ACM-ICPC 培训资料汇编

（4）

图论分册

（版本号 **1.0.0**）

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

序

2012 年 5 月，哈尔滨理工大学承办了 ACM-ICPC 黑龙江省第七届大学生程序设计竞赛。做为本次竞赛的主要组织者，我还是很在意本校学生是否能在此次竞赛中取得较好成绩，毕竟这也是学校的脸面。因此，当 2011 年 10 月确定学校承办本届竞赛后，我就给齐达拉图同学很大压力，希望他能认真训练参赛学生，严格要求受训队员。当然，齐达拉图同学半年多的工作还是很有成效，不仅带着黄李龙、姜喜鹏、程宪庆、卢俊达等队员开发了我校的 OJ 主站和竞赛现场版 OJ，还集体带出了几个比较像样的新队员，使得今年省赛我校取得了很好的成绩（当然，也承蒙哈工大和哈工程关照，没有派出全部大牛来参赛）。

在 2011 年 9 月之前，我对 ACM-ICPC 关心甚少。但是，我注意到我校队员学习、训练没有统一的资料，也没有按照竞赛所需知识体系全面系统培训新队员。2011-2012 年度的学生教练们做了一个较详细的培训计划，每周都会给 2011 级新队员上课，也会对老队员进行训练，辛辛苦苦忙活了一年——但是这些知识是根据他们个人所掌握情况来给新生讲解的，新生也是杂七杂八看些资料和做题。在培训的规范性上欠缺很多，当然这个责任不在学生教练。2011 年 9 月，我曾给老队员提出编写培训资料这个任务，一是老队员人数少，有的还要去百度等企业实习；二是老队员要开发、改造 OJ；三是培训新队员也很耗费精力，因此这项工作虽很重要，但却不是那时最迫切的事情，只好被搁置下来。

2012 年 8 月底，2012 级新生满怀梦想和憧憬来到学校，部分同学也被 ACM-ICPC 深深吸引。面对这个新群体的培训，如何提高效率和质量这个老问题又浮现出来。市面现在已经有了各种各样的 ACM-ICPC 培训教材，主要算法和解题思路都有了广泛深入的分析和讨论。同时，互联网博客、BBS 等中也隐藏着诸多大牛对某些算法的精彩论述和参赛感悟。我想，做一个资料汇编，采撷各家言论之精要，对新生学习应该会有较大帮助，至少一可以减少他们上网盲目搜索的时间，二可以给他们构造一个相对完整的知识体系。

感谢 ACM-ICPC 先辈们作出的杰出工作和贡献，使得我们这些后继者们可以站在巨人的肩膀上前行。

感谢校集训队各位队员的无私、真诚和抱负的崇高使命感、责任感，能够任劳任怨、以苦为乐的做好这件我校的开创性工作。

唐远新

2012 年 10 月

编写说明

本资料为哈尔滨理工大学 ACM-ICPC 集训队自编自用的内部资料，不作为商业销售目的，也不用于商业培训，因此请各参与学习的同学不要外传。

本分册大纲由黄李龙编写，内容由程宪庆、周洲等分别编写和校核。

本分册内容大部分采编自各 OJ、互联网和部分书籍。在此，对所有引用文献和试题的原作者表示诚挚的谢意！

由于时间仓促，本资料难免存在表述不当和错误之处，格式也不是很规范，请各位同学对发现的错误或不当之处向acm@hrbust.edu.cn邮箱反馈，以便尽快完善本文档。在此对各位同学的积极参与表示感谢！

哈尔滨理工大学在线评测系统（Hrbust-OJ）网址：http://acm.hrbust.edu.cn，欢迎各位同学积极参与AC。

国内部分知名 OJ：

杭州电子科技大学：http://acm.hdu.edu.cn

北京大学：http://poj.org

浙江大学：http://acm.zju.edu.cn

以下百度空间列出了比较全的国内外知名 OJ:

http://hi.baidu.com/leo_xxx/item/6719a5ffe25755713c198b50

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

目　录

第3章 图论

3.1 图的基本概念

编写：程宪庆　　校核：周洲

图的概念类似于地图，地图上有城市和道路，图可以用来表示一个个体集合以及这些个体之间的关系，个体可以指实在的物体、城市、或某些状态等，对应的关系则为物体之间的联系、交通道路、状态之间的转换关系等等。个体叫做顶点，关系叫做边。图是一个网状的抽象结构，很多问题都可以描述成包括若干顶点和边的图上的问题。

从关系的有向性上，可以分为有向图和无向图。

从边的性质上，可以分为有权图和无权图。权指两个顶点之间边的某种属性值，比如两个城市之间的道路有长度或者路费之类的代价。

从边数上，可以分为稠密图和稀疏图。假设一个图中的顶点数为 V，那么如果两个点之间只能有一条双向边，那么最多可能有 n*(n-1)/2 条边，如果要求所有的点之间都是可达的，那么最少要求有 n-1 条边，可以看到，最多和最少边数相差很大，所以稀疏图和稠密图所占空间也差很多。

图论问题的常见出错问题：题目中的各种对图的约束条件，如图的连通性，图的边的唯一性，拓扑排序的无回路性，最短路的的无负权，无负环，0 顶点图或单顶点图等等。

3.1.1 图的表示

图在程序中的表示方法主要有邻接矩阵和邻接表等几种，邻接矩阵和邻接表是比较常用且基础的两种。

邻接矩阵适用于稠密图或需要很快判两个顶点间是否有边或边权值是多少的情况。

邻接表在稀疏图的表示上更加节省内存而且在边的遍历上比邻接矩阵更方便。

邻接矩阵使用一个二维矩阵，矩阵的两个维度均为图的顶点数，下标表示顶点的编号。假设使用矩阵 graph[N][N]来表示一个图，对于无权图：可以使用 graph[i][j]表示 i 顶点和 j 顶点之间有边，反之使其为 0 表示两个顶点之间无边。对于有权图，则全 graph[i][j]等于相应的边的权值。对于无向图 graph[i][j]==graph[j][i]。

使用邻接矩阵表示无权图时可以使用 bool 型或 char 型数据来达到节省内存的目的。

邻接表使用一个线性表，表中的每个元素对应图的一个顶点，并且保存一个链表，链表的每个节点表示与从该顶点出发的一条边，链表的节点中保存与边有关的所有信息如该边指向的顶点，边的权值等。

下图中右图为左图的邻接表表示，第一列表示每个顶点的头结点，第二列为每个顶点发出的第一条边，边中保存了该边所指向的点的编号。在输入图的时候，每次需要知道要当前要插入的边的两个连接点与该边的其他需要的信息，然后在相应顶点所指向的链表上插入一个节点。

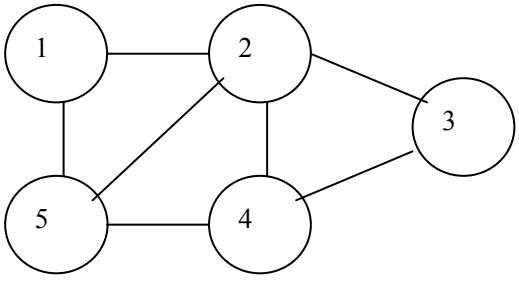


图 1-1

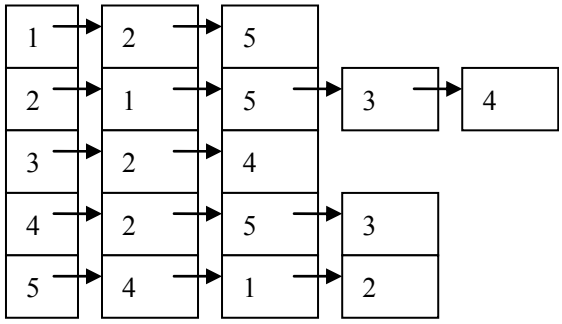


图 1-2

可以看出邻接矩阵需要的空间为 O(n^2)，而邻接表需要 O(V+E)，所以邻接表在稀疏图上的表示有着非常大的空间优势。

邻接表表示的一种实现：

```
struct Arc
{
   int next_arc;
   int point;
};
int node[V];
struct Arc arc[E];
```

使用 node 存储每个顶点， arc 存储每条边，node[i]表示第 i 个顶点指向的第一条边在 arc 中的位置，next_arc 表示和这条边同样出发点的下一条边在 arc 中的位置。

每个 node[i]都表示一个用数组实现的表示边的链表的表头，插入一条新边的过程，就是在该链表上插入一个节点，这里的链表是用数组实现，当然也可以使用动态分配去实现。

加入新边：

```
void AddEdge(int u,int v)
{
   arc[EdgeCount].next_arc=node[u];
   arc[EdgeCount].point=v;
   node[u]=EdgeCount;
   EdgeCount++;
}
```

`EdgeCount` 用来表示总共加入的边的数量，初始化为 `0`.

3.1.2 广度优先搜索

图的搜索是对一个图进行有序的遍历，以获得图中的*有用信息*，搜索是图论算法的基础。

广度优先搜索即在广度上对图进行遍历。

从图的一个顶点出发，首先访问该点，然后依次访问从该点出发指向的那些点，再按访问的顺序依次访问那些点所指向的点，比如上图：从 1 点出发，首先访问 1，然后访问 1 指向的 2、5，再访问 2 指向的 3、4，再去访问 5 指向的点，而 5 指向的 4 已经被访问了，到此所有的点都被访问过，对该图的广度优先遍历到此结束。

广度优先遍历是按每个点到源点的距离从小到大的顺序去依次遍历每个顶点，所以可以找到源点到每个点的无权最短路径。

广度优先搜索使用队列实现，每次将当前 访问节点指向的未访问节点依次加入队列，比如在访问 1 的时候依次将 2、5 入队，访问 1 后将 2 从队列中取出，再将与 2 相连的 3、4 入队，这样即实现了“一层一层”去遍历一个图，类似于二叉树的层次遍历。

因为每个点要入队、出队一次，每条边判断的次数也是一次，所以广度优先搜索的时间复杂度为 O(V+E)。

使用 1.1.1 中的邻接表实现的广度优先搜索：

```
int que[V];//数组模拟队列
int vis[V];//标记数组，vis[i]==1表示i已入队，0表示未入队
int front,rear;//队首和队尾指针

void bfs()
{
   front=rear=0;
   memset(vis,0,sizeof(vis));

   que[rear++]=0;//假设从0号顶点开始广度搜索，实际也可能没有0号顶点，
   //根据实际情况决定即可
```

```
    vis[0]=1;

    while(front<que)
    {
        int cur_node=que[front++];//取队首顶点
        int edge;
        //遍历队首顶点邻接的每一条边，判断该边指向顶点是否已入队
        for(edge=node[cur_node];edge!=-1;edge=arc[edge].next_arc)
        {
            if(!vis[arc[edge].point])
            {
                que[rear++]=arc[edge].point;
                vis[arc[edge].point]=1;
            }
        }
    }
}
```

3.1.3 深度优先搜索

深度优先搜索与广度不同，首先从一个点尽可能的走远，直到无法继续前进，再依次回退并试图寻找其他路径，类似于我们在玩走迷宫游戏时所经常使用的方法，比如上图，首先从 1 点出发，访问 2 点，再访问 3 点、4 点、5 点，然后无法继续前进了，就退回，判断 4 点是否有除 5 点以外可访问的未访问点，有的话再前进，没有就继续回退。

深度优先搜索可以使用递归实现，本质上是使用栈来实现，如果要用非递归的程序来实现深度优先搜索，需要主动的使用栈。

与广度优先搜索类似，可以证明深度优先搜索的时间复杂度也为 O(V+E)。

使用上述邻接表实现的深度优先搜索：

```
int vis[V];
void dfs(int v)
{
    vis[v]=1;
    int edge;
    for(edge=node[v];edge!=-1,edge=arc[edge].next_arc)
    {
        if(!vis[arc[edge].point])
        {
            dfs(arc[edge].point);
        }
    }
}
```

3.2 拓扑排序

编写：程宪庆　　校核：周洲

3.2.1 基本原理

拓扑排序是应用于**有向无回路图(Direct Acyclic Graph,简称 DAG)**上的一种排序方式，对一个有向无回路图进行拓扑排序后，所有的顶点形成一个序列，对所有边(u,v)，满足 u 在 v 的前面。该序列说明了顶点表示的事件或状态发生的整体顺序。比较经典的是在工程活动上，某些工程完成后，另一些工程才能继续，此时可以以工程为顶点，工程间的依赖关系为边建立图，用拓扑排序求得所有工程的合理执行顺序。

对一个 DAG 进行拓扑排序有两种方法，分别利用广度优先搜索与深度优先搜索。

首先介绍**顶点的度**，一个顶点的度指与该点相连的边的数量，对于有向图，一个顶点的度分为入度与出度，入度为指向该顶点的边的数量，出度即从该点出发的边的数量。

使用广度搜索：进行拓扑排序时，每次可以拿出的顶点一定是入度为 0 的点，即没有被指向的点，因为这样的点表示的事件没有依赖，在一个入度为 0 的点表示的事件执行完

之后，它所指向的顶点所依赖的点就少了一个，所以我们可以先将所有入度为 0 的点加入一个队列中，然后依次将它们所指向的点的入度减 1，再将入度变为 0 的点也依次加入队列，这样最后就可以得到一个拓扑有序的序列。

使用深度搜索：在对一个 DAG 进行深度优先搜索时，对于图上的一条边 u,v，一定有 v 比 u 更早退出 DFS 过程，而拓扑排序的顺序正好相反，对于一条边 u,v，需要 u 排在 v 前面，所以可以利用 DFS，将所有的点按照退出 DFS 的过程倒序排列，即得到一个图的拓扑序。

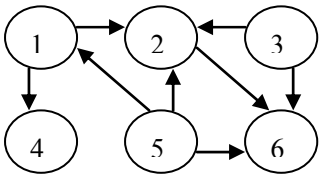


图 1.2.1

如对 1.2.1 进行拓扑排序，首先使用广度优先的方法：

找到所有入度为 0 的点，并加入队列，此时 3、5 被加入队列
取队首，即 3 顶点，将 2、6 的入度减 1
再取队首 5 顶点，将 1、2、6 的入度再减 1，此时 1 的入度为 0，加入队列
取队首 1 顶点，将 2、4 顶点的入度减 1，此时 2、4 的入度均为 0.加入队列
取队首 2 顶点，将 6 顶点的入度减 1，此时 6 的入度为 0，也加入队列
取队首 4 顶点，4 顶点没有指向任何顶点
取队首 6 顶点，6 顶点也没有指向任何顶点
此时队列为空，且所有顶点均参与了排序，此时得到一个拓扑序 3、5、1、2、4、6.

