图论（强化）

2021-10-17：强化图论相关数据结构和算法

图的应用：最短路径

Leetcode经典题目:743. 网络延迟时间

1：（单源点最短路径问题）Dijkstra算法

无向图的Dijkstra算法

题目描述

给你n个点，m条无向边，每条边都有长度d和花费p，给你起点s终点t，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入描述:

```
输入n,m，点的编号是1~n,然后是m行，每行4个数 a,b,d,p，表示a和b之间有一条边，且其长度为d，花费为p。
最后一行是两个数 s,t;起点s，终点t。n和m为0时输入结束。
(1<n<=1000, 0<m<100000, s != t)
```

输出描述:

```
输出 一行有两个数， 最短距离及其花费。
```

输入

```
3 2
1 2 5 6
2 3 4 5
1 3
0 0
```

输出

```
9 11
```

```
#include<iostream>
#include<vector>
using namespace std;
struct E{
    int next;
    int len;//路径长度
    int cost;//建造路径的花费
};
vector<E>edge[1001];//用于构建邻接表
int Dis[1001];//存储到各点的最短路径
int cost[1001];//存储到各点的最小花费
bool mark[1001];//标记点是否被归档（归档表示该点的最短路径和花费已经被确定）
int main(){
    int n,m,S,T;
    while(scanf("%d %d",&n,&m)!=EOF&&n!=0&&m!=0){
        for(int i=0;i<=n;i++){
            edge[i].clear();
        }
        for(int i=1;i<=m;i++){
            int a,b,l,c;
            cin>>a>>b>>l>>c;
            E tmp;
            tmp.next=b;
            tmp.len=l;
            tmp.cost=c;
            //注意此处push_back是个函数
            edge[a].push_back(tmp);
            tmp.next=a;
            edge[b].push_back(tmp);
        }
        cin>>S>>T;
        for(int i=1;i<=n;i++){
            Dis[i]=-1;
            mark[i]=false;
        }
        Dis[S]=0;
        mark[S]=true;
        int newP=S;
        for(int i=1;i<=n;i++){
            for(int j=0;j<edge[newP].size();j++){
                int next=edge[newP][j].next;
                int len=edge[newP][j].len;
                int Cost=edge[newP][j].cost;
                if(mark[next]==true)continue;

 if(Dis[next]==-1||Dis[next]>Dis[newP]+len||Dis[next]==Dis[newP]+len&&co
st[next]>cost[newP]+Cost){
                    Dis[next]=Dis[newP]+len;
                    cost[next]=cost[newP]+Cost;
```

```
            }
        }
        int min=123123123;
        //求已归档点中路径长度最小的点--此处有点耗时
        for(int i=1;i<=n;i++){
            if(mark[i]==true)continue;
            if(Dis[i]==-1)continue;
            if(min>Dis[i]){
                min=Dis[i];
                newP=i;
            }
        }
        mark[newP]=true;
    }
    cout<<Dis[T]<<" "<<cost[T]<<endl;
  }
}
```

有向图的Dijkstra算法

题目描述：743. 网络延迟时间

> 有 `n` 个网络节点，标记为 `1` 到 `n`。
>
> 给你一个列表 `times`，表示信号经过 **有向** 边的传递时间。 `times[i] = (ui, vi, wi)`，其中 `ui` 是源节点，`vi` 是目标节点， `wi` 是一个信号从源节点传递到目标节点的时间。
>
> 现在，从某个节点 `K` 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到信号，返回 `-1` 。

优先队列优化

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>> &times, int n, int k) {
        const int inf = INT_MAX / 2;
        vector<vector<pair<int, int>>> g(n);
        //创建邻接表
        for (auto &t : times) {
            int x = t[0] - 1, y = t[1] - 1;
            g[x].emplace_back(y, t[2]);
        }

        vector<int> dist(n, inf);
```

```
        dist[k - 1] = 0;
        //创建小顶堆-优先队列
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<>> q;
        //优先队列的入队建议使用q.emplace(),花销较小
        q.emplace(0, k - 1);
        while (!q.empty()) {
            auto p = q.top();
            q.pop();
            int time = p.first, x = p.second;
            //优先队列中所有的点不一定都会取用，只有当其表示的点严格小于已经确定的最
短路径时才会采用
            //通过优先队列从而不用用一个数组来确定某个点是否被访问
            if (dist[x] < time) {
                continue;
            }
            for (auto &e : g[x]) {
                int y = e.first, d = dist[x] + e.second;
                if (d < dist[y]) {
                    dist[y] = d;
                    q.emplace(d, y);
                }
            }
        }
        //max_element返回数组中的最大值的迭代器
        int ans = *max_element(dist.begin(), dist.end());
        return ans == inf ? -1 : ans;
    }
};

