圖論

edisonhello

2017 年 9 月 22 日

資訊的圖不是像數學那種有座標軸有角度的圖。一個資訊的圖（Graph）G 通常包含著點（Vertex）V 跟邊（Edge）E。我們通常可以把一個邊 e 表示成 $e = (u, v), u, v \in V$。

1 初見圖

1.1 分類、特徵

1. 有向圖（Direct graph）：代表這張圖的邊 $(u, v) \neq (v, u)$，且其中一個會是起點，另一個是終點。
2. 無向圖（Undirect graph）：跟上面的相反，每個邊的兩個方向都是可通行的，沒有起點終點之分。
3. 簡單圖（Simple graph）：當一個圖的邊沒有重邊（存在兩個以上的邊 e_i, e_j 且 $e_i = e_j$），且沒有自環（邊 $e_i = (u, v), u = v$）時，這個圖就叫簡單圖。
4. 完全圖（Complete graph）：當一個圖的所有點對 (u, v)，都存在一個邊 $e_i = (u, v)$，那這張圖就是完全圖。
5. 連通圖（Connected graph）：把所有邊變成無向邊之後，如果任兩個點都可以經由一些邊連接起來，那這張圖就是連通圖。
6. 二分圖（Bipartite graph）：如果能把這張圖的所有點分成兩部份，同一部份的點互沒有邊相接，那這張圖就是二分圖。

1.2 一些術語

1. 度數（degree）：指一個點連接著的邊數。如果是有向圖的話，會有入度（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。
2. 相鄰（adjacent）：無向圖中，兩個點相鄰若且唯若存在一個邊連接著這兩點；兩個邊相鄰若且唯若存在一個點連著這兩條邊。

3. 路徑（path）：指從一個點經由一串相鄰的邊到達另一個點。
4. 行跡（trace）：如果路徑經過的邊沒有重複，那這個路徑就是一個行跡。
5. 迴路（circuit）：如果行跡的起點跟終點是同一個點，那這就是一個迴路。
6. 簡單路徑（track）：如果路徑經過的點沒有重複，那這個路徑就是一個簡單路徑。
7. 環（cycle）：如果簡單路徑的起點跟終點一樣，那這個簡單路徑就是一個環。（所以起點跟終點是唯一重複的點）
8. 連通（connected）：如果兩個點之間有路徑，那這兩個點就連通。
9. 樹（tree）：一個無環無向連通圖就是一顆樹。
10. 有向無環圖（Directed Acyclic Graph）：看名字 (?)
11. 子圖（Subgraph）：如果有一個圖 G' 且 $V(G') \subseteq V(G) \wedge E(G') \subseteq E(G)$，那我們就稱 G' 是 G 的子圖。
12. 補圖（Complement graph）：簡單來講，若圖 G 與圖 H 互為補圖，則 G 跟 H 都有相同的點，且在 G 中的邊都不存在 H 中，但在 G 不存在的邊都存在於 H 中。
13. 同構（isomorphic）：如果兩個圖的點集 $V(G)$ 與 $V(H)$ 存在一種一一對應關係 f 且 $\forall(u,v) \in \{(u,v) \mid (u,v) \in E(G)\} \quad \exists(f(u), f(v)) \in E(H)$，則這兩個圖稱為同構。
14. 生成樹（Spanning tree）：如果有兩個圖 G,H 且 H 是一棵樹且 H 是 G 的子圖且 $V(G) = V(H)$，那 H 就是 G 的生成樹。

2 儲存與遍歷

2.1 儲存

要把圖存起來主要會用兩種方法。

1. 鄰接矩陣（Adjacency matrix）：開一個 $V \times V$ 的陣列 G，如果 u,v 間有邊，就把 G[u][v] 標記起來或改成這個邊的權值之類的。空間複雜度 $O(V^2)$，加邊刪邊查邊複雜度 $O(1)$。
2. 鄰接串列（Adjacency list）：開 V 個 vector 之類的東西，每個 vector 記這個點有跟誰相鄰，或是邊的其他資訊。空間複雜度 $O(V+E)$，加邊複雜度 $O(1)$，刪邊查邊複雜度 $O(E)$。如果開成 set 的話加邊刪邊查邊複雜度都是 $O(\lg E)$。