使用深度搜索的方法：
假设从 1 开始进行深度优先搜索，
搜索顺序依次为：1、2、6、4，或 1、4、2、6
此时并没有搜索到全部顶点，所以从下一个没有搜索过的顶点再执行深度优先搜索，
剩下的点的搜索顺序可以是 3、5 或者 5、3
这样根据搜索结束的顺序，即可以得到一个该图的拓扑序：3、5、1、2、6、4。

3.2.2 模板代码

```
struct Arc
{
   int point;
   int next_arc;
};

Arc arc[50005];
int node[5005];
int digree[5005];
int top[5005];

int main()
{
 int n,m;
 scanf("%d%d",&n,&m);
   queue<int>q;
```

```
    for(int i=1;i<=m;i++)
    {
        int a,b;
        scanf("%d%d",&a,&b);

        arc[i].next=node[a];
        arc[i].point=b;
        node[a]=i;
        digree[b]++;
    }

    for(int i=1;i<=n;i++)
    {
        if(digree[i]==0)
        {
            q.push(i);
        }
    }
    int l=0;
    while(!q.empty())
    {
        int x=q.front();
        top[l++]=x;//将 x 加入到拓扑序中
        q.pop();
        for(int e=node[x];e!=-1;e=arc[e].next)
        {
            digree[arc[e].point]--;
            if(digree[arc[e].point]==0)
            {
                q.push(arc[e].point);
            }
        }
    }
 return 0;
}
```

3.2.3 经典题目

1．题目出处/来源

HDOJ 1285

2．题目描述

有 N 个比赛队（1<=N<=500），编号依次为 1，2，3，。。。。，N 进行比赛，比赛结束后，裁判委员会要将所有参赛队伍从前往后依次排名，但现在裁判委员会不能直接获得每个队的比赛成绩，只知道每场比赛的结果，即 P1 赢 P2，用 P1，P2 表示，排名时 P1 在 P2 之前。现在请你编程序确定排名。

3．分析

一道简单的拓扑排序基础题目，唯一可能出现问题的地方就是题目要求同样拓扑序的两个编号小的要在前面，这点可以通过将普通拓扑排序中的队列改为使用优先队列或者堆来实现。

4．代码

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<queue>
using namespace std;

int graph[505][505];//表示图的数组
int digree[505];//存储每个顶点的入度

int main()
{
    int n,m;
    while(scanf("%d%d",&n,&m)!=EOF)
```

```
    {
        memset(graph,0,sizeof(graph));
        memset(digree,0,sizeof(digree));

        for(int i=0;i<m;i++)
        {
            int u,v;
            scanf("%d%d",&u,&v);
            if(!graph[u][v])//此处要注意，防止有重复边
            {
                graph[u][v]=1;
                digree[v]++;
            }
        }

        priority_queue<int,vector<int>,greater<int> >q;//使用 STL 中的优先队列，这种方式是
                                                       从小到大，注意最后一个>前面的空格，这个必须有。
        for(int i=1;i<=n;i++)
        {
            if(digree[i]==0)q.push(i);
        }

        bool first=1;
        while(!q.empty())
        {
            int cur=q.top();
            q.pop();
            if(first){cout<<cur;first=0;}
            else cout<<' '<<cur;

            for(int i=1;i<=n;i++)
            {
                if(graph[cur][i])
                {
                    digree[i]--;
                    if(digree[i]==0)
                    {
                        q.push(i);
                    }
                }
            }
        }
        printf("\n");
    }
    return 0;
}
```

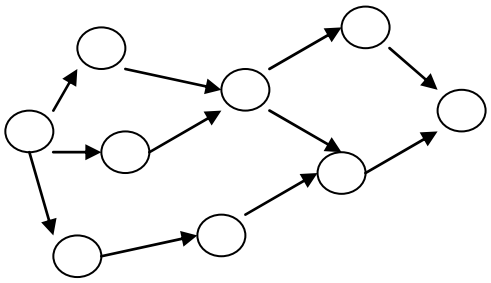
3.3 活动网络(AOE 网络)

编写：程宪庆　　校核：周洲

AOE 网络是指用边表示活动的网络，用弧表示一个活动，弧头表示活动结束，弧尾表示活动开始，弧长表示活动所需时间，每个顶点叫做一个事件，表示以其为弧尾的活动的结束和以其为弧头的活动的开始，一个活动结束相邻的下一个活动才能开始。

AOE 网络中的问题一般为求解在一个活动网络中所有活动全部完成所需的最少时间及影响该时间的关键活动。

例如用下图的网络表示一项工程，每条弧表示工程中的一个环节，弧长表示完成该环节所需时间，要求完成该项工程所需时间，显然就是要求从开始顶点到结束顶点所经过的最长路径，因为图是*有向无回路图（DAG）*，所以可以使用拓扑排序加动态规划的方法解出该时间。



以 a[i]表示第 i 个事件的最早开始时间，a[0]为 0，对原图进行拓扑排序，在取出一个入度为 0 的顶点之后要对它所指向的点的入度进行更新，在此时同时更新被指向的点所表示的事件完成所需时间为指向它的点所需完成时间+弧长最大的一个，如在找出入度为 0 的顶点 i 的时候对弧(i,j)进行更新，首先更新点 j 的入度为原值减 1，然后执行 a[j]=MAX(a[j],a[i]+g[i][j]);g[i][j]表示弧(i,j)完成所需时间。

而要找出影响该时间的关键活动，则需要知道哪些活动的开始时间是不能推迟的，这里可以通过上面已经求得的项目完成时间，从结束点开始向开始点进行逆向拓扑排序，在本次拓扑排序中求得每个顶点表示的事件的最晚开始时间，然后判断每个点的开始时间和最晚开始时间，如果两个时间值相等，则说明该以该事件开始的某些活动是不能推迟的，这样的活动所组成的从开始点到结束点的路径叫做***关键路径***，关键路径上的每一个事件都满两个时间相等，这样的路径上的弧所表示的活动都是影响项目完成速度的活动。

3.4 最小生成树 Prim

编写：程宪庆　　校核：周洲

在对一个无向图进行遍历时，根据遍历时每个顶点的前趋顶点和后继顶点间的关系，保留搜索时经过的边而放弃回边（即搜索时当前点与一个已搜索过的点之间的边），可以得到一棵树，该树叫做图的生成树。

生成树包含原图中的所有顶点（假设为 V 个），但只有 V-1 条边，它保证了无向图中每两个顶点之间都可以连通，且使用了最少的边数。

如果图是有权图，则每棵生成树的 N-1 条边有一个权值和，使这个权值和最小的生成树叫做最小生成树。

3.4.1 基本原理

构造最小生成树的 Prim 算法将原图的顶点分为两部分，假设原顶点集为 V，将其分为 S 与 V-S，S 为已经确定在最小生成树中的顶点，在算法的最初将任意一个节点作为 S 中的唯一一个元素，之后每次在 V-S 中寻找距离 S 中的点最近的一个顶点，作为下一个加入最小生成树中的节点，直至所有 N 个节点全部加入到最小生成树中后，最小生成树构造完毕。

对于一个使用邻接矩阵的图 graph[N][N]，可以使用一个数组 low[N]保存每个顶点到已加入最小生成树中的所有点的最小距离，每次寻找这个距离最小的一个点加入到最小生成树中，再根据这个点到其他没有加入生成树中的点的距离去更新其他点的 low 值，直到所有的点都加入到了最小生成树中。

举例说明，对于下图：

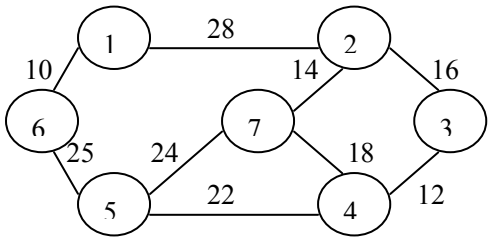


图 1.4.1

首先将 1 作为生成树的第一个点，然后计算每个点距离该未完成的树中顶点的最小距离，因为现在树中只有一个点，所以每个顶点到它的距离就是到生成树的最小距离，无法到达的记为无穷大，可以使用一个大整数表示。

根据这个规则，首先是 6 被加入，然后判断其他点到 6 的距离是否比原来到生成树的最小距离小，如果小，就将其更新。

6 加入之后 5 到该树的距离变为了 25，是当前最小的，所以下一个将 5 加入生成树，然后依次是 4、3、2、7，分别加入最小生成树中，最小生成树构成完毕，需要最小代价也求出来了。

因为每次加入一个顶点，要加入 N 个点，每次加入时寻找最近点需要遍历每个点，更新也需要遍历每个点，所以 Prim 需要 O(N^2)的时间。

Prim 算法对于使用邻接矩阵表示的图来说非常方便实现，所以在求稀疏图的最小生成树的时候常用 Prim 算法。

3.4.2 模板代码

```
#define INF 0x1f1f1f1f//定义一个整数常量，表示无穷大
//prim 函数返回得到的最小生成树的 n-1 条边的权值和
//参数 cost 为表示图的矩阵，n 为顶点个数
int prim(int cost[][200],int n)
{
    //low 表示每个点到生成树的最小距离，vis 表示一个点是否已加入生成树中
    int low[10000],vis[10000]={0};
    int i,j,p;
    int min,res=0;
    vis[0]=1;
    for(i=1;i<n;i++)low[i]=cost[0][i];
    for(i=1;i<n;i++)
    {
        min=INF;p=-1;
        for(j=0;j<n;j++)
        {
            if(0==vis[j]&&min>low[j])
            {
                min=low[j];
                p=j;
            }
        }
        //min==INF 说明找不到能够加入的点了，说明图是不连通的
        if(min==INF)return -1;

        res+=min;
        vis[p]=1;

        for(j=0;j<n;j++)
        {
            if(0==vis[j]&&low[j]>cost[p][j])
            {
                low[j]=cost[p][j];
            }
```

```
        }
    }
    return res;
}
```

3.4.3 经典题目

3.4.3.1 题目 1

1．题目出处/来源

HDOJ 1102

2．题目描述

有 N 个村庄，标记为 1-N，你现在要去建一些街道，使每两个村庄之间都能够连通，如果两个村庄 A、B 之间有一条街道，或者有一个村庄 C，C 与 A 和 C 与 B 之间都有街道相连，我们就说 A 和 B 是连通的，现在已知一些村庄之间的街道是已经修好的，你的任务是建设其他的街道，目的是让所有的村庄都是连通的，并且街道总长度最小。

3．分析

将已经修好的街道间的花费设为 0，然后使用 prim 算法求解最小生成树。

4．代码

```
#include<iostream>
#include<cstdio>
using namespace std;
#define INF 0x1f1f1f1f//定义一个整数常量，表示无穷大
//prim 函数返回得到的最小生成树的 n-1 条边的权值和
//参数 cost 为表示图的矩阵，n 为顶点个数
int prim(int cost[][200],int n)
{
    //low 表示每个点到生成树的最小距离，vis 表示一个点是否已加入生成树中
    int low[10000],vis[10000]={0};
    int i,j,p;
    int min,res=0;
    vis[0]=1;
    for(i=1;i<n;i++)low[i]=cost[0][i];
    for(i=1;i<n;i++)
    {
        min=INF;p=-1;
        for(j=0;j<n;j++)
        {
            if(0==vis[j]&&min>low[j])
            {
                min=low[j];
                p=j;
            }
        }
        //min==INF 说明找不到能够加入的点了，说明图是不连通的
        if(min==INF)return -1;

        res+=min;
        vis[p]=1;

        for(j=0;j<n;j++)
        {
            if(0==vis[j]&&low[j]>cost[p][j])
            {
                low[j]=cost[p][j];
            }
        }
    }
    return res;
}

int main()
{
    int n;
    int a[200][200];
```

```
    while(cin>>n)
    {
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                scanf("%d",&a[i][j]);
            }
        }
        int q;
        scanf("%d",&q);
        for(int i=0;i<q;i++)
        {
            int u,v;
            scanf("%d%d",&u,&v);
            a[u-1][v-1]=a[v-1][u-1]=0;
        }
        int res=prim(a,n);
        printf("%d\n",res);
    }
    return 0;
}
```

3.4.3.2 题目 2

1．题目出处/来源

HDOJ 2489

2．题目描述

对于一棵顶点和边都有权值的树，使用下面的等式来计算 Ratio

$$Ratio = \frac{\sum edge\ weight}{\sum node\ weight}$$

给定一个 n 个顶点的完全图及它所有顶点和边的权值，找到一个该图含有 m 个顶点的子图，并且让这个子图的 Ratio 值在所有 m 个顶点的树中最小。

3．分析

对于同一个 m 个顶点的子图，顶点的权值和是一定的，所以要让 Ratio 最小，只要该树的所有边权值和最小，所以只要找出该子图的最小生成树即可。

于是可以对这个 n 个顶点的图进行深度优先搜索，枚举所有 m 个点的子图，然后对这 m 个点进行求最小生成树，找到该子图中的 Ratio，然后在所有 m 个点的子图中找到 Ratio 的最小值。需要注意的一个问题是在求解过程中会出现浮点数除法，此处要判断精度。

4．代码

```
#include<set>
#include<map>
#include<cmath>
#include<queue>
#include<stack>
#include<vector>
#include<cstdio>
#include<cstring>
#include<iostream>
#include<algorithm>

using namespace std;

#define INF 0x1f1f1f1f
#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAX(a,b) ((a)>(b)?(a):(b))

int edge_wei[20][20];//边权
int node_wei[20];//点权
bool flag[20];
bool fl[20];
```

```
double res;
int dfs_cnt;
bool arr[20];//result array

int n,m;

int mst()//求最小生成树的函数
{
 int ret=0;
 int low[20]={0};
 int sta;
 for(int i=1;i<=n;i++)
 {
     if(fl[i])
     {
         sta=i;
         break;
     }
 }
 low[sta]=0;
 flag[sta]=1;
 for(int i=1;i<=n;i++)
 {
     if(fl[i])
     {
         low[i]=edge_wei[sta][i];
     }
 }
 for(int i=1;i<m;i++)
 {
     int min=INF;
     int loc;
     for(int j=1;j<=n;j++)
     {
         if(fl[j]&&!flag[j]&&low[j]<min)
         {
             min=low[j];
             loc=j;
         }
     }
     flag[loc]=1;
     ret+=low[loc];
     for(int j=1;j<=n;j++)
     {
         if(fl[j]&&!flag[j])
         {
             if(edge_wei[loc][j]<low[j])
             {
                 low[j]=edge_wei[loc][j];
             }
         }
     }
 }
 return ret;
}

void dfs(int v)
{
 fl[v]=1;
 dfs_cnt++;
 if(dfs_cnt==m)
 {
     memset(flag,0,sizeof(flag));
     int r=mst();
     int sum=0;
     for(int i=1;i<=n;i++)
     {
         if(fl[i])
```

```
            {
                sum+=node_wei[i];
            }
        }
        double res_tmp=(double)r/(double)sum;
        if(res_tmp-res<-(1e-9))//精度判断
        {
            res=res_tmp;
            for(int i=1;i<=n;i++)
            {
                arr[i]=fl[i];
            }
        }
        fl[v]=0;
        dfs_cnt--;
        return;
    }
    for(int i=v+1;i<=n;i++)
    {
        dfs(i);
    }
    fl[v]=0;
    dfs_cnt--;
}

int main()
{
    while(scanf("%d%d",&n,&m),n||m)
    {
        for(int i=1;i<=n;i++)
        {
            scanf("%d",&node_wei[i]);
        }
        for(int i=1;i<=n;i++)
        {
            for(int j=1;j<=n;j++)
            {
                scanf("%d",&edge_wei[i][j]);
            }
        }

        res=(double)INF;
        for(int i=1;i<=n-m+1;i++)
        {
            memset(fl,0,sizeof(fl));
            dfs_cnt=0;
            dfs(i);
        }

        int fir=1;
        for(int i=1;i<=n;i++)
        {
            if(arr[i])
            {
                if(fir)
                {
                    fir=0;
                    printf("%d",i);
                }
                else
                {
                    printf(" %d",i);
                }
            }
        }
        printf("\n");
    }
    return 0;
}
```

3.5 最小生成树 Kruskal

编写：程宪庆　　校核：周洲

3.5.1 基本原理

克鲁斯卡尔算法（Kruskal）每次选取没有参与构造最小生成树并且加入之后不会构成回路的边中权值最小的一条作为最小生成树的一条新边，直至选择了 V-1 条边，此时便构成了一棵最小生成树。为了判断一个边加入时是否会构成回路以决定是否选择该边，一般使用并查集来将所有已经互相连通的顶点加入到一个集合中，在决定是否将一条边作为最小生成树的新边时，只需要判断该边的两个顶点是否在同一集合中，如果是则舍去该边，否则将该边的两个点并至同一集合并将该边作为生成树的新边。

由于 kruskal 算法操作的是边，所以比较适合于稀疏图的最小生成树求解，每次取最小边可以使用排序预处理或者使用堆或优先队列实现，所需时间为 O(ElogV)，空间复杂度为 O(E)，而 prim 算法需要 O(N^2)的时间和空间。

Prim 算法与 Kruskal 算法均使用了贪心思想。

3.5.2 模板代码