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/network-delay-time/solution/wang-
luo-yan-chi-shi-jian-by-leetcode-so-6phc/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

（内化版）优先队列优化

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        //int n=times.size();
        vector<vector<pair<int,int>>>edges(n+1);
        //创建邻接表
        for(auto&d:times){
            edges[d[0]].push_back({d[1],d[2]});
        }
        //创建dist数组
        vector<int>dist(n+1,INT_MAX/2);
```

```
        //创建优先队列

 priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int
>> >que;

        //初始化
        dist[0]=dist[k]=0;//0是多余的
        //que.emplace({0,k});
        //emplace传入参数进行构造
        que.emplace(0,k);

        while(!que.empty()){
            pair<int,int>p=que.top();
            que.pop();
            //验证点是否是最新信息
            int nd=p.second,d=p.first;
            if(dist[nd]<d)continue;//?可不可以等于--不可以吧
            for(auto&[f,s]:edges[nd]){
                if(dist[f]>dist[nd]+s){
                    dist[f]=dist[nd]+s;
                    que.push({dist[f],f});
                }
            }
        }

        int ans=*max_element(dist.begin(),dist.end());
        return ans==INT_MAX/2?-1:ans;


    }
};
```

2：图的应用：最短路径-单源算法——Bellman_ford算法(解决负权回路的算法)

算法思想：

假设p为源点到节点的最短路径，显然这条路径上最多包含n - 1条边，那么我们可以通过 n - 1 次循环，每次循环松弛所有边，根据路径松弛定理，最终我们可以得到正确的答案，算法的正确性不作证明，可自行查阅。由于进行了全面的松弛，最后得到的结果根据三角形定则，一定有 dis[v] < dis[u] + w(假设有边u->v = w)，否则即存在负边回路。

原理 [编辑]

循环 [编辑]

每次循环操作实际上是对相邻节点的访问，第n次循环操作保证了所有深度为n的路径最短。由于图的最短路径最长不会经过超过$|V|-1$条边，所以可知贝尔曼-福特算法所得为最短路径。

负边权操作 [编辑]

与迪科斯彻算法不同的是，迪科斯彻算法的基本操作“拓展”是在深度上寻路，而“松弛”操作则是在广度上寻路，这就确定了贝尔曼-福特算法可以对负边进行操作而不会影响结果。

负权环判定 [编辑]

因为负权环可以无限制的降低总花费，所以如果发现第n次操作仍可降低花销，就一定存在负权环。

1. 该算法的特点是可以允许边的权值为负值;
2. 同时可以检验是否存在负权环