2.2 DFS and BFS

2.2.1 Depth-First Search 深度優先搜索

隨便從一個點開始進行 DFS，規則是遍歷與目前這個點連接的所有點，然後遞迴做下去。如果在有環的圖的話，通常為了避免走到重複的點，所以會開一個陣列來紀錄是否走過這個點。如果走過了就直接 return。DFS 的過程中因為每個點只會走一次，所以走的時候可以順便產生一棵生成樹。所以我們可以在走這棵樹的時候順便維護一些東西，例如子樹的大小之類的。

Algorithm 1: DFS

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

上面的 DFS 假設圖是以鄰接串列儲存的。時間複雜度 $O(V)$，空間複雜度 $O(V)$（紀錄是否走過的陣列）。

2.2.2 Breadth-First Search 廣度優先搜索

相較於 DFS，BFS 會優先把相鄰的節點都走完。方法是開一個 Queue，然後把所有相鄰的節點丟進去，每次都從 Queue 拿出東西來做事。因為這個性質，所以可以拿來解迷宮的最短逃生路線等。

Algorithm 2: BFS

```
1  void bfs(int s){
2      queue<int> q; q.push(s); u[s]=1;
3      while(q.size()){
4          for(int i:G[q.front()])if(!u[i]){
5              u[i]=1;
6              q.push(i);
7          }
8          q.pop();
9      }
10 }
```

假設圖也是以鄰接串列儲存，時空複雜度跟 DFS 都一樣。另外，如果以 Stack 模仿 BFS 來實作 DFS 的話空間的常數會稍微小一點。

2.3 例題們

1. TIOJ 1085：三維的迷宮。

2. TIOJ 1481：把邊編號，使得任意有兩個以上的邊的節點的所有邊的公因數是 1；或是無法達成。

3 Minimum Spanning Tree 最小生成樹

大概就是要求某個有邊權的圖中，總權和最小的生成樹的邊權之類的。在講 MST 的演算法之前先來提一下等一下會用到的東西。

3.1 Disjoint Set 並查集

所謂並查集就是一個可以查詢某兩個點是否在同一個集合之內的工具。支援三種操作：查詢一個點在哪個集合內，查詢兩個點是否在同一個集合內，以及把兩個點合併到同一個集合。實作上會開一個陣列，預設是 a[i]=i，合併兩個點的時候只要合併兩個點的集合號碼；查詢只要遞迴找到 a[i]==i 的點就好了。
因為不太會用文字說明，所以直接上 Code。

Algorithm 3: Disjoint set

```
1  int djs[100];
2  void init(){
3      for(int i=0;i<100;++i)djs[i]=i;
4  }
5  int Find(int x){
6      return djs[x]==x?x:Find(djs[x]);
7  }
8  void Union(int x,int y){
9      djs[Find(x)]=Find(y);
10 }
```

然而可以發現 Find 的複雜度是 $O(n)$。所以如果一直把很多東西加到同一個集合內的話複雜度就會到 $O(n^2)$，聽起來很糟，於是可以套用一個叫路徑壓縮的優化。簡單來講，在 Find 的時候把找到的祖先覆蓋掉原本這個點的祖先就可以達到這件事。另外還有一個啟發式合併的優化，就是每次在合併時判斷，把比較小的集合加入比較大的集合，這樣要修改的量就會比較少。對於這兩個優化的複雜度，如果只用路徑壓縮，複雜度會是 $O(\log_{2+\frac{f}{n}} n)$（f 是查詢次數）；只套用前者時是 $O(\lg n)$；兩個一起使用的時候是 $O(\alpha(n))$（$\alpha(n)$ 是阿克曼函數 $ack(n,n)$ 的反函數。因為這個函數的增長速度很快，所以在 $n \leq 2^{2^{2^{2^{16}}}} - 3$ 的情況下都小於 4）。
加上優化之後大概就長這樣。

