜度是 $O(V \log V)$，之後每一次查詢可以達到 $O(\log V)$。

樹其實還有更多複雜的性質，在下次圖論課當中會再提到。

4.4 習題

1. (IOI 2013)(TIOJ 1838) 有 $n \leq 10^5$ 個水洞，由 m 條水道連結，通過第 i 條水道都需要 T_i 天。這些水洞和水道形成很多棵樹。你必須要挖出 $n-m-1$ 條通過時間為 L 天的水道，使得所有水洞之間都可以通行，並且讓任兩個水洞間最久的通行時間最短。求該通行時間。

2. (TIOJ 1163) 有 $V \leq 30000$ 個地點和 $E \leq 50000$ 條雙向道路連接這些地點，每條雙向道路都在整修，第 i 條需要等 d_i 天之後才能使用。$Q \leq 50000$ 次詢問兩點之間要等到哪一天才能通行。

5 有向無環圖

有向無環圖可以當作樹的有向圖版本。有向無環圖也可以當作是 DP 的圖版本：每個點代表一個子問題、每個邊代表一種轉移，整個 DP 則是在有向無環圖上找到一種走法、一條最短或最長路。

因為有向無環圖很像樹，它也可以仿造樹定義「深度」：從該點到起點的最遠距離。算法也和樹差不多。

5.1 拓樸排序

一張有向圖的拓樸排序，就是一種所有點的排列方法 $\{v_0, v_1, \cdots, v_{n-1}\}$，使得對於兩點 v_i, v_j 且 $i \leq j$，不存在 (v_j, v_i) 這條邊。

顯然地，一張有向圖是有向無環圖若且唯若它存在拓樸排序，因此找到拓樸排序也可以當作判定有向無環圖的方法。

拓樸排序可能有很多種，若要找到任意一種，以下列出兩種方法：

1. Greedy：每次都找入度為 0 的點，並把它移除。如果過程中找不到入度為 0 的點，代表它不存在拓樸排序；如果所有的點都移除掉了，則拓樸排序就是點移除的順序。

2. DFS：直接對整張圖 DFS，紀錄各點的離開時間。如果 DFS 過程中造訪到一點 v 時對於某一條邊 (v, u) 滿足 u 已進入但尚未離開，代表圖存在環；否則各點按照離開時間後到前排序，即是拓樸排序。

時間複雜度均為一次遍歷的時間。

另外，拓樸排序也可以當作是 DP 的計算順序：在 Bottom-up DP 當中，一種合理的計算順序即是一種拓樸排序。

5.2 習題

1. (2016 入營考 pC)(Codeforces 510C)(UVa 200) 有 $n \leq 100$ 個字串，每個字串都由不超過 100 個小寫英文字母組成。求出一種排列英文字母的方法，使得所有字串是按照字典序排列的（或者不存在這種排列）。

2. (TIOJ 1092) 在一張只有一個起點一個終點且連通的有向無環圖當中，由 A 從起點開始，接著 B、A 輪流走到對方所在點指向的任一點。求誰有必勝走法。$V \leq 10^4, E \leq 10^5$。

6 單點源最短路徑

一個路徑的長度，即是這條路徑所有邊的權重和。單點源最短路徑問題，即是求出一個圖當中，一個點到所有點中，長度最短的簡單路徑長度是多少。

可以觀察到一個性質：由一個點到所有點的最短路徑經過的所有邊會形成原圖的一個生成樹，稱為最短路徑樹。因此這個問題也可以等價成：求出一個圖上某點起始的最短路徑樹。

6.1 鬆弛

前面有提到在所有邊的邊權相等的時候，可以用 BFS 求出最短路徑，但是在邊權不相等的時候卻行不通，原因是邊權會導致邊數和距離不呈正比。因此需要找到一個方法解決「多條邊距離比少條邊還短」這件事。因此，「鬆弛」（relaxation）的概念成了求單點源最短路徑的必要想法。

鬆弛可以想像成「抄捷徑」：假設現在已知一個路徑 $P(S,T)$，在其中經過了 a,b 兩點（也可以是起終點），在這條路徑上由 a 到 b 的距離為 L，而若可以找到一條路徑 $P(a,b)$ 的長度小於 L，那可以把原本路徑 a,b 間的一段以 $P(a,b)$ 取代，於是路徑變得更短了。這個動作就稱為「鬆弛」。

由於最短路徑樹的性質，我們常用一個邊來進行鬆弛：令存在一條邊 (a,b)，邊權為 W_{ab}，若當前由起點到 a,b 的路徑長分別為 L_a, L_b（不一定是最短路徑），在 $L_a + W_{ab} < L_b$ 的時候，即可更新起點到 b 的路徑。