```
int p[10005];//表示集合的数组
int r[10005];//按秩合并所需的秩

//并查集的查找
int find(int v)
{
    if(v!=p[v])p[v]=find(p[v]);
    return p[v];
}

//并查集的合并操作
void join(int u,int v)
{
    int a=find(u);
    int b=find(v);
    if(a==b)return;

    if(r[a]<r[b])
    {
        p[a]=b;
    }
    else if(r[a]>r[b])
    {
        p[b]=a;
    }
    else
    {
        p[a]=b;
        r[b]++;
    }
}

//初始化并查集
void init_set(int n)
{
    int i;
    for(i=1;i<=n;i++)
    {
        p[i]=i;
        r[i]=1;
    }
}
```

```
//定义边结构
struct Edge
{
    int u;
    int v;
    int weight;
};

struct Edge edge[50005];

//快速排序，也可以使用 qsort 或 sort
void quick_sort(struct Edge* start,struct Edge* end)
{
    if(start>=end)return;
    struct Edge* loc=start;
    struct Edge* iterator;
    struct Edge tmp;
    for(iterator=start;iterator!=end;iterator++)
    {
        if(iterator->weight<(end-1)->weight)
        {
            tmp=*loc;
            *loc=*iterator;
            *iterator=tmp;

            loc++;
        }
    }
    tmp=*loc;
    *loc=*(end-1);
    *(end-1)=tmp;

    quick_sort(start,loc);
    quick_sort(loc+1,end);
}

int kru(int n,int m)//kruskal 算法主要部分，传入 n 顶点数和 m 边数
{
    init_set(n);
    quick_sort(edge,edge+m);
    int i;
    int ret=0;//生成树的总权值
    int cnt=0;//已加入最小生成树的边的数量
    for(i=0;i<m;i++)
    {
        int u=edge[i].u;
        int v=edge[i].v;
        //如果两个顶点不在同一集合中，则不会形成环，于是该边被加入生成树
        if(find(u)!=find(v))
        {
            cnt++;
            ret+=edge[i].weight;
            join(u,v);
        }
        if(cnt==n-1)return ret;//已找到 n-1 条边，生成树构造完毕
    }
    return -1;
}
```

3.5.3 经典题目

1．题目出处/来源

POJ 1797

2．题目描述

题意为求从一个图的点 1 到点 n 的所有路径中权值最小边的最大值

3．分析

首先可以想到枚举所有的路径，找出每条路径中的最小值，然后比较出最大值，可以使用广度优先搜索的方法加上一些优化判断来解决。

第二种方法是依次寻找图中从大到小的所有边，直到将 1 点和 n 点之间连接出一条路径，此过程即为构造一棵最大权值生成树的过程。

4．代码

使用 kruskal 最大生成树的方法：

```
#include<iostream>
#include<cstdio>
#include<algorithm>
using namespace std;

struct Edge
{
    int u;
    int v;
    int w;
};

bool cmp(Edge e1,Edge e2)
{
    return e1.w>e2.w;
}

Edge e[1000005];

int p[1005];
int r[1005];

int find(int u)
{
    if(u!=p[u])p[u]=find(p[u]);
    return p[u];
}

void join(int u,int v)
{
    int a=find(u);
    int b=find(v);
    if(a==b)return;
    if(r[a]>r[b])p[b]=a;
    else if(r[a]<r[b])p[a]=b;
    else
    {
        p[a]=b;
        r[b]++;
    }
}

int main()
{
    int t;
    int cse=1;
    scanf("%d",&t);
    while(t--)
    {
        int n,m;
        scanf("%d%d",&n,&m);
        for(int i=1;i<=n;i++)
        {
            p[i]=i;
            r[i]=1;
        }
        for(int i=0;i<m;i++)
        {
            scanf("%d%d%d",&e[i].u,&e[i].v,&e[i].w);
```

```
        }
        sort(e,e+m,cmp);
        int res;
        for(int i=0;i<m;i++)
        {
            if(find(1)==find(n))break;
            int u=e[i].u;
            int v=e[i].v;
            if(find(u)!=find(v))
            {
                res=e[i].w;
                join(u,v);
            }
        }
        printf("Scenario #%d:\n%d\n\n",cse++,res);
    }
    return 0;
}
```

广度优先搜索的方法：

```
#include<iostream>
#include<queue>
#include<cstdio>
#include<cstring>
using namespace std;

int n,m;
int edge_cnt;

int head[1005];
int next[1000005];
int point[1000005];
int weight[1000005];

int dis[1005];//dis 表示 1 点到该点路径中最小权值的最大值
bool fl[1005];

void add_edge(int u,int v,int w)
{
    next[edge_cnt]=head[u];
    point[edge_cnt]=v;
    weight[edge_cnt]=w;
    head[u]=edge_cnt;
}

void fun()
{
    memset(dis,0,sizeof(dis));
    memset(fl,0,sizeof(fl));

    queue<int>q;

    q.push(1);
    fl[1]=1;
    dis[1]=0x1f1f1f1f;//源点要初始化成无穷，其他点初始化成 0

    while(!q.empty())
    {
        int u=q.front();
        q.pop();
        fl[u]=0;

        for(int e=head[u];e!=-1;e=next[e])
        {
            int v=point[e];
            if(weight[e]<dis[u])//如果该边小于 u 点的 dis 值，那么从 u 点到 v 点的所有路径中的最小
权值的最大值就是该边的权值
            {
                if(weight[e]>dis[v])//如果此时该边权值大于 v 点原 dis 值，那么更新 v 点 dis 值为
```

```
weight[e]，否则 v 点 dis 值不变
                {
                    dis[v]=weight[e];
                    if(!fl[v])
                    {
                        q.push(v);
                    }
                }
            }
            else if(dis[v]<dis[u])//如果该边权值大于 v 点 dis 值，则 v 点 dis 值为 dis[u]、dis[v]
中大的一个
            {
                dis[v]=dis[u];
                if(!fl[v])
                {
                    q.push(v);
                }
            }
        }
    }
}

int main()
{
    int t;
    scanf("%d",&t);
    int sen=1;
    while(t--)
    {
        memset(head,-1,sizeof(head));
        edge_cnt=0;
        scanf("%d%d",&n,&m);
        for(int i=1;i<=m;i++)
        {
            int u,v,w;
            scanf("%d%d%d",&u,&v,&w);
            edge_cnt++;
            add_edge(u,v,w);
            edge_cnt++;
            add_edge(v,u,w);
        }
        fun();
        printf("Scenario #%d:\n%d\n\n",sen++,dis[n]);
    }
    return 0;
}
```

3.6 最短路 Dijkstra

编写：程宪庆　　校核：周洲

在一个有向或无向图中寻找两个顶点间的某种代价最小的路径的问题称为最短路问题，如果只是需要求两个点间的最短路或一个点到若干点间的最短路，则为**单源最短路**问题，如果需要知道每两个顶点间的最短路，则为所有顶点间的最短路问题。

3.6.1 基本原理

设我们的源点为 v0,考虑每个除 v0 以外的点到 v0 的最短距离，如果与 v0 没有直接的边相连的话，则一定要通过其它的点间接才能到达 v0 或者根本就无法到达 v0，所以如果不存在负边的话，距离 v0 最近的点一定是与 v0 的有边直接相连并且距离最小的一个，设它为 v1,所以我们首先可以确定 v0 到 v1 的最短距离即为边(v0,v1)的权，而其他的顶点到 v0 的最短距离或者是它们与 v0 间直接连边的距离，或者是它们通过 v1 间接到达 v0 的距离，此时我们需要对所有 v1 指向的边进行一次判断，以确定它们是否可以通

过 v1 来经过更短的路径到达 v0，然后再在所有和 v0 直接或者间接相连的点中找到与其距离最小的一个顶点，设它为 v2,此时 v2 到 v0 的最短距离也可以得到确定，因为其它的点到 v0 的最小距离均不小于该距离，所以如果 v2 经过其它点到达 v0 也一定不短于这个距离，此时我们得到这样一个循环实现方式：

1.找到当前到源点距离最小的一个，可以确定源点到它的最短距离即为当前距离

2.对该点所指向的所有点进行判断是否经过该点间接到达源点的距离比原来更短

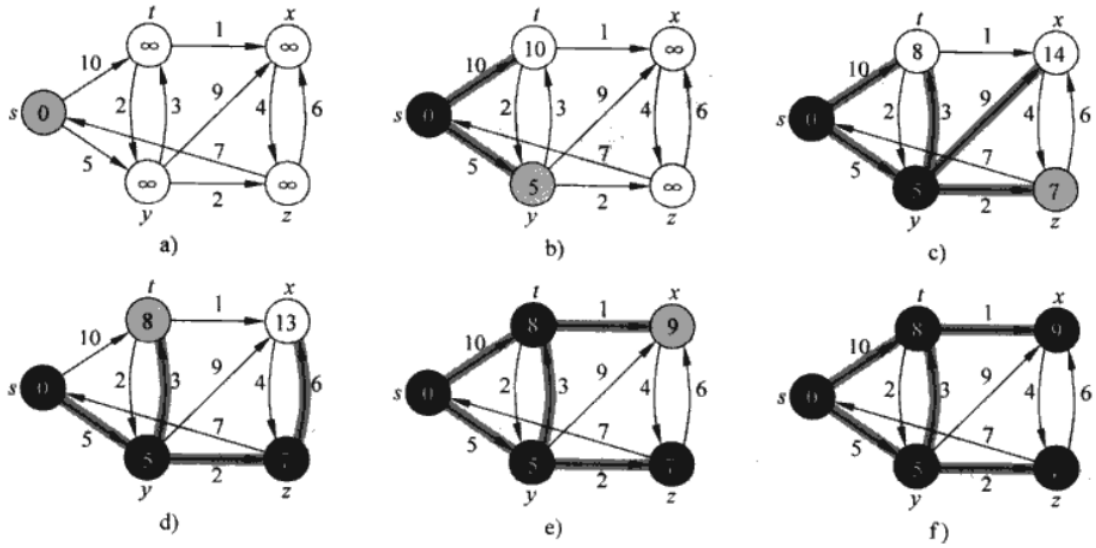
重复上述两种操作 n-1 次，便找出了源点到所有 n-1 个点的最短距离，因为找到源点到某点的最短距离要依赖于源点到其它点的最短距离，所以即使只是要求两个点间的最短距离，也要找到源点到所有点的最短距离。

如果使用邻接表存储图，则 Dijkstra 算法的过程如下：

1.初始化源点的最短距离为 0，将源点插入一个优先队列（可以使用堆）中，优先队列中的元素需要保存顶点编号与该顶点到源点的距离，排序规则为距离小的在前（根据实际要求也可能大的在前）。

2.从优先队列中取出队首元素，并将其标记为已找到最短距离，设其表示的顶点为 v1，对其指向的所有未确定最短距离的顶点 v2，以边(v1,v2)的权值+队首元素的距离值和 v2 的编号构造一个元素，并加入优先队列。

重复上述步骤直至队空，即可得到源点到每个点的最短距离。如只需找到到某个顶点的最短距离，则可在找到到该点的最短距离时便停止。



Dijkstra 算法的示例如上图：灰色顶点为每次找到的被确定为最短路径上的顶点，首先是源点 s，距离 s 最近的 y 点距离为 5，次近的为点 t 距离为 10，将 y 的最短距离确定后对 y 所指向的点进行判断，可以得到 y 到 s 的最短距离+(y,t)的距离比 t 原来到 s 的距离要短，于是将 t 的最短距离更新为 8，同理将 z 的距离更新为 7，x 的距离更新为 14，然后再寻找此时距离 s 最近的，为 z，距离是 7，如此循环，最终即可得到所有顶点到 s 的最短距离。但 Dijkstra 算法只能处理没有负边的情况，因为如果存在负边，那么当前确定的最短距离则不一定是最短距离，通过一条比它距离源点远的顶点以及一条很小的负边可能得到与源点更近的距离。

3.6.2 模板代码

邻接矩阵实现：

```
#define INF 0xfffffff
#define SIZE 150

int a[SIZE][SIZE]
```

```
int low[SIZE];

void DIJ(int n)//传入顶点个数 n，默认 0 为起点
{
    int i,j,k;
    low[0]=0;
    bool flag[SIZE]={0};
    flag[0]=1;
    for(i=1;i<n;i++)
    {
        low[i]=a[0][i];
    }

    for(i=1;i<n;i++)
    {
        int min=INF;
        for(j=0;j<n;j++)
        {
            if(flag[j]==0&&low[j]<min)
            {
                min=low[j];
                k=j;
            }
        }
        flag[k]=1;
        for(j=0;j<n;j++)
        {
            if(flag[j]==0&&a[k][j]+low[k]<low[j])
                low[j]=low[k]+a[k][j];
        }
    }
}
```

邻接表实现：

```
struct Arc
{
    int next_arc;
    int point;
    int weight;
};
int node[N];
struct Arc arc[M];

void insert_edge(int u,int v,int weight,int edge_num)
{
    arc[edge_num].next_arc=node[u];
    arc[edge_num].point=v;
    arc[edge_num].weight=weight;
    node[u]=edge_num;
}
//堆元素的结构
struct heap_elm
{
    int num;
    int dis;
};
//堆
struct heap_elm heap[M];
//在堆中插入一个元素，传入要插入的元素和堆的大小 len
void insert(struct heap_elm h,int* len)
{
    (*len)++;
    heap[*len]=h;
    int i=*len;
    while(i>0)
    {
        int j=(i>>1);
        if(heap[j].dis>heap[i].dis)
        {
            struct heap_elm tmp=heap[j];
```

```
            heap[j]=heap[i];
            heap[i]=tmp;

            i=j;
        }
        else break;
    }
}
//调整堆使其保持堆性质
void heapfi(int loc,int len)
{
    int left=(loc<<1);
    int right=left+1;
    int min_loc=loc;
    if(left<=len&&heap[left].dis<heap[min_loc].dis)
    {
        min_loc=left;
    }
    if(right<=len&&heap[right].dis<heap[min_loc].dis)
    {
        min_loc=right;
    }
    if(min_loc!=loc)
    {
        struct heap_elm tmp=heap[min_loc];
        heap[min_loc]=heap[loc];
        heap[loc]=tmp;

        heapfi(min_loc,len);
    }
}
//获得堆顶元素
struct heap_elm get()
{
    return heap[1];
}
//删除堆顶元素
void del(int* len)
{
    heap[1]=heap[*len];
    (*len)--;
    heapfi(1,*len);
}
//标记数组，表示一个顶点是否已找到最短距离
int vis[N];
//表示结果的数组，res[i]表示源点到 i 的最短距离
int res[N];
//Dijkstra 算法主要部分，传入顶点数 n，边数 m，源点 src
void dij(int n,int m,int src)
{
    memset(vis,0,sizeof(vis));
    int len=0;//初始化堆大小
    struct heap_elm h;
    h.dis=0;
    h.num=src;
    insert(h,&len);//将源顶插入堆
    while(len>0)
    {
        h=get();
        del(&len);
        if(vis[h.num])continue;//如果已找到最短距离则跳过
        res[h.num]=h.dis;
        vis[h.num]=1;
        int edge;
        for(edge=node[h.num];edge!=-1;edge=arc[edge].next_arc)
        {
            if(!vis[arc[edge].point])
            {
```