Algorithm 4: Disjoint set(optimized)

```
1  int djs[100],sz[100];
2  void init(){
3      for(int i=0;i<100;++i)djs[i]=i,sz[i]=1;
4  }
5  int Find(int x){
6      return djs[x]==x?x:djs[x]=Find(djs[x]);
7  }
8  void Union(int x,int y){
9      if(sz[y=Find(y)]>sz[x=Find(x)])swap(x,y);
10     djs[y]=x;
11     sz[y]+=x;
12 }
```

3.2 Kruskal's Algorithm

簡單來講，這個演算法滿 Greedy 的。主要內容是：將邊以邊權由小排到大，每次確認每個邊連接的兩個點是否在同一棵樹上。若是，那就跳過不管它；若非，就用這兩個點把這兩棵樹連起來。Greedy 的正確性也不難想。因為要合併兩棵生成樹的最好方法一定是取連接這兩棵樹的那些邊中邊權最小的那個邊做，所以以這種算法就可以構出對整個圖邊權最小的生成樹。對於維護哪兩個點是不是在同一棵樹這件事情的話，只要再用 Disjoint set 就可以解決了。複雜度是 $O(E \lg E)$。

3.3 Prim's Algorithm

上面那個演算法其實可以變成另一種類似的形式：如果要在一棵既有的生成樹上加一個樹外的點，那每次都取那些點中，離樹最近的點一定是最好的。一開始可以隨便取一個點，然後把跟這個點相鄰的點的距離改成他們之間的邊權，然後把這些距離加入一個 priority_queue 內，其他不相鄰的設成無限大。這樣每次只要選出 priority_queue 中最小的那個點，然後更新與這個點相鄰且未被加入生成樹的點的距離，如果有更新就加入 priority_queue 內，直到 priority_queue 是空的就做完了。總複雜度 $O((V+E)\lg E) = O(E \lg V)$。順帶一提，如果使用某種叫作費波那契堆（Fibonacci heap）的結構的話可以到 $O(E + V \lg V)$，然而幾乎不可能手刻，常數也大，而且複雜度也沒差多少，所以比賽上通常用 priority_queue 就夠了。

Algorithm 5: Prim's Algorithm

```
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;
    // (distance, pointNum)
vector<pair<int,int>> G[100]; (destination, distance)
int visit[100],d[100];
int prim(int st){
    memset(d,0x3f,sizeof(d)); d[st]=0;
    pq.push({0,st});
    int mst=0;
    while(pq.size()){
        if(pq.size() && visit[pq.top().second])pq.pop();
        if(pq.empty())break;
        int newpt=pq.top().second;
        visit[pq.top().second]=1;
        mst+=d[pq.top().second];
        for(auto i:G[newpt]){
            if(d[i.first]<=i.second)continue;
            d[i.first]=i.second;
            pq.push({d[i.first],i.first});
        }
    }
    return mst;
}
```

3.4 例題們

1. TIOJ 1211
2. TIOJ 1192：給 n（≤ 1000）個門鎖，每個門鎖有 q（≤ 1000）個條件，每個條件是在鑰匙的某個位子要是凹或凸的。問能不能打造兩把鑰匙打開全部的門。
3. TIOJ 1795：給邊權只有 0 或 1 的圖，問能不能構出總邊權為 k 的生成樹。

4 樹

總之樹有很多好玩的性質，也可以做一些好玩的事情。

1. 整個圖連通。
2. 樹中的 $E = V - 1$。
3. 任兩點存在唯一的簡單路徑。
4. 沒有環。

其實還有很多就是了。

4.1 一些樹語 (#)

