图论

庄博伟

ZUCC ACM Group

January 22nd 2022

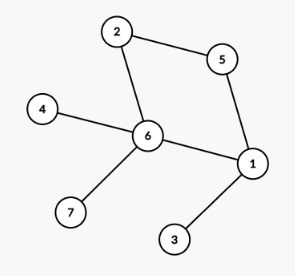
目录

图的定义

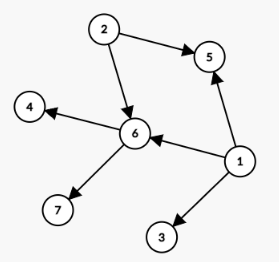
- 图由点 (Vertex) 和边 (Edge) 组成，边连接两个点，可以沿着边从一个点到另一个点。为了方便，我们接下来用 e(u, v) 来表示一条连接 u 和 v 的边

图的定义

- 图由点 (Vertex) 和边 (Edge) 组成，边连接两个点，可以沿着边从一个点到另一个点。为了方便，我们接下来用 e(u, v) 来表示一条连接 u 和 v 的边
- 边分为无向边（双向边）和有向边（单向边）两种，有向边 (u, v) 只能从 u 到 v，不能从 v 到 u
- 根据边的种类，图可以分为无向图和有向图两种



无向图



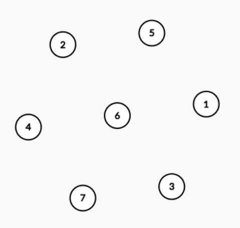
有向图

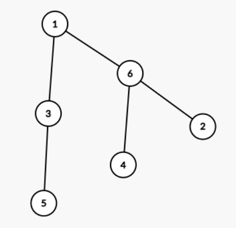
相邻

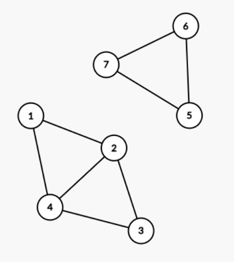
在无向图中，对于点 u 和点 v，若存在边 (u, v)，称 u 和 v 相邻 (Adjacent)

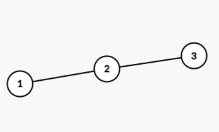
各种各样的图

- 图是一个非常广义的定义，所有由点和边组成的都可以称为图
- 一个点是图，树也是图，链也是图，没有点只有边也是图









路径、回路、环

- 路径 (Path)：多条边的组合，如路径 x -> y -> z，包含两条边 (x, y) 和 (y, z)。路径的长度就是组成路径的边的数量

路径、回路、环

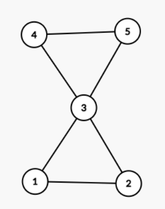
- 路径 (Path)：多条边的组合，如路径 x -> y -> z，包含两条边 (x, y) 和 (y, z)。路径的长度就是组成路径的边的数量
- 简单路径 (Simple path)：特殊的路径，要求不包含重复的边，连接的点也两两不同

路径、回路、环

- 路径 (Path)：多条边的组合，如路径 x -> y -> z，包含两条边 (x, y) 和 (y, z)。路径的长度就是组成路径的边的数量
- 简单路径 (Simple path)：特殊的路径，要求不包含重复的边，连接的点也两两不同
- 回路（Circuit）：也叫回路，开头和结尾同一个点的路径，要求不能包含重复的边

路径、回路、环

- 路径 (Path)：多条边的组合，如路径 x -> y -> z，包含两条边 (x, y) 和 (y, z)。路径的长度就是组成路径的边的数量
- 简单路径 (Simple path)：特殊的路径，要求不包含重复的边，连接的点也两两不同
- 回路（Circuit)：也叫回路，开头和结尾同一个点的路径，要求不能包含重复的边
- 环（Cycle / Simple circuit)：也叫简单回路，特殊的回路，要求起点和终点是唯一一对重复出现的点对

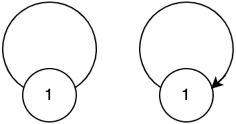


一个是回路但不是环的例子

$1 \to 2 \to 3 \to 4 \to 5 \to 3 \to 1$

简单图

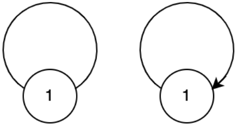
- 自环 (Self-loop)：一条边 u=v，称为自环，即自行成环



无向图自环　　有向图自环

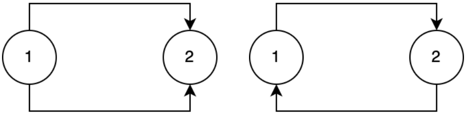
简单图

- 自环 (Self-loop)：一条边 u=v，称为自环，即自行成环



无向图自环　　有向图自环

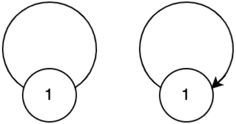
- 重边 (Multiple edge)：两条边连接的 u 和 v 相同，称为重边。需要注意的是有向图中，连接同样的两个点，但方向不同的话不视为重边