```
                struct heap_elm t;
                t.dis=h.dis+arc[edge].weight;
                t.num=arc[edge].point;
                insert(t,&len);
            }
        }
    }
}
```

3.6.3 经典题目

3.6.3.1 题目 1

1．题目出处/来源

HDOJ 3790

2．题目描述

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。

3．分析

比较基础的 Dijkstra 算法练习题，只是多了一个条件，可以使用邻接矩阵实现，但使用邻接表+优先队列实现是最好选择，SPFA 亦可。

4．代码

邻接矩阵的代码：

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;

#define INF 0x1f1f1f1f
#define SIZE 1005

int dis[SIZE][SIZE];
int cost[SIZE][SIZE];
int lowdis[SIZE];//每个点到源点的最短距离
int lowcost[SIZE];//在最短距离前提下的最小花费

void DIJ(int n,int s,int t)
{
    bool flag[SIZE]={0};
    flag[s]=1;
    for(int i=1;i<=n;i++)
    {
        lowdis[i]=dis[s][i];
        lowcost[i]=cost[s][i];
    }
    int k;
    for(int i=1;i<n;i++)
    {
        int mindis=INF;
        int mincost=INF;
        for(int j=1;j<=n;j++)
        {
            if(flag[j]==0)
            {
                if(lowdis[j]<mindis)
                {
                    mindis=lowdis[j];
                    mincost=lowcost[j];
                    k=j;
                }
                else if(lowdis[j]==mindis&&lowcost[j]<mincost)
                {
                    mindis=lowdis[j];
                    mincost=lowcost[j];
```

```
                    k=j;
                }
            }
        }
        flag[k]=1;
        for(int j=1;j<=n;j++)
        {
            if(flag[j]==0)
            {
                if(dis[k][j]+lowdis[k]<lowdis[j])
                {
                    lowdis[j]=lowdis[k]+dis[k][j];
                    lowcost[j]=lowcost[k]+cost[k][j];
                }
                else
if(dis[k][j]+lowdis[k]==lowdis[j]&&cost[k][j]+lowcost[k]<lowcost[j])
                {
                    lowdis[j]=lowdis[k]+dis[k][j];
                    lowcost[j]=lowcost[k]+cost[k][j];
                }
            }
        }
    }
    }

    int main()
    {
        int n,m;
        while(scanf("%d%d",&n,&m),n||m)
        {
            memset(dis,0x1f,sizeof(dis));
            memset(cost,0x1f,sizeof(cost));
            for(int i=1;i<=n;i++)
            {
                dis[i][i]=0;
                cost[i][i]=0;
            }
            for(int i=0;i<m;i++)
            {
                int u,v,d,c;
                scanf("%d%d%d%d",&u,&v,&d,&c);
                if(dis[u][v]>d||(dis[u][v]==d&&cost[u][v]>c))
                {
                    dis[u][v]=dis[v][u]=d;
                    cost[u][v]=cost[v][u]=c;
                }
            }
            int s,t;
            scanf("%d%d",&s,&t);
            DIJ(n,s,t);
            printf("%d %d\n",lowdis[t],lowcost[t]);
        }
        return 0;
    }
```

3.6.3.2 题目 2

1．题目出处/来源

hrbust 1339

2．题目描述

题目大意为两个人从同一个地方出发各自到不同的地方去，因为坐车的价钱和距离有关，所以两人能走的路越短越好，而且两个人同乘一辆车可以将花费进一步降低，现在给出每条路所连接的两个城市及该线路的花费以及两个人的出发点和他们各自的目的地，求他们需要的最小花费是多少。

3．分析

因为两个人同行的时候会让花费变小，所以他们会先尽量同行，再在某处分开各自朝

各自的目的地去，所以行走路线是一个 Y 形，所需花费即为 Y 形的三条线路所需花费之和，要让总花费最小，只要找到一个点，它到两个人的出发点以及他们各自的目的地三个地方的最短距离的和最小，那么此时三个距离的和就是他们所需的最小花费。
也就是说，只要分别找到三个点到所有点的最小花费，即可相应得到解。

4．代码

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<queue>
#include<cstring>
using namespace std;

#define INF 0x1f1f1f1f//无穷大，这里的无穷大不能太大，否则三个无穷大相加的时候会出现数据范围溢出

struct str//用于优先队列的元素
{
 int num;//在队列中的元素所在的点
 int cost;//源点到该点的花费
 str(int n,int c):num(n),cost(c){}//此为 C++中的构造函数
 str(){}
 friend bool operator < (str s1,str s2)//比较函数，用来定义优先队列中的优先级
 {
    return s1.cost>s2.cost;
 }
};

struct Arc
{
   int next_arc;
   int point;
   int cost;
};

Arc arc[20005];
int head[5005];
bool fl[5005];

int lowa[5005];
int lowb[5005];
int lowc[5005];
int C,A,B;

void dij(int src,int n,int* low)//low 表示求的是哪个到所有点的花费
{
   memset(fl,0,sizeof(fl));//标记数组
   priority_queue<str>q;//这里用了 STL 的优先队列容器，使用堆有时会更快，建议自己写堆结构
   q.push(str(src,0));
   int kk=0;
   while(kk<n&&!q.empty())
   {
      str s=q.top();
      q.pop();
      if(fl[s.num])continue;
      fl[s.num]=1;
      low[s.num]=s.cost;
      kk++;
      for(int e=head[s.num];e!=-1;e=arc[e].next_arc)
      {
         if(!fl[arc[e].point])
         {
            q.push(str(arc[e].point,arc[e].cost+s.cost));
         }
      }
   }
}
```

```
int main()
{
 int cse=1;
 int n,m;
 while(~scanf("%d%d",&n,&m))
 {
     memset(head,-1,sizeof(head));

     memset(lowc,0x1f,sizeof(lowc));
     memset(lowa,0x1f,sizeof(lowa));
     memset(lowb,0x1f,sizeof(lowb));

     scanf("%d%d%d",&C,&A,&B);

     for(int i=1;i<=m;i++)
     {
         int x,y,k;
         scanf("%d%d%d",&x,&y,&k);

         /////添加一条边
         arc[i].next_arc=head[x];
         arc[i].point=y;
         arc[i].cost=k;
         head[x]=i;

           /////添加反向边
         arc[m+i].next_arc=head[y];
         arc[m+i].point=x;
         arc[m+i].cost=k;
         head[y]=m+i;
     }

     dij(C,n,lowc);
     dij(A,n,lowa);
     dij(B,n,lowb);


     printf("Scenario #%d\n",cse++);
     int res=INF;
     if(lowc[B]>=INF||lowc[A]>=INF)
     {
         printf("Can not reah!\n");
         continue;
     }
     for(int i=1;i<=n;i++)
     {
         if(lowc[i]+lowa[i]+lowb[i]<res)
         {
             res=lowc[i]+lowa[i]+lowb[i];
         }
     }
     printf("%d\n",res);
     printf("\n");
 }
 return 0;
}
```

3.7 最短路 Bellman-Ford

编写：程宪庆　　校核：周洲

3.7.1 基本原理

对于源点 v0 到某个点的最短距离，Bellman-Ford 算法对其进行 N-1 次尝试松弛，一次松弛是指对于一条边(u,v)，判断 v0 到 u 的距离加上(u,v)的权值是否比 v0 到 v 原来的最短

距离短，如果是则更新 v0 到 v 的最短距离，Dijkstra 算法实际也使用了松弛操作，**松弛**是最短路问题的基本操作。

Bellman-Ford 算法的 **N-1 次尝试中每次都对所有 E 条边进行尝试松弛**，所以它的复杂度为 O(NE)，源点到每个点的最短路径中，最多经过除它们以外的 N-2 个顶点，所以 Bellman-Ford 算法对每个顶点进行 N-1 次松弛尝试，如果没有负环存在，则最后一定可以确定源点到每个顶点的最短距离，包括有负边存在的情况下。如果要判断是否有负环存在，对于 Bellman-Ford，可以运行第二次算法，判断是否还可再松弛，如果是，则说明有负环存在，对于 SPFA，如果有负环，则队列永远无法为空，因为每个顶点不可能入队超过 N 次，所以可以记下每个顶点入队次数，如果某个顶点入队超过 N 次，则说明有负环存在。

因为如果两个顶点都是从源点不可达的边，对其进行松弛尝试是没有意义的，所以 Bellman-Ford 算法可以使用队列进行一定的改进，每次只对当前源点可达的顶点进行松弛尝试。

首先将源点入队，然后重复从队首取出一个顶点，对所有该点指向的顶点进行松弛，如果松弛成功并且该不在队列中，直到队空，即可找出源点到所有顶点的最短距离。

3.7.2 模板代码

Bellman-Ford 算法：
使用邻接矩阵存储图

```
#define N 105

int res[N];//存储源点到每个顶点的最短距离值
int g[N][N];

void bellman(int n,int src)
{
    //初始化每个顶点的最短距离为无穷大
    memset(res,0x1f,sizeof(res));
    res[src]=0;//源点的最短距离为 0
    int i,j,k;
    for(i=1;i<n;i++)//松弛 n-1 次
    {
        //j、k 所在循环为对每条边进行松弛
        for(j=1;j<=n;j++)
        {
            for(k=1;k<=n;k++)
            {
                if(res[k]>res[j]+g[j][k])
                {
                    res[k]=res[j]+g[j][k];
                }
            }
        }
    }
}
```

存储图的边：

```
int res[N];//存储源点到每个顶点的最短距离值
struct Edge
{
    int u;
    int v;
    int t;
};

Edge edge[E];

bool bellman(int n,int m,int src)//n 点数，m 边数，src 源点
{
    //初始化每个顶点的最短距离为无穷大
```

```
    memset(res,0x1f,sizeof(res));
    res[src]=0;//源点的最短距离为 0
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            if(res[edge[j].u]+edge[j].t<res[edge[j].v])
            {
                res[edge[j].v]=res[edge[j].u]+edge[j].t;
            }
        }
    }
    for(int i=0;i<n;i++)//判断是否存在负环
    {
        for(int j=0;j<m;j++)
        {
            if(res[edge[j].u]+edge[j].t<res[edge[j].v])
            {
                return 1;
            }
        }
    }
    return 0;
}
```

SPFA 算法：

邻接矩阵表示：

```
#define N 105

int res[N];//存储源点到每个顶点的最短距离值
int g[N][N];
int cnt[N];//每个点入队次数，判断是否出现负环用

int que[N*N];//队列
bool in_que[N];//标记一个点是否已在队列中
int front;//队首位置
int rear;//队尾位置

void spfa(int n,int src)
{
    rear=front=0;
    que[++rear]=src;
    memset(res,0x1f,sizeof(res));
    memset(in_que,0,sizeof(in_que));
    res[src]=0;
    while(front<rear)
    {
        int cur=que[++front];
        in_que[cur]=0;
        int i;
        for(i=1;i<=n;i++)
        {
            if(res[cur]+g[cur][i]<res[i])
            {
                res[i]=res[cur]+g[cur][i];
                if(!in_que[i])
                {
                    que[++rear]=i;
                }
            }
        }
    }
}
```

邻接表表示：

```
int res[505];//存储源点到每个顶点的最短距离值
struct Arc
{
    int next_arc;
    int point;
```

```
    int t;
};
int node[5000];
struct Arc arc[6000];
int cnt[505];//顶点入队次数
bool fl[505];//标记顶点是否已入队

bool spfa(int n,int src)//这里将返回值设为 bool 以判断是否有负环
{
    //初始化每个顶点的最短距离为无穷大
    memset(res,0x1f,sizeof(res));
    memset(cnt,0,sizeof(cnt));
    memset(fl,0,sizeof(fl));

    res[src]=0;//源点的最短距离为 0
    queue<int>q;//使用了 STL 的队列容器
    q.push(src);//源点入队
    cnt[src]++;//源点队列次数+1
    fl[src]=1;//标记源点入队
    while(!q.empty())
    {
        int c=q.front();
        q.pop();
        fl[c]=0;//入队标记取消
        for(int e=node[c];e!=-1;e=arc[e].next_arc)
        {
            if(res[c]+arc[e].t<res[arc[e].point])
            {
                res[arc[e].point]=res[c]+arc[e].t;
                if(!fl[arc[e].point])
                {
                    q.push(arc[e].point);
                    cnt[arc[e].point]++;
                    if(cnt[arc[e].point]>=n)return 1;
                }
            }
        }
    }
    return 0;
}
```

3.7.3 经典题目

1．题目出处/来源

poj 3259

2．题目描述

有一个农场，里面有一些奇怪的单向的虫洞，通过虫洞的入口到出口可以让时间倒退，农场主想知道他是否能够通过农场中的路和虫洞走回出发点，并且使时间倒退，给出农场中的 N 块地、M 条路以及 W 个虫洞。

3．分析

由题意可以看出实际目标为寻找负值回路，所以可以使用 Bellman-Ford 或者 SPFA 去判断图中是否存在负权回路即可。

4．代码

Bellman-Ford:

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;

int res[505];//存储源点到每个顶点的最短距离值
struct Edge
{
    int u;
```

```
    int v;
    int t;
};

Edge edge[6000];

bool bellman(int n,int m,int src)//这里将返回值设为 bool 以判断是否有负环
{
    //初始化每个顶点的最短距离为无穷大
    memset(res,0x1f,sizeof(res));
    res[src]=0;//源点的最短距离为 0
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            if(res[edge[j].u]+edge[j].t<res[edge[j].v])
            {
                res[edge[j].v]=res[edge[j].u]+edge[j].t;
            }
        }
    }
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            if(res[edge[j].u]+edge[j].t<res[edge[j].v])
            {
                return 1;
            }
        }
    }
    return 0;
}

int main()
{
    int f;
    scanf("%d",&f);
    while(f--)
    {
        int n,m,w;
        scanf("%d%d%d",&n,&m,&w);
        for(int i=0;i<m;i++)
        {
            int s,e,t;
            scanf("%d%d%d",&s,&e,&t);
            edge[i].u=s,edge[i].v=e,edge[i].t=t;
            edge[i+m].u=e,edge[i+m].v=s,edge[i+m].t=t;
        }
        for(int i=0;i<w;i++)
        {
            int s,e,t;
            scanf("%d%d%d",&s,&e,&t);
            edge[i+2*m].u=s,edge[i+2*m].v=e,edge[i+2*m].t=-t;
        }
        if(bellman(n,2*m+w,1))printf("YES\n");
        else printf("NO\n");
    }
    return 0;
}
```

SPFA:

这里使用的是 SPFA 的邻接矩阵表示，如果使用邻接表效率会更高

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;

#define N 505
```

```
int res[N];//存储源点到每个顶点的最短距离值
int g[N][N];
int cnt[N];//每个点入队次数，判断是否出现负环用

int que[N*N];//队列
bool in_que[N];//标记一个点是否已在队列中
int front;//队首位置
int rear;//队尾位置

bool spfa(int n,int src)
{
    memset(cnt,0,sizeof(cnt));
    rear=front=0;
    que[++rear]=src;
    cnt[src]++;
    memset(res,0x1f,sizeof(res));
    memset(in_que,0,sizeof(in_que));
    res[src]=0;
    while(front<rear)
    {
        int cur=que[++front];
        in_que[cur]=0;
        int i;
        for(i=1;i<=n;i++)
        {
            if(res[cur]+g[cur][i]<res[i])
            {
                res[i]=res[cur]+g[cur][i];
                if(!in_que[i])
                {
                    que[++rear]=i;
                    cnt[i]++;
                    if(cnt[i]>=n)return 1;
                }
            }
        }
    }
    return 0;
}

int main()
{
    int f;
    scanf("%d",&f);
    while(f--)
    {
        int n,m,w;
        scanf("%d%d%d",&n,&m,&w);
        memset(g,0x1f,sizeof(g));
        for(int i=0;i<m;i++)
        {
            int s,e,t;
            scanf("%d%d%d",&s,&e,&t);
            g[s][e]=g[e][s]=(g[s][e]>t?t:g[s][e]);
        }
        for(int i=0;i<w;i++)
        {
            int s,e,t;
            scanf("%d%d%d",&s,&e,&t);
            g[s][e]=(g[s][e]<-t?g[s][e]:-t);
        }
        if(spfa(n,1))printf("YES\n");
        else printf("NO\n");
    }
    return 0;
}
```

3.8 所有顶点之间的最短路 Floyd

编写：程宪庆　　校核：周洲

3.8.1 基本原理

设图有 n 个顶点，边号分别为 1，2，...，n。，对于其中三个顶点 k，i，j，若 p 是从 i 到 j 的一条最短路径，且满足路径中所有节点的编号都小于等于 k。

如果 k 节点在路径 p 中没有出现，那么 p 的所有中间节点的编号便都在 1～k-1 范围内。

如果 k 在路径 p 中出现过，则该路径包含 i~k 与 k~j 两部分，两部分路径的节点除端点外均不大于 k-1。

所以得到 i~j 的途经顶点不大于 k 的最短路径的一个选择方法，假设使用 g[k][i][j]来表示 i~j 途经顶点不超过 k 的最短距离，则 g[k][i][j]=min(g[k-1][i][j],g[k-1][i][k]+g[k-1][k][j])。在决定途经顶点不超过 k 时每对顶点间的最短距离时只需要途经顶点不超过 k-1 时每对顶点间的最短距离，所以实际上不需要三维数组，只需要将上式改为：

g[i][j]=min(g[i][j],g[i][k]+g[k][j])即可。

将上式改为 if(g[i][k]&&g[k][j])g[i][j]=1;可以用来解决有向图的传递闭包问题，这是一类用来寻找两个元素之间是否存在某种可传递性的关系的问题，比如两个人之间是否有亲属关系，如果一个人 A 和另一个人 B 存在关系，而 B 和 C 也存在关系，那么就可以确定 A 和 C 也存在亲属关系。

3.8.2 模板代码