1. 根（root）：在樹中一個特別的點。有根的樹就叫有根樹，會有很多事情可以做。沒有根的話可以隨便設一個 (?)，下面很多討論都是基於有根樹。
2. 葉子（leaf）：無根樹中，度數為一的點就是葉子；有根樹中，度數為一且不是根的就是葉子。不過如果只有一個點，那根也是葉子。
3. 深度（depth）：有根樹中，離根的距離就是這個點的深度。
4. 高度（height）：有根樹中，離根最遠的距離就是這棵樹的高度。
5. 父節點（parent）：兩個相鄰的節點中，離根近的節點是另一個節點的父節點，反之為其子節點（child）。
6. 祖先（ancestor）：一個節點的祖先是他的父節點與父節點的祖先。
7. 子樹（subtree）：把某個點拔掉之後，產生的很多子圖就是子樹。

葉子、深度、高度、是否為祖先之類的的關係都可以在 DFS 的 $O(n)$ 時間處理完。高度跟深度只要在 DFS 的時候傳一個參數，每往下一層就加一就可以解決了。祖先的處理比較不直覺：DFS 時紀錄進去、出來每個點的時間，如果有某個點的時間段被一個點的時間段完全包含，那這個點就是那個點的祖先。

4.2 直徑、圓心

直徑是在樹中相鄰最遠的兩個點的簡單路徑，圓心（樹心 (?)）是拆掉之後，能使子樹中最大高度最小的點。找直徑的方法是先隨便戳一個點，DFS 找到離這個點最遠的點，這就是直徑的其中一個端點。再從這個點 DFS 一次最遠的點，那這兩個點就是直徑的兩個端點。而圓心會在直徑上面。所以也可以在第二次的 DFS 順便解決。至於直徑跟圓心這兩個找法的話可以自己證明看看。

4.3 Lowest Common Ancestor 最低共同祖先

兩個點的 LCA，就是兩個點的所有共同祖先中深度最低的那個。

4.3.1 倍增法

要求出兩點的 LCA，可以在 DFS 的時候順便維護這個點的一層、兩層、四層等等的祖先。在尋找 LCA 的時候，因為如果一個點的 k 輩祖先是另一個點的祖先，那所有的 $\geq k$ 輩祖先也都會是他的祖先。所以我們可以二分搜 k，如果這個點的 k 輩祖先還不是另一個點的祖先，那就把這個點往上提到他的這個祖先然後繼續做，直到這個點是另一個點的祖先，於是我們就可以找到了。

4.4 例題

1. TIOJ 1163：有 V（≤ 30000）個地點跟 E（≤ 50000）條雙向道路，每條道路都各必須等到某個時間之後才能通行。回答 Q（≤ 50000）筆詢問兩個點的最早可通行時間。

5 Directed Acyclic Graph 有向無環圖

如果這是一張圖，它是有向的，而且沒有環，那它就是有向無環圖。而一個好的 DP 順序也可以視為一個 DAG。

5.1 Topological ordering 拓樸排序

這是一個 DAG 若且唯若它有拓樸排序。所謂拓樸排序就是對於點的排序 $\{v_1, v_2, v_3, ..., v_n\}$，使得 $\forall(v_i, v_j) \wedge i \leq j$，沒有任何路徑從 v_j 到達 v_i。
找到拓樸排序大概常常會用這兩種方法 (吧)，複雜度都是穩穩的 $O(n)$。

1. Greedy：每次拔掉入度為零的點，然後把跟這個點相鄰的所有邊拔掉，然後更新接著的點的入度。如果點還沒拔完但是找不到入度為零的點，代表這張圖上其實有環。
2. DFS：隨便亂走。拓樸排序就是 DFS 的時候離開順序的相反。在 DFS 的時候，如果發現戳到了某個已經走過但是還沒離開的點，那就代表有環。