```
class Solution {
private:
    vector<int>dist;//存储最短路径的值
    bool Bellman_Flod(vector<vector<int>>& g,int n){
        //进行n-1次松弛操作
        int E=g.size();
        for(int i=0;i<n;i++){
            //通过每条边进行松弛
            for(int j=0;j<E;j++){
                if(dist[g[j][1]]>dist[g[j][0]]+g[j][2])
                dist[g[j][1]]=dist[g[j][0]]+g[j][2];
            }
        }
        //检验是否存在负权环--如果还能松弛表明存在负权环
        for(int j=0;j<E;j++){
            if(dist[g[j][1]]>dist[g[j][0]]+g[j][2])
            return false;
        }
        return true;
    }
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        dist.resize(n+1,INT_MAX/2);
        dist[0]=0;//dist【0】不存在
        dist[k]=0;//初始化
        if(!Bellman_Flod(times,n))return -1;//存在负权环
        int ans=*max_element(dist.begin(),dist.end());
        return ans==INT_MAX/2?-1:ans;
    }
};
```

3：单源算法——Spfa算法(强大而又精妙的万能单源算法)（改进的Bellman_ford算法）

最短路径快速算法（英语：Shortest Path Faster Algorithm (SPFA)），国际上一般认为是队列优化的Bellman-Ford 算法，一般仅在中国大陆被称为SPFA，是一个用于求解有向带权图单源最短路径的算法。这一算法被认为在随机的稀疏图上表现出色，并且适用于带有负边权的图。[1] 然而SPFA在最坏情况的时间复杂度与 Bellman-Ford 算法相同，因此在非负边权的图中使用堆优化的Dijkstra 算法有可能优于SPFA。[2] SPFA算法首先在1959年由Edward F. Moore作为广度优先搜索的扩展发表[3]。相同算法在1994年由段凡丁重新发现。[4]

给定一个有向带权图$G=(V,E)$和一个源点s，SPFA算法可以计算从s到图中每个节点v的最短路径。其基本思路与 Bellman-Ford 算法相同，即每个节点都被用作用于松弛其相邻节点的备选节点。但相较于 Bellman-Ford 算法，SPFA算法的改进之处在于它并不盲目地尝试所有节点，而是维护一个备选的节点队列，并且仅有节点被松弛后才会放入队列中。整个流程不断重复直至没有节点可以被松弛。

下面是这个算法的伪代码。[5]这里的Q是一个备选节点的先进先出队列，$w(u,v)$ 是边(u,v)的权值。

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<int>dist(n+1,INT_MAX/2);//最短路径
        vector<int>cut(n+1,0);//表示每个点入队的次数，当入队的次数大于等于n时表示存在负权环
        vector<vector<pair<int,int>>>edges(n+1);//构建邻接表
        for(auto&e:times){
            edges[e[0]].push_back({e[1],e[2]});
        }
        //设置一个队列存储可以松弛的点
        queue<int>que;
        vector<bool>onque(n+1,false);//标记是否在队列中
        //初始化
        dist[0]=dist[k]=0;
        que.push(k);
        onque[k]=true;
        //开始处理
        while(!que.empty()){
            int t=que.front();
            que.pop();
            onque[t]=false;
            for(auto&[v,w]:edges[t]){
                if(dist[v]>dist[t]+w){
                    dist[v]=dist[t]+w;
                    if(!onque[v]){
                        que.push(v);
                        onque[v]=true;
                        ++cut[v];
                        if(cut[v]>=n){
                            //存在负权环
                            return -1;
                        }
                    }
                }
            }
```

```
        }

        int ans=*max_element(dist.begin(),dist.end());
        return ans==INT_MAX/2?-1:ans;


    }
};
```

4:多源算法——Floyd算法(简洁而优雅的算法)

该算法通常用以解决所有节点对的最短路径，该算法利用了这样一个有趣的事实：如果从节点 i 到节点 j，如果存在一条更短的路径话，那么一定是从另一个节点 k 中转而来，即有 d[i][j] = min(d[i][j],d[i][k]+d[k][j])，而d[i][k]和d[k][j]可以用一样的思想去构建，可以看出这是一个动态规划的思想。在构建i,j中，我们通过枚举所有的k值来进行操作。但是，该算法无法判断负权回路。

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        //Floyd算法
        //邻接矩阵实现
        vector<vector<int>>edges(n+1,vector<int>(n+1,INT_MAX/2));
        //邻接矩阵初始化
        for(auto&g:times){
            edges[g[0]][g[1]]=g[2];
        }
        for(int i=1;i<=n;i++)edges[i][i]=0;

        //注意遍历的顺序
        //最外层是中间结点的遍历
        for(int m=1;m<=n;m++){
            for(int j=1;j<=n;j++){
                for(int i=1;i<=n;i++){
                    edges[i][j]=min(edges[i][j],edges[i][m]+edges[m]
[j]);
                }
            }
        }
        int res=0;
        for(int i=1;i<=n;i++){
            res=max(res,edges[k][i]);
```