```
int i,j,k;
for(k=0;k<n;k++)
{
   for(i=0;i<n;i++)
   {
      for(j=0;j<n;j++)
      {
         g[i][j]=min(g[i][j],g[i][k]+g[k][j]);//g 为表示图的邻接矩阵
      }
   }
}
```

3.8.3 经典题目

3.8.3.1 题目 1

1．题目出处/来源

acm.hnu.cn/online 12519

2．题目描述

TOM 想要依次去 N 个城市旅行，他列好了去这 N 个城市的先后顺序，给出每两个城市间的花费，TOM 想知道，按他的旅行顺序去依次参观这 N 个城市最少需要多少钱，每个城市可以重复去。

3．分析

所求为按照给定顺序依次到达每个城市需要最小花费，因为每个城市可以重复到达，根据最小+最小=最小的原理，所以给定顺序中的每两个相邻城市间的最小花费相加即可得到按此顺序行遍 N 个城市所需最小花费，因此可以使用 floyd 求出每两个城市间互相到达的最小花费，即可相应得到所需解。

4．代码

```
#include<iostream>
#include<cstdio>
```

```
#include<cstring>
using namespace std;

#define INF 0x1f1f1f1f

int n;

int g[205][205];//每两个城市间的花费

int a[205];//访问城市的顺序

void floyd()
{
    int i,j,k;
    for (k=0;k<n;k++)
        for (i=0;i<n;i++)
            for (j=0;j<n;j++)
                if (g[i][k]+g[k][j]<g[i][j])
                    g[i][j]=g[i][k]+g[k][j];
}

int main()
{
    int t;
    scanf("%d",&t);
    while(t--)
    {
        memset(g,0,sizeof(g));
        scanf("%d",&n);
        for(int i=0;i<n;i++)
        {
            scanf("%d",&a[i]);
        }
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                scanf("%d",&g[i][j]);
                if(g[i][j]==-1)//如果是-1 表示两个城市不可直接到达，距离设为无穷大
                {
                    g[i][j]=INF;
                }
            }
        }

        floyd();

        int res=0;//求得 res 即是所需答案

        for(int i=0;i<n-1;i++)
        {
            if(g[a[i]][a[i+1]]>=INF)
            {
                res=INF;
                break;
            }
            res+=g[a[i]][a[i+1]];
        }
        res+=g[a[n-1]][a[0]];
        if(res>=INF)
        {
            printf("impossible\n");
        }
        else
        {
            printf("%d\n",res);
        }
    }
    return 0;
```

```
}
```

3.8.3.2 题目 2

1．题目出处/来源

hrbust 1348

2．题目描述

给出一个有向带权图 G，针对该图有如下的两种操作：

（1）标记该图的一个点

（2）找到两点间的只通过已标记点的最短路径长度

输入：

输入包括多组测试，每组测试中第一行包括三个整数 N，M，Q，N 表示图中的节点数量，N<=300，

M 表示边的数量，M<=100000;Q 表示执行多少次操作，Q<=100000，所有点被编号为 0,1,2,...,N-1，

最初所有的点都是未标记的，接下来 M 行每行包括三个整数 x,y,c,表示从 x 到 y 有一条边长度为 c,c>0，然

后为 Q 行，每行表示一次操作，0 x 表示将点 x 标记，1 x y 表示查找 x 到 y 的只通过已标记点的最短路径长度，

N=M=Q=0 是输入结束。

输出：

输出以一行"Case #:"开始，#表示为第几组测试，从 1 开始

对于 0 x 操作，如果 x 已经被标记过了，输出"ERROR! At point x".

对于 1 x y 操作，如果 x 或 y 没有被标记过，输出"ERROR! At path x to y"，如果无法从 x 通过标记过的节

点到达 y,输出"No such path"，否则输出要求的最短路径长度，每组测试后有一个空行。

3．分析

题目的难点在于求解两个点的最短路，因为每次要求两个点间只通过已标记点的最短路而且每两个点间的最短路都有可能求到，所以可以考虑使用 floyd 算法求出所有点间的最短路，而起初被标记的点的数量是 0，所以每两个点间的最短路都应为无穷大，每次标记一个点就对每两个点间的距离进行一次松弛操作。

4．代码

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;

int g[305][305];
bool fl[305];//一个顶点是否被标记

int n,m,q;

int main()
{
 int cse=1;//第几组数据
 while(scanf("%d%d%d",&n,&m,&q))
 {
     if(n==0&&m==0&&q==0)break;

     memset(g,0x1f,sizeof(g));
```

```
        memset(fl,0,sizeof(fl));

        printf("Case %d:\n",cse++);
        for(int i=0;i<n;i++)//自己到自己的距离设为 0
        {
            g[i][i]=0;
        }
        for(int i=0;i<m;i++)
        {
            int a,b,w;
            scanf("%d%d%d",&a,&b,&w);
            g[a][b]=(w<g[a][b]?w:g[a][b]);//防止重边
        }
        for(int i=0;i<q;i++)
        {
            int c,a,b;
            scanf("%d",&c);
            if(c)
            {
                scanf("%d%d",&a,&b);
                if(fl[a]==0||fl[b]==0)
                {
                    printf("ERROR! At path %d to %d\n",a,b);
                }
                else if(g[a][b]>=0x1f1f1f1f)
                {
                    printf("No such path\n");
                }
                else
                {
                    printf("%d\n",g[a][b]);
                }
            }
            else
            {
                scanf("%d",&a);
                if(fl[a])
                {
                    printf("ERROR! At point %d\n",a);
                }
                else
                {
                    fl[a]=1;
                    //下面两层循环为松弛操作
                    for(int j=0;j<n;j++)
                    {
                        for(int k=0;k<n;k++)
                        {
                            if(g[j][a]+g[a][k]<g[j][k])
                            {
                                g[j][k]=g[j][a]+g[a][k];
                            }
                        }
                    }
                }
            }
        }
        printf("\n");
    }
    return 0;
}
```

3.9 差分约束与最短路

编写：程宪庆　　校核：周洲

3.9.1 基本原理

差分约束系统描述的是一个类似于下列关系的不等式组

```
x1-x2<=0
x1-x5<=-1
x2-x5<=1
x3-x1<=5
x4-x1<=4
x4-x3<=-1
x5-x3<=-3
x5-x4<=-3
```

考虑最短路问题中的松弛部分，即对于某条边(u,v)，if(res[v]>res[u]+(u,v))res[v]=res[u]+(u,v),可以发现如下不等式：

dis[v]<=dis[u]+(u,v)，dis 为顶点实际距离源点的最短距离。

于是可以根据不等式来构造一个有向图，以 x[i]作为源点到 i 点的最短距离，x[i]-x[j]<=k 中的 k 作为边(j,i)的权值，使用最短路问题的算法来解决差分约束问题，由于差分约束方程构造出的图可能有负边，所以比较适合使用 Bellman-Ford 或者 SPFA 算法来求得差分约束系统的一组解，如果出现负环，则说明原方程组没有可行解。

3.9.2 解题思路

关键为差分约束方程的寻找，较常见的错误为方程寻找不全。

3.9.3 经典题目

1．题目出处/来源

ZOJ 2770

2．题目描述

题目描述的是刘备攻打陆逊时的连营，输入为大营个数 n，每个大营中的最多士兵数 Ci，以及若干个三元组 i,j,k 表示第 i 个大营到第 j 个大营的士兵总数最少有 k 个。求刘备最少有多少士兵。

3．分析

根据题意，可以找出以下几个不等关系：

(1).第 i 个到第 j 个大营最小有 k 个士兵，即 s[j]-s[i-1]>=k,s[i-1]-s[j]<=-k,即可知在对应的图中有边(j,i-1)的权值为-k。

(2).根据每个大营中最多有 c[i]个士兵以及最少有 0 个士兵，可得 s[i]-s[i-1]>=0 及 s[i]-s[i-1]<=c[i]两个不等式，根据这三类不等式，建立有向图，利用 Bellman-Ford 或者 SPFA 求得顶点 n 到顶点 0 的最短路，即可得全部最少有多少士兵。

4．代码

```
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;

#define EDGE_COUNT 30000
#define PNT_COUNT 2000
#define INF 0x1f1f1f1f

struct edge
{
 int adj;
 int point;
 int weight;
 edge(int a,int p,int w):adj(a),point(p),weight(w){}
 edge():adj(0),point(0),weight(0){}
```

```
};

edge e[EDGE_COUNT];
int dis[PNT_COUNT];

int c[1005];

bool bellman_ford(int src,int edge_count,int pnt_count)
{
 memset(dis,0x1f,sizeof(dis));
 dis[src]=0;
 for(int i=1;i<pnt_count;i++)
 {
     for(int j=1;j<=edge_count;j++)
     {
         if(dis[e[j].adj]<INF&&dis[e[j].adj]+e[j].weight<dis[e[j].point])
         {
             dis[e[j].point]=dis[e[j].adj]+e[j].weight;
         }
     }
 }
 for(int i=1;i<=edge_count;i++)
 {
     if(dis[e[i].adj]<INF&&dis[e[i].adj]+e[i].weight<dis[e[i].point])
     {
         return 0;
     }
 }
 return 1;
}

int main()
{
 int n,m;
 while(~scanf("%d%d",&n,&m))
 {
     for(int i=1;i<=n;i++)
     {
         scanf("%d",&c[i]);
     }
     for(int i=1;i<=m;i++)
     {
         int u,v,w;
         scanf("%d%d%d",&u,&v,&w);
         e[i]=edge(v,u-1,-w);//u--v 兵营中最多有 w 个士兵，以 i 为边的编号
     }
     for(int i=1;i<=n;i++)
     {
         e[m+i]=edge(i-1,i,c[i]);//第 i 个兵营中最多有 c[i]个士兵 m+i 为这些边的边号
         e[m+n+i]=edge(i,i-1,0);//并且最少有 0 士兵，m+n+i 为这些边的边号
     }
     if(bellman_ford(n,m+n+n,n+1))printf("%d\n",-dis[0]);//注意负号
     else printf("Bad Estimations\n");
 }
 return 0;
}
```

3.10 最大流

最大流问题研究的是在一个每条边被赋予一个容量的图中，由源点到汇点最多可以稳定的保持多大的总流量。类似于在一个输水管道网络中或在一个电路网络中，从一个点到另一个点可以输送的最大水流或者电流。还有很多看似无关的问题也可以抽象为最大流问题然后根据流守恒等性质来加以解决。

编写：程宪庆　　校核：周洲

3.10.1 基本原理

最大流问题的解决一般基于两种方法，即**增广路算法**与**预流推进算法**。

在一个图中，***残留网络***指在既有的容量和已具备的流量条件下，网络中仍然可以继续增大流量的边所组成的网络，在该网络中的一条从源点流向汇点的路径叫做一条***增广路***，增广路算法利用不断寻找增广路并在其上对流量进行更新的方法寻找网络的最大流。

图的**割**可以用来表示对图的一个划分，将原图 G=(V,E)的顶点集 V 分为 S、T 两部分，让源点 s 在 S 中，汇点 t 在 T 中，能够通过 S、T 间的最大净流量为割(S,T)的容量，***最小割***为图中具有最小容量的割。

最大流最小割定理：在网络的一个流量状态下，通过图的任意一个割的流量都与该流量相同，所以具有最小容量的割的的容量就是该图的流量的最大值即最大流。

对残留网络进行广度优先搜索寻找增广路并对找到的增广路上所经过的边的流量进行更新，找到该路径上可增加流量最小的一条边(i,j), 对于该增广路上的每条边，将正向边的流量增加 cap(i,j)-flow(i,j), 反向边的流量减少相同值，反复执行直到找不到增广路为止，这样就得到了计算最大流的基础的 ***EK 算法***：

1. 初始化容量 c 和流量 f;
2. 对图中满足 c(i,j)>f(i,j)的边进行广度优先搜索，并记录广搜过程中最小的 c(i,j)-f(i,j)，设其值为 m;
3. 若在 1 中没有找到汇点，则算法结束，若找到汇点，则从汇点开始沿着在广搜中找到的源点到汇点的路径反向回到源点，并对路径上的每条边(i,j)执行 f(i,j)+=m,f(j,i)-=m;

另一种增广路算法 Dinic 算法按广度优先搜索对原图进行分层操作，计算出每个顶点与源点之间的最少边数，每次只在边数相差 1 的两个顶点间进行增广，使用深度优先搜索每次进行多次增广，直到不能增广再重新进行广搜分层，如果广搜找不到汇点，则算法结束，因为使用了分层和多次增广，Dinic 的速度要比 EK 快很多。

1. 初始化容量 c 和流量 f;
2. 对满足 c(i,j)>f(i,j)的边进行广度优先搜索，标记每个顶点 i 到源点的无权最短距离 d[i]，若广搜没有到达汇点，则算法结束
3. 在满足 d[i]==d[j]-1 的边中进行深度优先搜索，每次深搜更新流量，深搜结束回到 2.