重边　　非重边

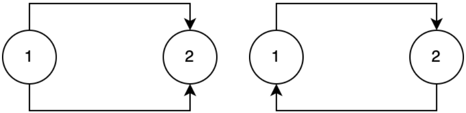
简单图

- 自环 (Self-loop)：一条边 u=v，称为自环，即自行成环



无向图自环　有向图自环

- 重边 (Multiple edge)：两条边连接的 u 和 v 相同，称为重边。需要注意的是有向图中，连接同样的两个点，但方向不同的话不视为重边



重边　非重边

- 没有自环和重边的图就是简单图 (Simple graph)

入度与出度

入度和出度是仅存于有向图中的概念，无向图没有这个东西

- 入度：一个点的入度等于指向该点的边的数量，即以该点作为 v 的边的数量
- 出度：一个点的出度等于该点指出去的边的数量，即以该点作为 u 的边的数量

无向图的连通

- 对于图上两个点 u 和 v，若存在以 u 为起点，v 为终点的路径，则称 u 和 v 是连通的 (Connected)

无向图的连通

- 对于图上两个点 u 和 v，若存在以 u 为起点，v 为终点的路径，则称 u 和 v 是连通的 (Connected)
- 对于一张无向图，若图中任意两点均连通，我们称该图是连通图 (Connected graph)

无向图的连通

- 对于图上两个点 u 和 v，若存在以 u 为起点，v 为终点的路径，则称 u 和 v 是连通的 (Connected)
- 对于一张无向图，若图中任意两点均连通，我们称该图是连通图 (Connected graph)
- 对于一个点集，若点集中任意两点均连通，并且点集内的点和点集外的点均不连通，则称该点集构成的子图为一个连通块 (Connected component，也叫连通分量)

连通块与并查集

- 连通块的概念、形式，其实和我们之前教的并查集很像，一个连通块相当于一个集合

连通块与并查集

- 连通块的概念、形式，其实和我们之前教的并查集很像，一个连通块相当于一个集合
- 考虑一张图初始时有 N 个点和 0 条边，就相当于并查集的初始状态。现在不断往图里加边 (u, v)，连接 u 和 v 所在的连通块，相当于并查集的合并操作

连通块与并查集

- 连通块的概念、形式，其实和我们之前教的并查集很像，一个连通块相当于一个集合
- 考虑一张图初始时有 N 个点和 0 条边，就相当于并查集的初始状态。现在不断往图里加边 (u, v)，连接 u 和 v 所在的连通块，相当于并查集的合并操作
- 1 个 N 个点的连通块最少需要 N-1 条边来连接，1 个 N 个点的集合最少需要 N-1 次合并

连通块与并查集

- 连通块的概念、形式，其实和我们之前教的并查集很像，一个连通块相当于一个集合
- 考虑一张图初始时有 N 个点和 0 条边，就相当于并查集的初始状态。现在不断往图里加边 (u, v)，连接 u 和 v 所在的连通块，相当于并查集的合并操作
- 1 个 N 个点的连通块最少需要 N-1 条边来连接，1 个 N 个点的集合最少需要 N-1 次合并

有向图的连通

- 有向图的连通和无向图类似，u 和 v 的连通需要一条 u 起点 v 终点的路径

有向图的连通

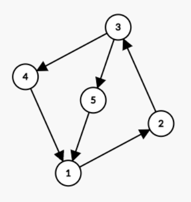
- 有向图的连通和无向图类似，u 和 v 的连通需要一条 u 起点 v 终点的路径
- 在无向图中，连通一定是双向的，u 和 v 连通意味着 v 和 u 也一定连通。但这在有向图中并不一定成立

有向图的连通

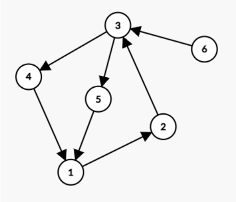
- 有向图的连通和无向图类似，u 和 v 的连通需要一条 u 起点 v 终点的路径
- 在无向图中，连通一定是双向的，u 和 v 连通意味着 v 和 u 也一定连通。但这在有向图中并不一定成立
- 对于一张有向图，若图中任意两点均连通，称为强连通 (Strongly connected)