5.2 例題

1. Codeforces 510C、UVa 200：有 n（≤ 100）個字串，每個包含 ≤ 100 個字母。求定義一種英文字母大小關係使輸入是由小到大排序的。

5.3 單點源最短路徑

如果要重複取得以某個點為起點，以各個點為終點的路徑最短長度，那可以考慮這個解法。
不難發現，由每個點到起點的最短路徑的邊的聯集會是一棵原圖上的生成樹。所以這個問題也可以等價成求出這棵生成樹。如果題目不只要求距離的話，在做的時候只要順便紀錄轉移的來源就可以了。
另外，如果圖上存在負環的話，那求出最短路徑就是 NP-Complete problem 了。

5.4 Relaxation 鬆弛

在一個既存的路徑上，對於兩個點 s,t，由起點到這兩個點的距離分別為 L_s,L_t，如果可以找到一條邊使得 $L_s + W_{st} \leq L_t$ 的話（其中 W_{st} 代表 s 到 t 中這條邊的長度），那我們就可以用這條邊來「鬆弛」這整條路。所以如果要求出樹的話，只要在鬆弛的時候維護好每個節點是從誰走過來的就可以了。

5.5 Dijkstra's Algorithm

那個字我也不會唸。
如果現在的圖的邊權都是非負的，從上面那個生成樹的想法可以想到：某個不在目前的部份生成樹上且離起點最短的點一定會在最後的生成樹上。如果發現了這件事情的話，我們就可以用鬆弛來維護目前那些不在樹上的點到起點的距離，再套用 Prim's 的實作概念就可以完成這棵樹了。複雜度 $O(E \lg V)$。

5.6 Bellman-Ford Algorithm

上面提到的演算法不能處理負權邊，但是這個可以。主要想法是不斷的利用鬆弛。
把每條邊都拿來鬆弛一次整張圖的話，複雜度是 $O(E)$。但是因為樹的最大高度只有 V，所以我們最多只需要做 $V-1$ 次鬆弛就夠了，總複雜度 $O(VE)$。如果在第 V 次的鬆弛發現還能繼續鬆弛的話，那就代表這個圖上有負環。另外如果加上「有被鬆弛過的點才能拿來鬆弛其他點」的優化的話，複雜度有機會到達 $O(V+E)$。

6 多點源最短路徑

當題目要求的不只是一個點到各個點而是各點到各點的距離的時候，可以選擇做 V 次的單點源最短路徑，沒有負邊的情況 $O(VE \lg V)$，有負邊時 $O(V^2E)$，看起來在有負邊的時候有點糟。

6.1 Floyd-Warshall Algorithm

使用 DP 的話，我們可以紀錄 $dp_{k,i,j}$ 為把前 k 個點選擇性的當作中繼點之後由 i 到 j 的最短距離。如此一來就有個簡單的轉移：$dp_{k,i,j} = min\{dp_{k-1,i,j}, dp_{k-1,i,k} + dp_{k-1,k,j}\}$。可以發現因為每次計算的時候只會取到第 $k-1$ 次的 dp，所以可以滾動。如果需要還原的話，只要在取用中繼點的時候更新一個轉移來源的陣列就可以了。對於負環，只要找有沒有 $dp_{k,i,i} \leq 0$ 就好了。時間複雜度 $O(n^3)$，空間可以 $O(n^2)$。

6.2 例題們

1. TIOJ 1034：給 $N \times N$（≤ 20）的地圖，每個地圖上的點都有值。Q（$\leq N^4$）筆詢問兩個點的路徑中，可以把一個點的值改成 0 的狀況下，最小的總和是多少。
2. TIOJ 1641：有 N（$\leq 10^4$）個點，M（$\leq 2 \times 10^5$）條邊，第 i 條邊有值 C_i。如果經過了這條邊，那你身上的貨物量就要增加 C_i 倍。從起點出發時為 1 單位貨物，求到終點時貨物最輕能是多少。
3. TIOJ 1096：給 N（≤ 100）個點，求最小的環長度。
4. TIOJ 1028：給 N（≤ 13）個點跟一些想要到的點，求最短長度使得每個想去的點都被經過至少一次，如果有多組就輸出最小字典序的那組。