另外，如果要求出最短路徑樹的話，可以簡單地對每一個點記錄「這個點在路徑樹上的前一個點」是誰，演算法結束之後便可以找到路徑。

6.2 Bellman-Ford Algorithm

理解「鬆弛」的概念後，我們便很容易得到一個單點源最短路徑的演算法。一開始起點的最短路徑長為 0，其它點因為還沒有找到路徑，暫且設為無限（實作上可用一個很大的數代替）。不斷的窮舉圖上的所有邊，每一個邊都拿來進行鬆弛。

可以發現，當窮舉完 k 次後，如果起點到某一點的最短路徑邊數少於 k 的話，那這個路徑就已經被求出來了。由於最短路徑最多只能包含 $V-1$ 條邊，所以重複窮舉 $V-1$ 次邊之後，所有的最短路徑都被求出來了。

但是有個例外：如果在圖上存在一個環，其邊權加總小於 0（稱為「負環」），這時再窮舉第 V 次邊的話，會發現仍然可以鬆弛。這時就只能得到「這個圖有負環」的結論。事實上，在有負環的圖上，最短路徑是個 NP-Complete 問題，目前沒有多項式時間的解法。

此演算法時間複雜度為 $O(VE)$，額外空間複雜度 $O(V)$（記錄起點到各點的距離）。

這個演算法還有一個優化：一個點必須被鬆弛過才有需要拿來鬆弛其它點。最差複雜度同 Bellman-Ford，但是運氣好的時候可以到 $O(V+E)$。

6.3 Dijkstra's Shortest Path Algorithm

Bellman-Ford algorithm 把所有可能的最短路徑都窮舉過一次，所以它可以處理負權邊。但是在邊權重都非負的圖上，有一個很好用的性質：假若有一個部分完成的最短路徑樹，那麼當前離根最近且不在樹上的點，必定會在最短路徑樹上。

由這個特性，容易聯想到之前提到求最小生成樹的 Prim's algorithm，是一個點一個點完成最小生成樹；同樣地，也可以用相同的方法，一個點一個點完成最短路徑樹。只要把離當前樹的距離，改成離起點的距離即可。也可看成是一種對鬆弛順序 greedy 的法則。

Algorithm 3: Dijkstra's Shortest Path Algorithm

```
1  array dis[V] = {infinity}, parent[V] //最短路徑樹上的前一點
2  array in_tree[V] = {false} //記錄該點是否已在最短路徑樹中
3  procedure dijkstra(int start)
4      priority_queue<pair<dis, vertex>> pq //最小堆
5      parent[start] = start
6      pq.push({dis[start] = 0, start})
7      for i = 1 to V //每次加入一點
8          now = −1
9          while pq is not empty and in_tree[now = pq.top().vertex]
10             pq.pop()
11         end while
12         if (now == −1)
13             return //圖不連通
14         end if
15         in_tree[now] = true
16         for each vertex next adjacent to now
17             if not in_tree[next] and dis[now] + weight(now, next) < dis[next]
18                 pq.push({dis[next] = dis[now] + weight(now, next), next})
19             end if
20         end for
21     end for
22 end procedure
```

複雜度也和 Prim's algorithm 相同，通常以 priority_queue 實作，複雜度 $O(E \log V)$。

7 全點對最短路徑

有時候我們會需要一次知道所有點對之間的最短路徑。當然最簡單的方法是將單點源最短路徑做 V 次，這樣在非負權圖上複雜度 $O(VE \log V)$，有負權的圖則有 $O(V^2E)$。

7.1 Floyd-Warshall Algorithm

看起來負權圖的複雜度有些差強人意。不過這時候我們想到了一個老技巧：動態規劃。

因為要求的是所有點對，所以不像單點源一樣被綁在一個初始點上面。一個路徑除了起點終點，剩下的都是中繼點。於是，將中繼點一個一個考慮進來，可以把「由 i 點中途經過前 k 點抵達 j 點的最短路徑」作為 DP 狀態。很容易得到轉移式：

$$dp_{k,i,j} = \min(dp_{k-1,i,k} + dp_{k-1,k,j}, dp_{k-1,i,j})$$

由於計算順序的關係，可以直接使用滾動 DP，是一個非常好寫的演算法。如果需要記錄路徑，把中繼點存下來，事後用遞迴方式讀取即可。

這個演算法也可以判斷負環，直接看有沒有任何 $dp_{k,i,i} < 0$ 即可。

Algorithm 4: Floyd-Warshall Algorithm

```
1  array weight[V][V] //鄰接矩陣
2  array dis[V][V], mid[V][V] = {−1} //結果和中繼點
3  procedure floyd_warshall(int start)
4      copy weight to dis
5      for i = 0 to V − 1
6          dis[i][i] = 0
7      end for
8      for k = 0 to V − 1
9          for i = 0 to V − 1
10             for j = 0 to V − 1
11                 if dis[i][j] > dis[i][k] + dis[k][j]
12                     dis[i][j] = dis[i][k] + dis[k][j]
13                     mid[i][j] = k
14                 end if
15             end for
16         end for
17     end for
18 end procedure
```