```
        }
        return res==INT_MAX/2?-1:res;
    }
};
```

图的应用：拓扑排序

在计算机科学领域，有向图的**拓扑排序**或**拓扑测序**是对其顶点的一种线性排序，使得对于从顶点u到顶点v的每个有向边uv，u在排序中都在v之前。

拓扑序列的特点：

- 有向图、无环
- 各个结点之间存在先后关系或并行关系。就像某些课程有先修课程一样

LeetCode经典题目：802. 找到最终的安全状态

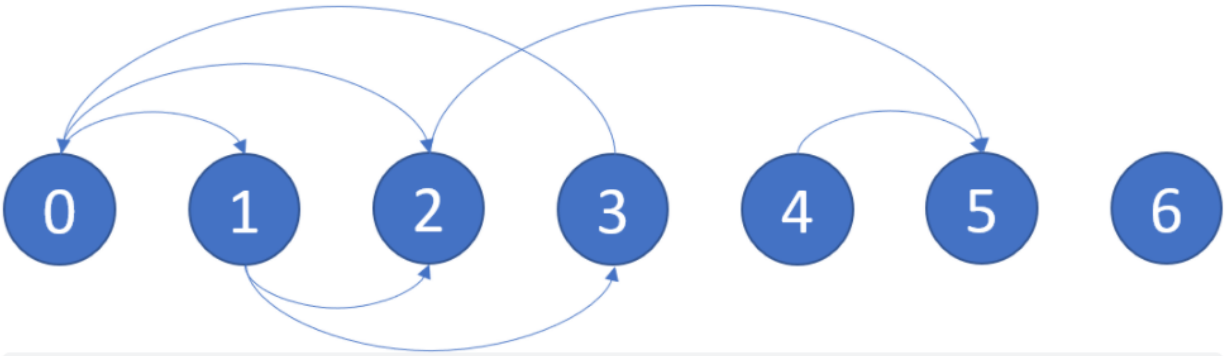
在有向图中，以某个节点为起始节点，从该点出发，每一步沿着图中的一条有向边行走。如果到达的节点是终点（即它没有连出的有向边），则停止。

对于一个起始节点，如果从该节点出发，**无论每一步选择沿哪条有向边行走**，最后必然在有限步内到达终点，则将该起始节点称作是 **安全** 的。

返回一个由图中所有安全的起始节点组成的数组作为答案。答案数组中的元素应当按 **升序** 排列。

该有向图有 `n` 个节点，按 `0` 到 `n - 1` 编号，其中 `n` 是 `graph` 的节点数。图以下述形式给出：`graph[i]` 是编号 `j` 节点的一个列表，满足 `(i, j)` 是图的一条有向边。

示例 1：




```
输入：graph = [[1,2],[2,3],[5],[0],[5],[],[]]
输出：[2,4,5,6]
解释：示意图如上。
```

方法1：深度优先拓扑排序（三色法）

1. 求拓扑排序如果用深度优先算法+**visited**数组（三色法）求出的是一个逆拓扑序列
2. 三色法：
   - 白色（**0**）：表示未标记，从未访问
   - 灰色（**1**）：表示第一次访问，并搜索其后继结点
   - 深色（**2**）：表示第二次访问，此时其后继结点都被访问，于是可以输出
3. 从任意一个未标记的结点开始进行深度优先搜索，直到所有的结点都被搜索

深度优先搜索+三色法的关键是：通过三色法来表示结点的状态或访问的次数；

本题经过理解转换就是求除去自环后的图中的其他结点！！！（拓扑排序是可以用来检验自环的）

- 白色（用 0 表示）：该节点尚未被访问；
- 灰色（用 1 表示）：该节点位于递归栈中，或者在某个环上；
- 黑色（用 2 表示）：该节点搜索完毕，是一个安全节点。

```
class Solution {
private:
    vector<int>visited;
    bool dfs(vector<vector<int>>& graph,int x){
        //存在闭环
        if(visited[x]==1){
            return false;
        }else if(visited[x]==2){
```

```
            return true;
        }
        if(visited[x]==0){
            visited[x]=1;
        }
        for(int i=0;i<graph[x].size();i++){
            if(!dfs(graph,graph[x][i])){
                return false;
            }
        }
        visited[x]=2;
        return true;

    }
public:
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
        //深度优先，超时了
        //修改一下——深度优先+三色法
        int n=graph.size();
        //0表示未访问，1表示在栈中或在环中，2表示结点安全
        visited.resize(n,0);

        for(int i=0;i<n;i++){
            if(visited[i]==1||visited[i]==2)continue;
            dfs(graph,i);
        }
        vector<int>res;
        for(int i=0;i<n;i++){
            if(visited[i]==2)res.push_back(i);
        }
        return res;
    }
};
```

方法2：广度优先拓扑排序（队列）

- 将入度未**0**的点入队
- 从入度为**0**的点开始广度优先搜索，并将搜索到的点的入度减一，如过入度减为**0**就将其入队，直到队列为空
- 如果仍有结点未被输出，则说明图中存在回路
- 这道题需要先把图反向得到返图再拓扑排序