有向图的连通

- 有向图的连通和无向图类似，u 和 v 的连通需要一条 u 起点 v 终点的路径
- 在无向图中，连通一定是双向的，u 和 v 连通意味着 v 和 u 也一定连通。但这在有向图中并不一定成立
- 对于一张有向图，若图中任意两点均连通，称为强连通 (Strongly connected)
- 若将所有的有向边换成无向边后，形成的无向图连通，则称该有向图为弱连通 (Weakly connected)



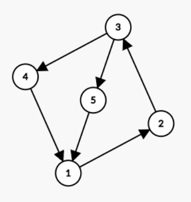
强连通图



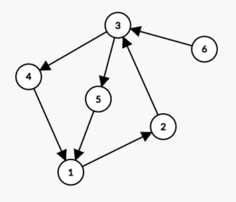
弱连通图

有向图的连通

- 有向图的连通和无向图类似，u 和 v 的连通需要一条 u 起点 v 终点的路径
- 在无向图中，连通一定是双向的，u 和 v 连通意味着 v 和 u 也一定连通。但这在有向图中并不一定成立
- 对于一张有向图，若图中任意两点均连通，称为强连通 (Strongly connected)
- 若将所有的有向边换成无向边后，形成的无向图连通，则称该有向图为弱连通 (Weakly connected)



强连通图

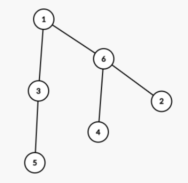


弱连通图

- 和连通分量类似的，有强连通分量和弱连通分量

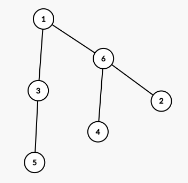
树和图的关联

树也是一种特殊的图，那么树特殊在哪呢



树和图的关联

树也是一种特殊的图，那么树特殊在哪呢



- 用上面的定义来讲，树 = 无向 + 连通 + 简单图
- 反过来讲，N 个点 N-1 条边的无向连通图一定是树

无权图和有权图

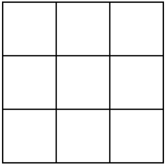
- 方便起见，今天不讲点权，只讲边权。一些图的边会带有权值，权值可能代表这条边的长度，或者通过这条边需要消耗或者可以获得的物品数量

无权图和有权图

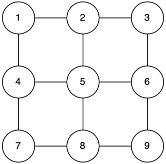
- 方便起见，今天不讲点权，只讲边权。一些图的边会带有权值，权值可能代表这条边的长度，或者通过这条边需要消耗或者可以获得的物品数量
- 没有边权的图称为无权图，无权图也可以看作是所有边权均为 1 的有权图

无权图和有权图

- 方便起见，今天不讲点权，只讲边权。一些图的边会带有权值，权值可能代表这条边的长度，或者通过这条边需要消耗或者可以获得的物品数量
- 没有边权的图称为无权图，无权图也可以看作是所有边权均为 1 的有权图
- 其实无权图我们已经接触过很多了，之前讲 BFS 时经常碰到的迷宫就是无权图的一种



迷宫



迷宫转化成无权图

图的存储

一般存图都用邻接表，对每个点存储所有相邻的点，用 vector 实现

图的存储

一般存图都用邻接表，对每个点存储所有相邻的点，用 vector 实现

- 无向无权图：

```
vector<int> G[N];
void add_edge(int u, int v) {
  G[u].push_back(v);
  G[v].push_back(u);
}
```

图的存储

一般存图都用邻接表，对每个点存储所有相邻的点，用 vector 实现

- 无向无权图：

```
vector<int> G[N];
void add_edge(int u, int v) {
  G[u].push_back(v);
  G[v].push_back(u);
}
```

- 有向有权图：

```
struct Edge {
  int to, w;
};
vector<Edge> G[N];
void add_edge(int u, int v, int w) {
  G[u].push_back({v, w});
}
```

图的存储

一般存图都用邻接表，对每个点存储所有相邻的点，用 vector 实现

- 无向无权图：

```
vector<int> G[N];
void add_edge(int u, int v) {
  G[u].push_back(v);
  G[v].push_back(u);
}
```