ISAP 算法与 Dinic 算法不同，对原图进行与汇点最短距离的标记，并且在增广过程中直接对其进行更新，所以避开了 Dinic 中重新标记距离的过程，每次按标记递减 1 的规则在残留网络中寻找可增广边，找到汇点则进行一次增广，如果在某点处找不到标记递减 1 并且有可行流的边，则对其距离标号进行更新，改为与其相邻的点中最小标号+1。

1. 初始化流网络;
2. 进行一次反向广度优先搜索，找到每个顶点到汇点的最短距离;
3. 沿着满足 d[i]==d[j]+1 并且 c(i,j)>f(i,j)的边向前寻找
4. 若在 3 中找到满足条件的点：若该点是汇点，则进行一次增广，若该点不是汇点，重复 3，若没有找到满足条件的点，在满足 c(i,j)>f(i,j)的边中寻找 d 值最小的，并将该点的 d 值设为这个最小值加 1，回到该点的上一个点，继续重复 3;

在计算过程一般中使用一个辅助数组 num[]记录每个距离标号的顶点数，比如距离汇点为 4 的点有 5 个，则 num[4]=5,每次更新距离时对 num 数组同时更新，如果某个 num 值为 0，则计算过程结束，这个优化被叫做 Gap 优化。

3.10.2 解题思路

求解网络流问题的关键是模型图的建立，寻找到流量与相应问题间的对应数量关系。

3.10.3 模板代码

EK 算法：

```
int ek(int st,int ed,int src,int tar)//st ed 节点编号范围，src tar 源点 汇点
{
    int res=0;
    int pre[N];
    int mn[N];
    memset(fl,0,sizeof(fl));
    while(1)
    {
        memset(pre,-1,sizeof(pre));
        mn[src]=INF;
        memset(fl,0,sizeof(fl));
        queue<int>q;
        q.push(src);
        while(!q.empty())
        {
            int t=q.front();
            q.pop();
            if(fl[t])continue;
            fl[t]=1;
            for(int i=st;i<=ed;i++)
            {
                if(!fl[i]&&cap[t][i]-flow[t][i]>0)
                {
                    pre[i]=t;
                    mn[i]=MIN(cap[t][i]-flow[t][i],mn[t]);
                    q.push(i);
                }
            }
            if(pre[tar]!=-1)
            {
                for(int i=tar;pre[i]!=-1;i=pre[i])
                {
                    flow[pre[i]][i]+=mn[tar];
                    flow[i][pre[i]]-=mn[tar];
                }
                break;
            }
        }
        if(pre[tar]==-1)break;
        else res+=mn[tar];
    }
    return res;
}
```

Dinic 算法：

```
#define INF 0x1f1f1f1f
#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAX(a,b) ((a)>(b)?(a):(b))

#define N 500

int cap[N][N];//容量
int flow[N][N];//流量

int lev[N];//层次
bool vis[N];//标记

int que[100000];//队列

//BSF 找层次网络，一次寻找多条增广路径
```

```
//st 最小顶点标号，ed 最大顶点标号，src 源点标号，tar 汇点标号
bool bfs(int st,int ed,int src,int tar)//st 最小点 ed 最大点 src 源点 tar 汇点
{
    int front;//队首
    int rear;//队尾
    front=rear=0;

    que[front++]=src;
    lev[src]=0;
    memset(vis,0,sizeof(vis));
    vis[src]=1;

    while(rear<front)
    {
        int t=que[rear];
        rear++;

        for(int i=st;i<=ed;i++)
        {
            if(!vis[i]&&cap[t][i]>flow[t][i])
            {
                vis[i]=1;
                lev[i]=lev[t]+1;
                que[front++]=i;
            }
        }
    }
    return lev[tar]<INF;
}

//利用层次网络进行增广，每次 DFS 寻找的是从该节点出发进行 DFS 增加的总流量
//mn 表示从源点至该节点可增广流量
int dfs(int v,int st,int ed,int tar,int fl)//fl 表示源点到当前顶点的流量
{
    int ret=0;
    if(v==tar||fl==0)return fl;
    for(int i=st;i<=ed;i++)
    {
        if(fl==0)break;
        if(cap[v][i]>flow[v][i]&&lev[v]+1==lev[i])
        {
            int f=MIN(fl,cap[v][i]-flow[v][i]);//沿 i 点向下可用最大流量
            int tt=dfs(i,st,ed,tar,f);//沿 i 点向下实际增广的流量
            if(tt<=0)continue;
            ret+=tt;
            fl-=tt;//每次修改 fl
            flow[v][i]+=tt;
            flow[i][v]-=tt;
        }
    }
    return ret;
}

int dinic(int st,int ed,int src,int tar)
{
    int ret=0;
    while(bfs(st,ed,src,tar))//存在可增广路
    {
        int r=dfs(src,st,ed,tar,INF);
        if(r==0)break;
        ret+=r;
    }
    return ret;
}
```

ISAP 算法：

```
/* 最大流 SAP 邻接表
```

```
 * GAP + 当前弧优化
 * GAP: 间隙优化
 * 当前弧：是的每次寻找增广路时间变为 O(V)
 * */

const int INF = INT_MAX / 3;
const int MAXN = 20000 + 5;
const int MAXM = 200000 + 5;

struct Edge {
 int u, v;
 int c;
 int next;
 Edge() {}
 Edge(int tu, int tv, int tc, int tn) : u(tu), v(tv), c(tc), next(tn) {}
};

Edge E[MAXM * 3];
// head[]顶点弧表表头
int nE, head[MAXN], cnt[MAXN], que[MAXN], d[MAXN], low[MAXN], cur[MAXN];

void addEdge(int u, int v, int c, int rc = 0) { // c 正向弧容量，rc 反向弧容量
 E[nE]= Edge(u, v, c, head[u]);
 head[u] = nE++;
 E[nE]= Edge(v, u, rc, head[v]);
 head[v] = nE++;
}

void initNetwork(int n = MAXN) { // head[] 数组初始化为-1
 memset(head, -1, sizeof(head[0])*n);
 nE = 0;
}

int maxflow(int n, int source, int sink) {
 int *fr = que, *ta = que;;
 for (int i = 0; i < n; ++i) d[i] = n, cnt[i] = 0;
 cnt[n] = n - 1, cnt[0]++, d[sink] = 0;
 *ta++ = sink;
 while (fr < ta) {
     int v = *fr++;
     for (int i = head[v]; i != -1; i = E[i].next) {
         if (d[E[i].v] == n && E[i^1].c > 0) {
             d[E[i].v] = d[v] + 1;
             cnt[n]--;
             cnt[d[E[i].v]]++;
             *ta++ = E[i].v;
         }
     }
 }
 int flow = 0, u = source, top = 0;
 low[0] = INF;
 for (int i = 0; i < n; ++i) cur[i] = head[i];
 while (d[source] < n) { // que 类似于 pre 数组，存的是边
     int &i = cur[u];
     for (; i != -1; i = E[i].next) {
         if (E[i].c > 0 && d[u] == d[E[i].v] + 1) {
             low[top+1] = low[top] < E[i].c ? low[top] : E[i].c;
             que[top+1] = i;
             ++top;
             u = E[i].v;
             break;
         }
     }
     if (i != -1) {
         if (u == sink) {
             int delta = low[top];
             for (int p = 1, i; p <= top; ++p) {
                 i = que[p];
```

```
                E[i].c -= delta;
                E[i^1].c += delta;
            }
            flow += delta;
            u = source;
            low[0] = INF;
            top = 0;
        }
    } else {
        int old_du = d[u];
        cnt[old_du]--;
        d[u] = n - 1;
        for (int i = head[u]; i != -1; i = E[i].next) {
            if (E[i].c > 0 && d[u] > d[E[i].v]) d[u] = d[E[i].v];
        }
        cnt[++d[u]]++;
        if (d[u] < n) cur[u] = head[u];
        if (u != source) {
            u = E[que[top]].u;
            --top;
        }
        if (cnt[old_du] == 0) break;
    }
}
return flow;
}
```

3.10.4 经典题目

3.10.4.1 题目 1

POJ 3084

2．题目描述

有一些建在一起的房间，互相之间通过门相连，一个门的控制开关在它连接的两个房间中的一个里，在有开关的房间可以任意进入没开关的另一侧房间，而在另一侧的房间中要进入有开关的房间中则需要门是开着的，现在有的房间中有入侵者，同时有一个非常重要的房间需要保护，问要保护那个房间不被入侵，最少要关上几道门。

3．分析

对于在开关侧房间的入侵者，无论关多少个门，都无法阻止他进入开关连接的另一侧房间，对于在开关另一侧房间的入侵者，只要关上这个开关，就可以阻止他进入有开关的一侧房间，所以可以以房间为顶点，门为边，将两种情况分别连无穷大的容量与 1 的容量，另外设一个单独的源点，将其到每个有入侵者的房间连无穷大的边，求出从源点到保护房间的最大流，即可知道最少要关几个门。

4．代码

```
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;

int m,n;

#define INF 0x1f1f1f1f
#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAX(a,b) ((a)>(b)?(a):(b))

#define SIZE 30

int flow[SIZE][SIZE];
int cap[SIZE][SIZE];
int lev[SIZE];
int mn[SIZE];

int que[100000];
```

```
//BSF 找层次网络，一次寻找多条增广路径
//st 最小顶点标号，ed 最大顶点标号，src 源点标号，tar 汇点标号
bool bfs(int st,int ed,int src,int tar)
{
    memset(lev,0x1f,sizeof(lev));

    int front;
    int rear;
    front=rear=0;

    que[front++]=src;
    lev[src]=0;

    while(rear<front)
    {
        int t=que[rear];
        rear++;

        for(int i=st;i<=ed;i++)
        {
            if(cap[t][i]>flow[t][i])
            {
                if(lev[t]+1<lev[i])
                {
                    lev[i]=lev[t]+1;
                    que[front++]=i;
                }
            }
        }
    }
    return lev[tar]<INF;
}

//利用层次网络进行增广，每次 DFS 寻找的是从该节点出发进行 DFS 增加的总流量
//mn 表示从源点至该节点可增广流量
int dfs(int v,int st,int ed,int src,int tar)
{
    int ret=0;
    if(v==tar)return mn[tar];
    for(int i=st;i<=ed;i++)
    {
        if(mn[v]==0)break;
        if(cap[v][i]>flow[v][i]&&lev[v]+1==lev[i])
        {
            mn[i]=MIN(mn[v],cap[v][i]-flow[v][i]);
            int tt=dfs(i,st,ed,src,tar);
            ret+=tt;
            mn[v]-=tt;//每次修改 mn[v]
            flow[v][i]+=tt;
            flow[i][v]-=tt;
        }
    }
    if(ret==0)
    {
        lev[v]=INF;
    }
    return ret;
}

int dinic(int st,int ed,int src,int tar)
{
    int ret=0;
    while(bfs(st,ed,src,tar))//存在可增广路
    {
        memset(mn,0x1f,sizeof(mn));
        int r=dfs(src,st,ed,src,tar);
        if(r==0)break;
        ret+=r;
    }
```

```
    return ret;
}

int main()
{
    //printf("%d\n",INF);
    int t;
    scanf("%d",&t);
    while(t--)
    {
        memset(flow,0,sizeof(flow));
        memset(cap,0,sizeof(cap));
        scanf("%d%d",&m,&n);
        for(int i=0;i<m;i++)
        {
            char ch[5];
            int c;
            scanf("%s%d",ch,&c);
            if(strcmp(ch,"I")==0)
            {
                cap[m][i]=INF;
            }
            for(int j=0;j<c;j++)
            {
                int num;
                scanf("%d",&num);
                cap[i][num]=INF;
                if(cap[num][i]<INF)
                {
                    cap[num][i]++;
                }
            }
        }
        int res=dinic(0,m,m,n);
        if(res<INF)printf("%d\n",res);
        else printf("PANIC ROOM BREACH\n");
    }
    return 0;
}
```

3.10.4.2 题目 2

1．题目出处/来源

ZOJ 2760

2．题目描述

给定一个带权有向图 G=(V, E)和源点 s、汇点 t，问 s-t 边不相交最短路最多有几条。（1 <= N <= 100）

3．分析

使用最短路算法找到 s 到所有点的最短路和所有点到 t 的最短路，可以使用 floyd 或者 dijkstra 算法，然后保留所有 dis[s][i]+dis[j][t]+g[i][j]==dis[s][t]并且 dis[s][i],dis[j][t],g[i][j]都不为无穷大的边，将其容量设为 1，其他边容量设为 0，求出从 s 到 t 的最大流即得解。

要注意的地方是题目输入数据中对角线处有可能不给 0，要自己处理成 0.

4．代码

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<queue>
using namespace std;

#define INF 0x1f1f1f1f
#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAX(a,b) ((a)>(b)?(a):(b))

#define N 500
```

```
int g[N][N];
bool fl[N];
int r[N][N];

int cap[N][N];
int flow[N][N];

int lev[N];
bool vis[N];

int que[100000];

//BSF 找层次网络，一次寻找多条增广路径
//st 最小顶点标号，ed 最大顶点标号，src 源点标号，tar 汇点标号
bool bfs(int st,int ed,int src,int tar)
{
    int front;
    int rear;
    front=rear=0;

    que[front++]=src;
    lev[src]=0;
    memset(vis,0,sizeof(vis));
    vis[src]=1;

    while(rear<front)
    {
        int t=que[rear];
        rear++;

        for(int i=st;i<=ed;i++)
        {
            if(!vis[i]&&cap[t][i]>flow[t][i])
            {
                vis[i]=1;
                lev[i]=lev[t]+1;
                que[front++]=i;
            }
        }
    }
    return lev[tar]<INF;
}

//利用层次网络进行增广，每次 DFS 寻找的是从该节点出发进行 DFS 增加的总流量
//mn 表示从源点至该节点可增广流量
int dfs(int v,int st,int ed,int tar,int fl)
{
    int ret=0;
    if(v==tar||fl==0)return fl;
    for(int i=st;i<=ed;i++)
    {
        if(fl==0)break;
        if(cap[v][i]>flow[v][i]&&lev[v]+1==lev[i])
        {
            int f=MIN(fl,cap[v][i]-flow[v][i]);
            int tt=dfs(i,st,ed,tar,f);
            if(tt<=0)continue;
            ret+=tt;
            fl-=tt;//每次修改 fl
            flow[v][i]+=tt;
            flow[i][v]-=tt;
        }
    }
    return ret;
}

int dinic(int st,int ed,int src,int tar)
{
    int ret=0;
```

```
    while(bfs(st,ed,src,tar))//存在可增广路
    {
        int r=dfs(src,st,ed,tar,INF);
        if(r==0)break;
        ret+=r;
    }
    return ret;
}

int main()
{
    int n;
    while(~scanf("%d",&n))
    {
        for(int i=1;i<=n;i++)
        {
            for(int j=1;j<=n;j++)
            {
                scanf("%d",&g[i][j]);
                if(g[i][j]==-1)g[i][j]=INF;
                if(i==j)g[i][j]=0;
                r[i][j]=g[i][j];
            }
        }

        int src,tar;
        scanf("%d%d",&src,&tar);
        src++,tar++;
        if(src==tar)
        {
            printf("inf\n");
            continue;
        }
        for(int k=1;k<=n;k++)
        {
            for(int i=1;i<=n;i++)
            {
                for(int j=1;j<=n;j++)
                {
                    if(r[i][k]+r[k][j]<r[i][j])
                    {
                        r[i][j]=r[i][k]+r[k][j];
                    }
                }
            }
        }


        for(int i=1;i<=n;i++)
        {
            for(int j=1;j<=n;j++)
            {
                if(r[src][i]+r[j][tar]+g[i][j]==r[src][tar]&&g[i][j]<INF)
                {
                    cap[i][j]=1;
                }
                else
                {
                    cap[i][j]=0;
                }
            }
        }
        memset(flow,0,sizeof(flow));
        int res=dinic(1,n,src,tar);
        printf("%d\n",res);
    }
    return 0;
}
```

3.11 最小费用最大流

编写：程宪庆　　校核：周洲

3.11.1 基本原理

最小费用最大流问题是在普通的最大流问题上加了另一个条件，即要每条边单位流量所需的费用，要在求得最大流的情况下所需费用最小。

求最小费用可以转化成求解一个最短路问题，而求最大流需要广度搜索，所以可以使用 SPFA 最短路算法和 EK 最大流算法结合的方法求解最小费用最大流问题。

3.11.2 模板代码

```
#define INF 0x3f3f3f3f
#define MIN(a,b) ((a)<(b)?(a):(b))

int mat[55][55];
int n,k;

int head[5005];
struct Arc
{
    int next_arc;
    int point;
    int adj;
    int cost;
    int cap;
};
struct Arc arc[25000];

int pre[5005];
int dis[5005];
bool fl[5005];

int max_flow;
int min_cost;

int edge_cnt;

void add(int u,int v,int cst,int cp)
{
    arc[edge_cnt].next_arc=head[u];
    arc[edge_cnt].point=v;
    arc[edge_cnt].adj=u;
    arc[edge_cnt].cost=cst;
    arc[edge_cnt].cap=cp;
    head[u]=edge_cnt;
}

void cost_flow(int src,int tar)
{
    while(1)
    {
        memset(pre,-1,sizeof(pre));
        memset(dis,0x3f,sizeof(dis));
        memset(fl,0,sizeof(fl));

        queue<int>q;
        q.push(src);
        dis[src]=0;

        while(!q.empty())
        {
            int u=q.front();
            q.pop();
```