```
class Solution {
public:
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
        //玩一手拓扑排序
        //将边反向
        int n=graph.size();
```

```
        vector<vector<int>>edges(n);
        for(int i=0;i<n;i++){
            int n1=graph[i].size();
            for(int j=0;j<n1;j++){
                edges[graph[i][j]].emplace_back(i);
            }
        }
        //统计各个顶点的入度
        vector<int>indegree(n,0);
        for(int i=0;i<n;i++){
            int n1=edges[i].size();
            for(int j=0;j<n1;j++){
                ++indegree[edges[i][j]];
            }
        }
        //将入度为0的点入队列
        queue<int>que;
        vector<bool>ans(n,false);
        for(int i=0;i<n;i++){
            if(!indegree[i]){
                que.push(i);
                ans[i]=true;
            }
        }
        //开始搞事情
        while(!que.empty()){
            int d=que.front();
            que.pop();
            for(int i=0;i<edges[d].size();i++){
                int temp=edges[d][i];
                --indegree[temp];
                if(!indegree[temp]){
                    que.push(temp);
                    ans[temp]=true;
                }
            }
        }
        //寻找答案
        vector<int>res;
        for(int i=0;i<n;i++){
            if(ans[i])res.push_back(i);
        }
        return res;
    }
};
```

LeetCode相关题目：

✓	207. 课程表	955	54.3%	中等	
✓	630. 课程表 III	76	36.8%	困难	
✓	210. 课程表 II	686	54.3%	中等	
✓	1462. 课程表 IV	125	42.8%	中等	

图的应用：关键路径

AOE网是一个带权的DAG（有向无环图），用顶点表示事件，有向边表示活动，边上的权表示完成该活动持续的时间！

- 只有在某个顶点所代表的事件发生后，从该顶点出发的各活动才能发生
- 只有在进入某顶点的各活动都结束后，该顶点所代表的事件才能开始

5909. 并行课程 III

给你一个整数 `n` ，表示有 `n` 节课，课程编号从 `1` 到 `n` 。同时给你一个二维整数数组 `relations` ，其中 `relations[j] = [prevCoursej, nextCoursej]` ，表示课程 `prevCoursej` 必须在课程 `nextCoursej` **之前** 完成（先修课的关系）。同时给你一个下标从 **0** 开始的整数数组 `time` ，其中 `time[i]` 表示完成第 `(i+1)` 门课程需要花费的 **月份** 数。

请你根据以下规则算出完成所有课程所需要的 **最少** 月份数：

- 如果一门课的所有先修课都已经完成，你可以在 **任意** 时间开始这门课程。
- 你可以 **同时** 上 **任意门课程** 。

请你返回完成所有课程所需要的 **最少** 月份数。

注意： 测试数据保证一定可以完成所有课程（也就是先修课的关系构成一个有向无环图）。

```
class Solution {
public:
    int minimumTime(int n, vector<vector<int>>& relations, vector<int>&
time) {
        //构建有向图
        //每个结点的开始时间为其前趋结点中的最晚完成时间
```

```cpp
        //每个结点的结束时间为：最晚完成时间+完成该课程所需的时间
        //int n=time.size();
        vector<vector<int>>edges(n+1);
        vector<int>indegree(n+1,0);//每个结点的入度
        for(auto&v:relations){
            edges[v[0]].push_back(v[1]);
            ++indegree[v[1]];
        }
        vector<int>endtime(n+1,0);//表示的没门课程的最早完成时间
        queue<int>que;
        //入度为0的结点入队--从1开始
        int res=0;
        for(int i=1;i<=n;i++){
            if(indegree[i]==0){
                que.push(i);
                endtime[i]=time[i-1];
            }
        }
        while(!que.empty()){
            int t=que.front();
            que.pop();
            for(auto u:edges[t]){
                endtime[u]=max(endtime[u],endtime[t]+time[u-1]);
                --indegree[u];
                if(indegree[u]==0)que.push(u);
            }
        }
        for(int i=1;i<=n;i++){
            res=max(res,endtime[i]);
        }
        return res;
    }
};
```