- 有向有权图：

```
struct Edge {
   int to, w;
};
vector<Edge> G[N];
void add_edge(int u, int v, int w) {
  G[u].push_back({v, w});
}
```

有向图不用加 $v \to u$ 这条边，有权图需要额外记边权

最短路问题

最短路问题是图论中最典型的一类问题，从 u 到 v 的最短路指的是长度最短的从 u 到 v 的路径。

- 在无权图上，路径的长度 $=$ 组成路径的边的数量
- 在有权图上，路径的长度 $=$ 组成路径的边的权值和

最短路问题

最短路问题是图论中最典型的一类问题，从 u 到 v 的最短路指的是长度最短的从 u 到 v 的路径。

- 在无权图上，路径的长度 = 组成路径的边的数量
- 在有权图上，路径的长度 = 组成路径的边的权值和

有时候题目只要求你求出从 1 个固定的点 x 到其余所有点的最短路，这个叫单源最短路。相对的，有多个起点的叫多源最短路

目录

BFS 的用途

Breadth First Search，广度优先搜索，也叫宽度优先搜索

- 之前讲 BFS 的时候都是用的迷宫格子图。实际上只要是无权图，都可以使用 BFS 来求出单源最短路
- 有向图和无向图写 BFS 代码是完全一样的，毕竟把无向图每条边拆成两条有向边，就变成有向图了

BFS 的算法过程

```
vector<int> G[N];
int d[N];
void bfs(int src, int n) {
  for (int i = 1; i <= n; i++) d[i] = -1;
  d[src] = 0;
  queue<int> q;
  q.push(src);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int v : G[u]) {
      if (d[v] == -1) {
        d[v] = d[u] + 1;
        q.push(v);
      }
    }
  }
}
```

- d[i] 表示从 src 到 i 的最短路，src 为起点，n 为图中点的数量

BFS 的算法过程

```cpp
vector<int> G[N];
int d[N];
void bfs(int src, int n) {
  for (int i = 1; i <= n; i++) d[i] = -1;
  d[src] = 0;
  queue<int> q;
  q.push(src);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int v : G[u]) {
      if (d[v] == -1) {
        d[v] = d[u] + 1;
        q.push(v);
      }
    }
  }
}
```

- d[i] 表示从 src 到 i 的最短路，src 为起点，n 为图中点的数量
- 将起点加入队列，每次取出队列头部的点 u，若与 u 相邻的点中存在没有访问过的点 v，则更新 d_v 并且加入队列中，重复操作直到队列为空

BFS 的算法过程

```
vector<int> G[N];
int d[N];
void bfs(int src, int n) {
  for (int i = 1; i <= n; i++) d[i] = -1;
  d[src] = 0;
  queue<int> q;
  q.push(src);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int v : G[u]) {
      if (d[v] == -1) {
        d[v] = d[u] + 1;
        q.push(v);
      }
    }
  }
}
```

- d[i] 表示从 src 到 i 的最短路，src 为起点，n 为图中点的数量
- 将起点加入队列，每次取出队列头部的点 u，若与 u 相邻的点中存在没有访问过的点 v，则更新 d_v 并且加入队列中，重复操作直到队列为空

可以发现按这段代码跑，每个点只会进行一次入队和出队

BFS 的算法原理

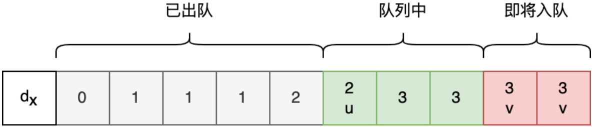
BFS 的队列与其说存储的是点，不如说存储的是从起点到该点的一条路径。每次往队列中加入新点，其实是用已有路径 $d_u +$ 一条边 (u, v) 得到一条新路径 d_v

BFS 的算法原理

BFS 的队列与其说存储的是点，不如说存储的是从起点到该点的一条路径。每次往队列中加入新点，其实是用已有路径 d_u+ 一条边 (u, v) 得到一条新路径 d_v