```
            fl[u]=0;

            for(int e=head[u];e!=-1;e=arc[e].next_arc)
            {
                if(arc[e].cap>0&&dis[u]+arc[e].cost<dis[arc[e].point])
                {
                    dis[arc[e].point]=dis[u]+arc[e].cost;
                    pre[arc[e].point]=e;
                    if(!fl[arc[e].point])
                    {
                        fl[arc[e].point]=1;
                        q.push(arc[e].point);
                    }
                }
            }
        }

        if(pre[tar]==-1)break;

        int min=INF;

        for(int i=tar;pre[i]!=-1;i=arc[pre[i]].adj)
        {
            min=MIN(min,arc[pre[i]].cap);
        }
        for(int i=tar;pre[i]!=-1;i=arc[pre[i]].adj)
        {
            arc[pre[i]].cap-=min;
            arc[pre[i]^1].cap+=min;
        }

        max_flow+=min;
        min_cost+=min*dis[tar];
    }
}
```

3.11.3 经典题目

1．题目出处/来源

POJ 3422

2．题目描述

有一个 N*N 的矩阵，每个位置有一个非负整数，从左上角到右下角走，经过每个位置的时候就将方格中的整数加到 sum 中，sum 初始为 0，问走 K 次之后能够获得的 sum 的最大值是多少。

3．分析

将每个位置作为两个顶点，两个顶点间容量为 1 费用为该位置的整数，两个位置若是相连的，则从一个位置的两个顶点分别到另一个位置的两个顶点建立四条边，容量为无穷，费用为 0，添加附加源点和汇点，源点与 1 点，汇点与 n*n*2 点连接容量为 K 费用为 0 的边，求出最大费用即可。

4．代码

以下只给出了主函数部分，最小费用流直接调用上面模板即可，因为要求最大费用，所以将费用设为了负数，求完最小费用再取相反数即为最大费用。

```
int main()
{
    while(~scanf("%d%d",&n,&k))
    {
        max_flow=0;
        min_cost=0;

        memset(head,-1,sizeof(head));

        for(int i=1;i<=n;i++)
        {
```

```
            for(int j=1;j<=n;j++)
            {
                scanf("%d",&mat[i][j]);
            }
        }

        edge_cnt=0;

        for(int i=1;i<=n;i++)
        {
            for(int j=1;j<=n;j++)
            {
                int p3=(i-1)*n+j;
                int p4=(i-1)*n+j+n*n;

                add(p3,p4,-mat[i][j],1);
                edge_cnt++;

                add(p4,p3,mat[i][j],1);
                edge_cnt++;


                add(p3,p4,0,k);
                edge_cnt++;

                add(p4,p3,0,0);
                edge_cnt++;

                if(i<n)
                {
                    int p1=(i-1)*n+j;
                    int p2=i*n+j;

                    add(p1+n*n,p2,0,k);
                    edge_cnt++;

                    add(p2,p1+n*n,0,0);
                    edge_cnt++;

                }
                if(j<n)
                {
                    int p1=(i-1)*n+j;
                    int p2=(i-1)*n+j+1;

                    add(p1+n*n,p2,0,k);
                    edge_cnt++;
                    add(p2,p1+n*n,0,0);
                    edge_cnt++;
                }
            }
        }

        add(0,1,0,k);
        edge_cnt++;

        add(1,0,0,0);
        edge_cnt++;

        edge_cnt++;
        add(n*n*2,n*n*2+1,0,k);
        edge_cnt++;
        add(n*n*2+1,n*n*2,0,0);

        cost_flow(0,2*n*n+1);

        printf("%d\n",-min_cost);
    }
    return 0;
```

```
}
```

3.12 有上下界的最大流

参考文献：

王桂平.《图论算法理论、实现及应用》 北京大学出版社 2011 年

编写：程宪庆　　校核：周洲

3.12.1 基本原理

上下界最大流对普通的最大流问题附加了一个流量上界和下界，求在每条边的流量不超过此上、下界的情况下可得的最大或最小流量，普通的最大流问题可以看成是下界为 0，上界是容量的最大流问题。

设原网络各边容量上界为 c,下界容量为 b，源点为 s,汇点为 t.

要求解有上下界的最大或最小流，首先要确定是否可以得到一个可行流，实现该目的需要构造一个伴随网络：

1. 增加两个顶点 Vs,Tt,分别为附加源点和附加汇点
2. 对原图中每个顶点 vi,增加弧(vi,Vt),容量为 vi 发出的所有弧的流量下界和
3. 对原图中每个顶点 vi,增加弧(Vs,vi),容量为进入 vi 的所有弧的流量下界和
4. 对原图中的每条弧(u,v)，将其容量修改为 c(u,v)-b(u,v)。
5. 增加弧(s,t),(t,s)，容量上界为 INF，下界为 0.
6. 求 Vs 到 Vt 的最大流

通过在伴随网络中求最大流的过程，可以得到为了满足上下界条件，并且在所有边在除去下界的情况下最少需要的流量，因为所有弧的流量下界都由附加源点和附加汇点承担了，如果在该最大流中，所有从 Vs 发出的弧都满载，则原网络存在满足上下界条件的流，否则不存在可行流。

要求解满足流量上下界的流中的最大流，在利用上述方法得到可行流之后的基础上运行最大流算法，并在最大流算法中注意保证每条弧的流量在减少之后不能少于流量下界，即减少的流量不能多于该弧的流量与其下界之差，如此即可得到满足条件的最大流。

反之在求满足流量上下界的最小流时，首先找到可行流，然后在可行流基础上，在原图中以汇点为源点，源点为汇点，即源汇点交换再将可行流放大，如果汇点到源点有路径并且该路径上的弧在可行流中都并没有达到上界，那么该过程就会将汇点到源点的流放大，相应的源点到汇点的流应会减少，将该流扩大为最大流，此时源点到汇点的流量即为最小流。

3.12.2 经典题目

1．题目出处/来源

POJ 2396

2．题目描述

要针对一个多赛区竞赛制定预算，该预算是一个行代表不同各类支出、列代表不同赛区支出的矩阵，要制定一个满足所有约束条件且行列和满足要求的预算。输入矩阵行、列数和每行、每列的和，以及若干个对某些元素的约束条件。

3．分析

以各赛区及各类型支出分别为顶点建图，一个赛区到一个类型的弧表示该赛区在该类型上的花费，另设源点汇点，源点连所有赛区，流量上下界均为该赛区费用和，汇点连所有类型，流量上下界为该类型费用和，对每个约束条件中的 r,q 按相应的约束条件修改相应弧的流量上下办，注意重复输入，上界应取最小的，下界应取最大的，因题目 special judge,求出可行流即可。

4. 代码

```
int main()
{
    int t;
    scanf("%d",&t);
    bool first=1;
    while(t--)
    {
        memset(cap,0,sizeof(cap));
        memset(g1,0,sizeof(g1));
        memset(g2,0x1f,sizeof(g2));
        memset(flow,0,sizeof(flow));
        scanf("%d%d",&m,&n);
        for(int i=0;i<m;i++)//以 1 为源点，此处为输入每行的和，第 i 行对应顶点 i
        {
            int nu;
            scanf("%d",&nu);

            g1[1][i+2]=MAX(g1[1][i+2],nu);
            g2[1][i+2]=MIN(g2[1][i+2],nu);
        }
        for(int i=0;i<n;i++)//输入 n 列的和以 m+n+2 为汇点
        {
            int nu;
            scanf("%d",&nu);

            g1[m+i+2][m+n+2]=MAX(g1[m+i+2][m+n+2],nu);
            g2[m+i+2][m+n+2]=MIN(g2[m+i+2][m+n+2],nu);
        }

        cap[1][m+n+2]=cap[m+n+2][1]=INF;

        int c;
        scanf("%d",&c);

        for(int i=0;i<c;i++)
        {
            int r,q,v;
            char ch;
            scanf("%d%d %c%d",&r,&q,&ch,&v);
            if(r!=0&&q!=0)
            {
                if(ch=='=')
                {
                    g1[r+1][q+m+1]=MAX(g1[r+1][q+m+1],v);
                    g2[r+1][q+m+1]=MIN(g2[r+1][q+m+1],v);
                }
                else if(ch=='<')
                {
                    g2[r+1][q+m+1]=MIN(g2[r+1][q+m+1],v-1);
                }
                else
                {
                    g1[r+1][q+m+1]=MAX(g1[r+1][q+m+1],v+1);
                }
            }
            else
            {
                if(r==0&&q==0)
                {
                    if(ch=='=')
                    {
                        for(int i1=0;i1<m;i1++)
                        {
                            for(int j1=0;j1<n;j1++)
                            {
                                g1[i1+2][m+2+j1]=MAX(g1[i1+2][m+2+j1],v);
                                g2[i1+2][m+2+j1]=MIN(g2[i1+2][m+2+j1],v);
                            }
```

```
            }
        }
        else if(ch=='<')
        {
            for(int i1=0;i1<m;i1++)
            {
                for(int j1=0;j1<n;j1++)
                {
                    g2[i1+2][m+2+j1]=MIN(g2[i1+2][m+2+j1],v-1);
                }
            }
        }
        else
        {
            for(int i1=0;i1<m;i1++)
            {
                for(int j1=0;j1<n;j1++)
                {
                    g1[i1+2][m+2+j1]=MAX(g1[i1+2][m+2+j1],v+1);
                }
            }
        }
    }
    else if(r==0)
    {
        if(ch=='=')
        {
            for(int i=0;i<m;i++)
            {
                g1[i+2][m+q+1]=MAX(g1[i+2][m+q+1],v);
                g2[i+2][m+q+1]=MIN(g2[i+2][m+q+1],v);
            }
        }
        else if(ch=='<')
        {
            for(int i=0;i<m;i++)
            {
                g2[i+2][m+q+1]=MIN(g2[i+2][m+q+1],v-1);
            }
        }
        else
        {
            for(int i=0;i<m;i++)
            {
                g1[i+2][m+q+1]=MAX(g1[i+2][m+q+1],v+1);
            }
        }
    }
    else
    {
        if(ch=='=')
        {
            for(int i=0;i<n;i++)
            {
                g1[r+1][m+2+i]=MAX(g1[r+1][m+2+i],v);
                g2[r+1][m+2+i]=MIN(g2[r+1][m+2+i],v);
            }
        }
        else if(ch=='<')
        {
            for(int i=0;i<n;i++)
            {
                g2[r+1][m+2+i]=MIN(g2[r+1][m+2+i],v-1);
            }
        }
        else
        {
            for(int i=0;i<n;i++)
            {
```

```
                        g1[r+1][m+2+i]=MAX(g1[r+1][m+2+i],v+1);
                    }
                }
            }
        }

        for(int i=0;i<m;i++)
        {
            for(int j=0;j<n;j++)
            {
                cap[i+2][j+m+2]=g2[i+2][j+m+2]-g1[i+2][j+m+2];//修改每条边的流量
            }
        }

        for(int i=1;i<=m+n+2;i++)
        {
            int num=0;
            for(int j=1;j<=m+n+2;j++)
            {
                num+=g1[i][j];
            }
            cap[i][m+n+3]=num;//顶点 i 到附加汇点的容量为 i 流出的下界和
            num=0;
            for(int j=1;j<=m+n+2;j++)
            {
                num+=g1[j][i];
            }
            cap[0][i]=num;
        }

        dinic(0,m+n+3,0,m+n+3);//该处直接调用上面的最大流函数即可，以 0 为附加源点，m+n+3 为附加汇点

        if(!first)//判断是否第一组测试数据~
        {
            printf("\n");
        }
        first=0;

        int jud=1;//判断是否得到了可行流
        for(int i=1;i<=m+n+3;i++)
        {
            if(flow[0][i]!=cap[0][i])
            {
                jud=0;
                break;
            }
        }
        if(jud==0)
        {
            printf("IMPOSSIBLE\n");
            continue;
        }