時間複雜度 $O(V^3)$，空間複雜度 $O(V^2)$。從虛擬碼可以看出這個演算法的常數極小（快的 judge 可以兩秒跑完 $V = 1000$）。

7.2 Johnson's Algorithm

Floyd-Warshall 很好寫，但是看起來在稀疏圖上，跑 V 次 Dijkstra 的 $O(VE\log V)$ 會更好一點，然而 Dijkstra 只能用在非負權圖上。於是，這個演算法的目的就是把帶負權的圖變成正權圖。

所使用的方法，是為每一個點加上一個點權 $H(i)$，並且令所有邊的新權重，等於原權重加上起點的點權、減掉終點的點權。

不難證明這個調整邊權的作法有兩個性質：

1. 同一點對 (S,T) 的最短路徑 $P(A,B)$ 並不會改變，但權重會加上 $H(S)-H(T)$。

2. 任何環的權重都不改變。

因此這是一個良好的調整邊權作法。那要如何讓每個邊權都變為非負呢？我們先列出式子：

$$w(i,j)+H(i)-H(j)\geq 0,\forall(i,j)\in E$$

然後移個項：

$$H(i)+w(i,j)\geq H(j)$$

有沒有似曾相識？還記得我們之前做鬆弛的條件是 $d(i)+w(i,j)<d(j)$，若令 $H(i)=d(i)$，上式代表的是「不能再進行鬆弛」，也就是說任一組單點源最短路徑的解都是符合條件的點權。至此這個演算法大致成形：對任一點做 Bellman-Ford（順便判斷有沒有負環），得到點權之後，用調整完的邊權做 V 次 Dijkstra，時間複雜度 $O(VE\log V)$（如果用費波納契堆是 $O(V^2\log V+VE)$），在較稀疏的圖上表現會比 Floyd-Warshall 好。

然而還有一個小問題。前面有提過，當圖為有向圖的時候，一次遍歷有可能無法遍歷完全圖，也就是有可能會發生有點沒有點權的狀況（因為找不到最短路）。這個問題很好解決：新增一個點，讓它連到所有點，並且邊權都是 0，再對這個點做 Bellman-Ford 即可。

7.3 習題

1. (2015 北市賽 pE)(No judge) 有一個 $n\times n$（$n\leq 20$）的矩形戰場，每個 1×1 格子都有一定數量的敵兵。有 $Q\leq N^4$ 次任務，在每一次任務中你必須要從某一個格子移動到另一個格子（只能上下左右移動），如果你過程中和某些敵兵在同一個格子，視為你遇到了這些敵兵。行動前你可以發射一枚炸彈使某一個格子的敵兵在這次任務中暫時消失。求你每次任務至少要遇到多少敵兵。

2. (TIOJ 1641) 有 $N\leq 10^4$ 個城市、$M\leq 2\times 10^5$ 條連接兩城市的單向道。你有一單位的貨物要從 A 運到 B，但是每經過一條道路後，你必須要多帶原先 C_i 倍的貨物，其中 C_i 是該條道路的「方便率」。求到達終點時你最少會有多少單位的貨物。

3. (TIOJ 1096) 現在有 $n \leq 100$ 個捷運站，由一些連接兩站的單向捷運線連接，每條捷運線都有搭乘時間。求最短需要多少時間才能從某站出發，搭至少一條捷運線之後再回到原站。

4. (TIOJ 1028) 有 $n \leq 13$ 個景點，景點和景點間有一些雙向道路連接，每條都有它的長度。你現在想要從某點出發去其中的一些景點（當然途中可以經過你沒有要去的景點，也可以重複經過道路或景點），求一個最短的走法可以經過所有的景點。如果有很多種，求字典序最小的那種。

5. (Codeforces 543B) 有 $N \leq 3000$ 個城市、$M \leq \frac{n(n-1)}{2}$ 條連接兩城市的雙向道，每條道路有自己的長度。保證任兩城市都能互相藉由道路往來。現在你要摧毀一些道路，但必須使由 s_1 到 t_1 的距離不超過 l_1，且由 s_2 到 t_2 的距離不超過 l_2。求你最多可以摧毀幾條道路，或沒有任何方法滿足條件。