这些路径最重要的性质有两点

- 入队越早的路径，长度一定越小
- 每条入队的路径 d_x 一定是 src 到 x 的最短路



性质 1 比较好理解，理解性质 2 就理解了 BFS，下面慢慢讲

松弛

在讲性质 2 之前，需要引入一个松弛的概念：

```
if (d[u] + e.w < d[v]) {
  d[v] = d[u] + e.w;
}
```

用已得到的较短路径 d_u 和边 (u,v) 的权值去优化 d_v，称为松弛，这个过程和 DP 非常的像，不如说本质就是 DP

松弛

在讲性质 2 之前，需要引入一个松弛的概念：

```
if (d[u] + e.w < d[v]) {
  d[v] = d[u] + e.w;
}
```

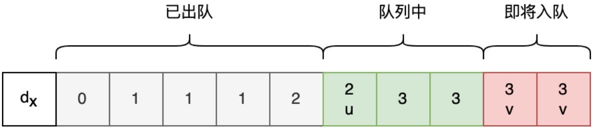
用已得到的较短路径 d_u 和边 (u,v) 的权值去优化 d_v，称为松弛，这个过程和 DP 非常的像，不如说本质就是 DP

BFS 代码中的这段就是松弛

```
d[v] = d[u] + 1;
```

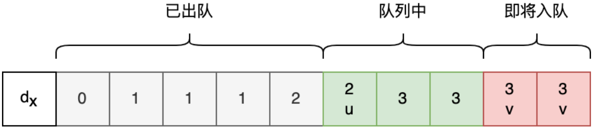
BFS 的算法原理

再接着讲第二个性质：每个入队的路径 d_x 一定是 src 到 x 的最短路



BFS 的算法原理

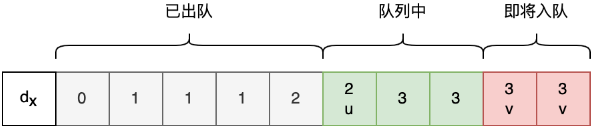
再接着讲第二个性质：每个入队的路径 d_x 一定是 src 到 x 的最短路



- 这个性质其实和第一个性质有关：入队越早的路径，路径长度越小

BFS 的算法原理

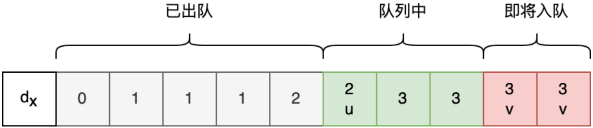
再接着讲第二个性质：每个入队的路径 d_x 一定是 src 到 x 的最短路



- 这个性质其实和第一个性质有关：入队越早的路径，路径长度越小
- 再考虑到无权图所有边权相等，被早入队的路径松弛一定优于被晚入队的路径松弛

BFS 的算法原理

再接着讲第二个性质：每个入队的路径 d_x 一定是 src 到 x 的最短路



- 这个性质其实和第一个性质有关：入队越早的路径，路径长度越小
- 再考虑到无权图所有边权相等，被早入队的路径松弛一定优于被晚入队的路径松弛
- 因此每条路径第一次被松弛到（即入队）的时候，就已经是最优的了

BFS 的复杂度

- 时间复杂度：每个点只会入队一次，因此时间复杂度是 O(N)，N 为点数
- 空间复杂度：队列最多同时有 N 个元素，因此空间复杂度 O(N)

BFS 与有权图

- BFS 仅适用于求无权图的最短路，并不适用于求有权图的最短路

BFS 与有权图

- BFS 仅适用于求无权图的最短路，并不适用于求有权图的最短路
- 原因在于有权图上边权不同，用较短的路径松弛不一定更优

BFS 与有权图

- BFS 仅适用于求无权图的最短路，并不适用于求有权图的最短路
- 原因在于有权图上边权不同，用较短的路径松弛不一定更优
- 早入队的路径不一定都比晚入队的更短

BFS 与有权图

- BFS 仅适用于求无权图的最短路，并不适用于求有权图的最短路
- 原因在于有权图上边权不同，用较短的路径松弛不一定更优
- 早入队的路径不一定都比晚入队的更短
- 路径入队时也不一定直接就是最短

目录

简介

Dijkstra 是一种用于有权图求单源最短路的算法

使用 Dijkstra 的有权图有一个要求，所有的边权均不能为负值

Dijkstra 的基本思路

- BFS 在有权图上无法保证每条路径入队时都是最短路
- 那我们退而求其次，尝试保证另一个性质：保证每条路径出队时都是最短路

Dijkstra 的基本思路

性质

每条路径出队时都是最短路

这个新性质可以非常好保证，我们只需要保证每次出队的（即每次挑出来松弛别人的）那条路径是队列中最短的路径就可以了

Dijkstra 的基本思路

性质

每条路径出队时都是最短路

这个新性质可以非常好保证，我们只需要保证每次出队的（即每次挑出来松弛别人的）那条路径是队列中最短的路径就可以了

用归纳法证明一下：

- 先假设这个结论正确，即已出队的均为最短路

Dijkstra 的基本思路

性质

每条路径出队时都是最短路

这个新性质可以非常好保证，我们只需要保证每次出队的（即每次挑出来松弛别人的）那条路径是队列中最短的路径就可以了

用归纳法证明一下：

- 先假设这个结论正确，即已出队的均为最短路
- 那么贪心一下，还存在于队列的路径中，最短的那一条一定无法再被松弛了。因为出队的路径都已经松弛过了，而还在队列中的路径都比这一条长，边权值又都是非负数

Dijkstra 的基本思路

性质

每条路径出队时都是最短路

这个新性质可以非常好保证，我们只需要保证每次出队的（即每次挑出来松弛别人的）那条路径是队列中最短的路径就可以了

用归纳法证明一下：

- 先假设这个结论正确，即已出队的均为最短路
- 那么贪心一下，还存在于队列的路径中，最短的那一条一定无法再被松弛了。因为出队的路径都已经松弛过了，而还在队列中的路径都比这一条长，边权值又都是非负数
- 无法再被松弛 = 最短路，第一次出队的 d_{src} 显然是最短路，结论成立

Dijkstra 的基本思路

- Dijkstra 和 BFS 的算法过程其实是一样的：每次挑选没选过的、最短的路径，通过该路径去松弛其他的路径
- 区别只是在于应用于无权图上时，每个 d_x 第一次松弛到时就是最短路，而应用于有权图上时，需要多次松弛才会变成最短路

代码实现实例

```cpp
struct Edge {
  int to, w;
}
vector<Edge> G[N];
struct Node {
  int u, d;
  bool operator < (const Node &tmp) const {
    return d > tmp.d;
  }
}
int d[N];
bool vis[N];
void dijkstra(int src, int n) {
  for (int i = 1; i <= n; i++) {
    d[i] = 1e9; vis[i] = false;
  }
  d[src] = 0;
  priority_queue<Node> q;
  q.push({src, d[src]});
  while (!q.empty()) {
    int u = q.top().u; q.pop();
    if (vis[u]) continue;
    vis[u] = true;
    for (Edge e : G[u]) {
      int v = e.to;
      if (!vis[v] && d[u] + e.w < d[v]) {
        d[v] = d[u] + e.w;
        q.push({v, d[v]});
      }
    }
  }
}
```

目录

拓扑排序

拓扑排序是专门用于有向图上的算法，这个算法和 DAG(Directed Acyclic Graph，有向无环图) 经常同时出现，无环，也就是说不能有强连通分量

算法过程

```cpp
int in[N]; //入度
void top(int n) {
  for (int i = 1; i <= n; i++) {
    for (int j : G[i]) in[j]++;
  }
  queue<int> q;
  for (int i = 1; i <= n; i++) {
    if (in[i] == 0) q.push(i);
  }
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int v : G[u]) {
      in[v]--;
      if (in[v] == 0) q.push(v);
    }
  }
}
```

维护每个点的入度，每次挑选一个入度为 0 的点，删去该点以及从该点指出去的边，直到点被删完

算法过程

```cpp
int in[N]; //入度
void top(int n) {
  for (int i = 1; i <= n; i++) {
    for (int j : G[i]) in[j]++;
  }
  queue<int> q;
  for (int i = 1; i <= n; i++) {
    if (in[i] == 0) q.push(i);
  }
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int v : G[u]) {
      in[v]--;
      if (in[v] == 0) q.push(v);
    }
  }
}
```

维护每个点的入度，每次挑选一个入度为 0 的点，删去该点以及从该点指出去的边，直到点被删完

拓扑序

一个点的拓扑序指的是该点在拓扑排序中是第几个出队的

显然一张图的拓扑序并不唯一

拓扑排序的应用

- 判断一张有向图是否 DAG（有没有环）：如果点还没删完，但找不到入度为 0 的点了，说明这张有向图有环
- DAG 上 DP，即按照拓扑的形式来转移状态

几个基于拓扑的小定理

- 对于一张 DAG，某点 x 能到达所有点的充要条件是：x 是唯一入度为 0 的点
- 对于一张 DAG，某点 x 能被所有点到达的充要条件是：x 是唯一出度为 0 的点
- 对于一张 DAG 上的任意拓扑序，任何点不可能到达拓扑序小于它的点

这里的结论绝对不能死背，尝试去理解，去自己证明，才能让你的图论入门

END