        for(int i=0;i<m;i++)
        {
            for(int j=0;j<n;j++)
            {
                if(j>0)printf(" ");
                printf("%d",flow[i+2][j+m+2]+g1[i+2][j+m+2]);//输出可行流
            }
            printf("\n");
        }
    }
    return 0;
}
```

3.13 树的最小支配集，最小点覆盖与最大独立集

参考文献：

《图论及应用》冯林、金博、于瑞云 哈尔滨工业大学出版社 2012 年 3 月第一版

编写：周洲　　校核：程宪庆

3.13.1 基本原理

定义 1：对于图 G = (V, E) 来说，最小支配集指的是从 V 中取尽可能少的点组成一个集合，使得 V 中剩余的点都与取出来的点有边相连。也就是说，设 V`是图 G 的一个支配集，则对于图中的任意一个顶点 u，要么属于集合 V`，要么与 V`中的顶点相邻。在 V`中除去任何元素后 V`不再是支配集，则支配集 V`是极小支配集。称 G 的所有支配集中顶点个数最少的的支配集为最小支配集，最小支配集中的顶点个数成为支配数。

定义 2：对于图 G = (V, E) 来说，最小点覆盖值得是从 V 中取尽可能少的点组成一个集合，使得 E 中所有的边都与取出来的点相连。也就是说设 V`是图 G 的一个顶点覆盖，则对于图中任意一条边（u, v）,要么 u 属于集合 V`，要么 v 属于集合 V`。在 V`中除去任何元素之后 V`不再是顶点覆盖，则 V`是极小顶点覆盖。称 G 的所有顶点覆盖中顶点个数最少的覆盖为最小点覆盖。

定义 3：对于图 G = （V, E）来说，最大独立集指的是从 V 中取尽量多的点组成一个集合，使得这些点之间没有边相连。也就是说设 V`是图 G 的一个独立集，则对于图中任意一条边（u, v）,u 和 v 不能同时属于集合 V`，甚至可以 u 和 v 都不属于集合 V`。在 V`中添加任何不属于 V`元素后 V`不再是独立集，则 V`是极大独立集。称 G 的所有顶点独立集中顶点个数最多的独立集为最大独立集。

对于任意图 G 来说，最小支配集、最小点覆盖与最大独立集问题不存在多项式时间的解法。不过如果图 G 是一棵树，求这三种特殊的集合倒是十分容易。目前有两种解法，一种基于贪心思想，另一种基于树形动态规划思想，这两种算法都可以解决上面的 3 个问题。

1. 贪心法求树的最小支配集、最小点覆盖和最大独立集

(1) 基本算法

首先介绍贪心法解树的最小支配集、最小点覆盖和最大独立集。以最小支配集为例，对于书树上的最小支配集问题，贪心策略是首先选择一点为根，按照深度优先遍历得到遍历序列，按照所得序列的反向序列的顺序进行贪心，对于一个既不属于支配集又不与支配集中顶点相连的点来说，如果它的父节点不属于支配集，将其父节点加入支配集。

这里注意到贪心的策略中贪心的顺序非常重要，按照深度优先遍历得到遍历序列的反方向进行贪心，可以保证对于每个点来说，当其子树都被处理过后才会轮到该节点的处理，保证了贪心的正确性。

伪代码：

① 以 1 号点深度优先搜索整棵树，求出每个点在深度优先遍历序列中的编号和每个点的父亲点编号。

② 按照深度优先序列的反向序列检查，如果当前点既不属于支配集也不与支配集中点相连，且它的父亲点不属于支配集，将其父亲节点加入支配集，支配集中点的个数加 1。标记当前节点、当前节点的父节点和当前节点的父节点的父节点，因为这些节点要么属于支配集（当前点的父节点），要么与支配集中的点

相连(当前节点和当前节点的父节点的父节点)。

最小点覆盖与最大独立集与上面的做法相似，都需要先求得深度优先遍历序列，按照反方向贪心以保证贪心的正确性。对于最小点覆盖来说，贪心的策略是，如果当前点和当前点的父节点都不属于覆盖集合，则将父节点加入到顶点覆盖集合，并标记当前节点和其父节点都被覆盖。对于最大独立集来说，贪心策略是如果当前节点没有被覆盖，则将当前节点加入独立集，并标记当前节点和其父节点都被覆盖。

需要注意的是由于默认程序中根节点和其他节点的区别在于根节点的父节点是其自身，所以三种问题对于根节点的处理也不同。对于最小支配集和最大独立集，需要检查根节点是否满足贪心条件，但是对于最小点覆盖不可以检查根节点。

（2）具体实现与分析

采用链式前向星存储整棵树。对于 DFS（），newpos[i]表示深度优先遍历序列的第 i 个点是哪个点，now 表示当前深度优先遍历序列已经有多少个点了。Select[]用于深度优先遍历的判重，p[i]表示点 i 的父节点的编号。对于 greedy（）,s[i]如果为 true,表示第 i 个点被覆盖。Set[i]表示点 i 属于要求的点集。

2. 树形动态规划法求解树的最小支配集

（1） 基本算法

由于这是在树上求最值的问题，显然可以用树形动态规划，只是状态的设计比较复杂。还是以最小支配集为例，为了保证动态规划的正确性，对于每个点设计了三种状态，这三种状态的意义如下：

① dp[i][0]: 表示点 i 属于支配集，并且以点 i 为根的子树都被覆盖了的情况下支配集中所包含最少的点的个数。

② dp[i][1]: i 不属于支配集，且以 i 为根的子树都被覆盖，且 i 被其中不少于 1 个子节点覆盖的情况下支配集中所包含最少的点的个数。

③ dp[i][2]: i 不属于支配集，且以 i 为根的子树都被覆盖，且 i 没被子节点覆盖的情况下支配集中所包含最少点的个数。

那么对于第一种状态，dp[i][0]等于每个儿子节点的 3 种状态（其儿子是否被覆盖没有关系）的最小值之和加 1，即只要每个以 i 的儿子为根的子树都被覆盖，再加上当前点 i，所需要的最少的点的个数，方程如下：

dp[i][0] = 1 + Σmin(dp[u][0], dp[u][1], dp[u][2]) （u 表示 i 的儿子）

对于第二种状态，如果点 i 没有子节点，那么 dp[i][1] = INF；否则，需要保证它的每个以 i 的儿子为根的子树都被覆盖，那么要取每个儿子节点的前两种状态的最小值之和，因为此时 i 点不属于支配集，不能支配其子节点，所以子节点必须已经被分配，与子节点的第三状态无关。如果当前所选的状态中，每个儿子都没有被选择进入支配集，即在每个儿子的前两种状态中，第一种状态都不是所需点最小的，那么为了满足第二种状态的定义，需要重新选择点 i 的第一个儿子的状态为第一种状态，这时取花费最少的一个点，即取 min（dp[u][0]-dp[u][1]）的儿子节点 u，强制取其第一种状态，其他的儿子节点取第二种状态，转移方程为

if (i 没有子节点) dp[i][1] = INF;

else dp[i][1] = Σmin(dp[u][0], dp[u][1]) + inc （u 为 i 的儿子）

其中对于 inc 有：

if (上面的式子中Σmin(dp[u][0], dp[u][1]) 中包含某个 dp[u][0]) inc = 0；

else inc = min（dp[u][0] – dp[u][1]） (u 为 i 的儿子)

对于第三种状态，i 不属于支配集，且以 i 为根的子树都被覆盖，又 i 没被子节点覆

盖，那么说明点 i 和点 i 的儿子节点都不属于支配集，则点 i 的第三种状态只与其儿子的第二种状态有关，方程为

dp[i][2] = Σdp[u][1] (其中 u 为 i 的儿子)

对于最小点覆盖问题，为每个点设计了两种状态，这两种状态的意义如下：

①dp[i][0]表示点 i 属于点覆盖，并且以点 i 为根的子树中所连接的边都被覆盖的情况下点覆盖集中所包含最少点的个数。

②dp[i][1]表示点 i 不属于点覆盖，并且以 i 为根的子树中所连接的边都被覆盖的情况下点覆盖集中所包含最少点的个数。

对于第一种状态 dp[i][0].等于每个儿子节点的两种状态的最小值之和加上 1，方程如下：

dp[i][0] = 1 + Σ min(dp[u][0], dp[u][1]) (u 是 i 的儿子)

对于第二种状态 dp[i][1], 要求所有与 i 相连的边都被覆盖，但是 i 点不属于点覆盖，那么 i 点所有的子节点都必须属于点覆盖，即对于点 i 的第二种状态与所有子节点的第一种状态有关，在树枝上等于所有子节点第一种状态的和。方程如下：

dp[i][1] = Σ dp[u][0] (u 是 i 的儿子)

对于最大独立集的问题，为其每个点设计了两种状态，这两种状态的意义如下：

①dp[i][0]表示点 i 属于独立集的情况下，最大独立集中点的个数。

②dp[i][1]表示点 i 不属于独立集的情况下，最大独立集中点的个数。

对于第一种状态 dp[i][0]，由于 i 点属于独立集，它的字节点都不能属于独立集，所以对于点 i 的第一种状态，只与子节点的第二种状态有关，等于其所有子节点的第二种状态的值的和加 1，方程如下：

dp[i][0] = 1 + Σdp[u][1] (u 是 i 的儿子节点)

对于第二种状态 dp[i][1], 由于点 i 不属于独立集，所以点 i 的子节点可以属于独立集，也可以不属于独立集，所取的字节点的状态应该是所表示的独立集个数较大的那个，方程如下：

dp[i][1] = Σ max(dp[u][0], dp[u][1])

3.13.2 模板代码

首先是深度优先遍历，得到深度优先遍历序列。

```
int p[maxn];
bool select[maxn];
int newpos[maxn];
int now;
int n, m;
void DFS(int x)
{
    newpos[now++] = x;
    int k;
    for (k=head[x]; k!=-1; k=edge[k].next)
    {
        if (!select[edge[k].to])
        {
            select[edge[k].to] = true;
            p[edge[k].to] = x;
            DFS(edge[k].to);
        }
    }
}
```

对于最小支配集，贪心函数如下：

```
int greedy()
{
    bool s[maxn] = {0};
```

```
    bool set[maxn] = {0};
    int ans = 0;
    int i;
    for (i=n-1; i>=0; i--)
    {
        int t = newpos[i];
        if (!s[t])
        {
            if (!set[p[t]])
            {
                set[p[t]] = true;
                ans++;
            }
            s[t] = true;
            s[p[t]] = true;
            s[p[p[t]]] = true;
        }
    }
    return ans;
}
```

对于最小点覆盖，贪心函数如下：

```
int greedy()
{
    bool s[maxn] = {0};
    bool set[maxn] = {0};
    int ans = 0;
    int i;
    for (i=n-1; i>=1; i--)
    {
        int t = newpos[i];
        if (!s[t] && !s[p[t]])
        {
            set[p[t]] = true;
            ans++;
            s[t] = true;
            s[p[t]] = true;
        }
    }
    return ans;
}
```

对于最大独立集，贪心函数如下：

```
int greedy()
{
    bool s[maxn] = {0};
    bool set[maxn] = {0};
    int ans = 0;
    int i;
    for (i=n-1; i>=0; i--)
    {
        int t = newpos[i];
        if (!s[t])
        {
            set[t] = true;
            ans++;
            s[t] = true;
            s[p[t]] = true;
        }
    }
    return ans;
}
```

使用样例：

```
int main()
{
    /*读入图信息*/
    memset(select,0,sizeof(select));
    now = 0;
```

```
    select[1] = true;
    p[1] = 1;
    DFS(1);
    printf("%d\n",greedy());
}
```

该方法经过一次深度优先遍历和一次贪心得到最终解，第一步的时间复杂度是 O（m）,由于这是一棵树，m = n-1。第二步是 O(n),一共是 O(n)。

在下面的代码中，u 表示当前正在处理的节点，p 表示 u 节点的父节点。

对于最小支配集，动态规函数如下：

```
void DP(int u, int p)
{
    dp[u][2] = 0;
    dp[u][0] = 1;
    bool s = false;
    int sum = 0,inc = INF;
    int k;
    for (k=head[u]; k!=-1; k=edge[k].next)
    {
        int to = edge[k].to;
        if (to == p)
        continue;
        DP(to, u);
        dp[u][0] += min(dp[to][0], min(dp[0][1],dp[to][2]));
        if (dp[to][0] <= dp[to][1])
        {
            sum += dp[to][0];
            s = true;
        }
        else
        {
            sum += dp[to][1];
            inc = min(inc,dp[to][0]-dp[to][1]);
        }
        if (dp[to][1] !=INF && dp[u][2]!=INF)
            dp[u][2] += dp[to][1];
        else dp[u][2] = INF;
    }
    if (inc == INF && !s)
        dp[u][1] = INF;
    else
    {
        dp[u][1] = sum;
        if (!s)
        dp[u][1] += inc;
    }
}
```

对于最小点覆盖，动态规划函数如下：

```
void DP()
{
    dp[u][0] = 1;
    dp[u][1] = 0;
    int k, to;
    for (k=head[u]; k!=-1; k=edge[k].next)
    {
        to = edge[k].to;
        if (to == p)
        continue;
        DP(to,u);
        dp[u][0] += min(dp[to][0], dp[to][1]);
        dp[u][1] += dp[to][0];
    }
}
```

对于最大独立集，动态规划函数如下：

```
void DP()
{
    dp[u][0] = 1;
    dp[u][1] = 0;
    int k, to;
    for (k=head[u]; k!=-1; k=edge[k].next)
    {
        to = edge[k].to;
        if (to == p)
        continue;
        DP(to, u);
        dp[u][0] += dp[to][1];
        dp[u][1] += max(dp[to][0], dp[to][1]);
    }
}
```

由于求的是每个点分别在几种状态下的最优值，所以要比较 dp[root][0] 和 dp[root][1] 的值，取较优的一个作为最终的答案。

由于使用的是树状动态规划，所以整个算法的时间复杂度是 O（n）.

3.14 二分图最大匹配

参考文献：

《图论及应用》冯林、金博、于瑞云 哈尔滨工业大学出版社 2012 年 3 月第一版

《图论算法理论、实现及应用》王桂平、王衍 任嘉辰 北京大学出版社 2011 年 1 月第一版

编写：周洲　　校核：程宪庆

3.14.1 基本原理

二分图基本概念：

一个无向图 G=<V, E>，如果存在两个集合 X、Y，使得 X∪Y=V， X∩Y=Φ，并且每一条边 e={x，y}有 x∈X，y∈Y，则称 G 为一个二分图(bipartite graph)。常用<X, E, Y>来表示一个二分图。若对 X 中任一 x 及 Y 中任一 y 恰有一边 e∈E，使 e = {x, y}, 则称 G 为完全二分图(complete bipartite graph)。当|X| = m，|Y| = n 时，完全二分图 G 记为 Km,n。

二分图的性质：

定理：无向图 G 为二分图的充分必要条件是，G 至少有两个顶点,且其所有回路的长度均为偶数。

匹配：设 G=<V, E>为二分图，如果 M⊆E，并且 M 中没有任何两边有公共端点。M=Φ 时称 M 为空匹配。

最大匹配:G 的所有匹配中边数最多的匹配称为最大匹配。

完全匹配:若 X(Y)中所有的顶点都是匹配 M 中的端点。则成 M 为完全匹配。若 M 既是 X-完全匹配又是 Y-完全匹配，则称 M 为 G 的完全匹配。

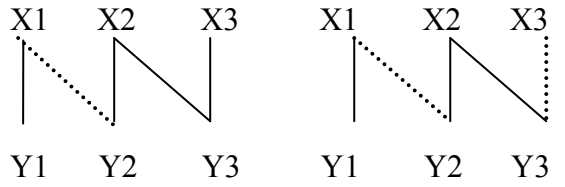
注意:最大匹配总是存在但未必唯一；X(Y)-完全匹配及 G 的完全匹配必定是最大的，但反之则不然；X(Y)-完全匹配未必存在。

下面引入几个术语：

设 G=<V, E>为二分图，M 为 G 的一个匹配。

M 中边的端点称为 M-顶点，其它顶点称为非 M-顶点。

增广路径:除了起点和终点两个顶点为非 M-顶点，其他路径上所有的点都是 M=顶点。而且它的边为匹配边、非匹配边交替出现。



图（1） 图（2）

图（1）中列出了一个匹配：[X1,Y2]。图（2）是在这个匹配的基础上的两个增广路径：

X2 Y2 X1 Y1 和 X3 Y3。

我们来验证一下:增广路径 X2→Y2→X1→Y1 中，起止点 X2、Y1 为非 M-顶点。而中间点 Y2、X1 都是 M-顶点。

边{X2, Y2}, {X1, Y1}为非匹配边，而边{Y2, X1}为匹配边，满足匹配边与非匹配边交替出现。

同理 X3→Y3 路径也满足增广路径的要求。

借助这幅图，来描述一下增广路径的性质。

1. 有奇数条边。
2. 起点在二分图的左半边，终点在右半边。
3. 路径上的点一定是一个在左半边，一个在右半边，交替出现。
4. 整条路径上没有重复的点。
5. 起点和终点都是目前还没有配对的点，而其它所有点都是已经配好对的。
6. 路径上的所有第奇数条边都不在原匹配中，所有第偶数条边都出现在原匹配中。
7. 最后，也是最重要的一条，把增广路径上的所有第奇数条边加入到原匹配中去，并把增广路径中的所有第偶数条边从原匹配中删除（这个操作称为增广路径的取反），则新的匹配数就比原匹配数增加了 1 个。

3.14.2 解题思路

下面介绍求最大匹配的一种算法：匈牙利算法

匈牙利算法的基本模式是:

```
初始时最大匹配为空
while 找得到增广路径
    do 把增广路径加入到最大匹配中去
```

搜索增广路径的方法是 DFS，写一个递归的函数。当然也可以用 BFS。 至此，理论基础部份讲完了。但是要完成匈牙利算法，还需要一个重要的定理：

如果从一个点 A 出发，没有找到增广路径，那么无论再从别的点出发找到多少增广路径来改变现在的匹配，从 A 出发都永远找不到增广路径。

3.14.3 模板代码

```
#include <iostream>
#include <string.h>
using namespace std;
int n,k;  //n矩阵规格，k星体数量
int V1,V2;         //二分图顶点集
 /*矩阵的行列分别属于二分图的两个顶点集V1、V2，其中行x∈V1，列y∈V2*/
bool grid[501][501];  //存储数据方式：可达矩阵
bool vis[501];      //记录V2的点每个点是否已被搜索过
int link[501];       //记录 V2中的点y 在 V1中 所匹配的点x的编号
int m;  //最大匹配数
bool dfs(int x)
{
    for(int y=1;y<=V2;y++)
        if(grid[x][y] && !vis[y])  //x到y相邻(有边) 且 节点y未被搜索
        {
            vis[y]=true;     //标记节点y已被搜索
            if(link[y]==0 || dfs(link[y])) //link[y]==0 : 如果y不属于前一个匹配M
            {            //find(link[y] : 如果被y匹配到的节点可以寻找到增广路
                link[y]=x;   //那么可以更新匹配M'(用M替代M')
                return true;  //返回匹配成功的标志
            }
        }
    return false;  //继续查找V1下一个x的邻接节点
}

void search(void)
{
    for(int x=1;x<=V1;x++)
    {
        memset(vis,false,sizeof(vis)); //清空上次搜索时的标记
        if(dfs(x))  //从V1中的节点x开始寻找增广路
            m++;
    }
    return;
}

int main(void)
{
    cin>>n>>k;
    V1=V2=n;

    int x,y;        //坐标(临时变量)
    for(int i=1;i<=k;i++)
    {
        cin>>x>>y;
        grid[x][y]=true;   //相邻节点标记
    }
    search();

    cout<<m<<endl;

    return 0;
}
```

3.14.4 经典题目

3.14.4.1 题目 1

1．题目出处: HDU 2063 过山车

2．题目描述: RPG girls 今天和大家一起去游乐场玩，终于可以坐上梦寐以求的过山车了。可是，过山车的每一排只有两个座位，而且还有条不成文的规矩，就是每个女生必须

找个个男生做 partner 和她同坐。但是，每个女孩都有各自的想法，举个例子把，Rabbit 只愿意和 XHD 或 PQK 做 partner，Grass 只愿意和 linle 或 LL 做 partner，PrincessSnow 愿意和水域浪子或伪酷儿做 partner。考虑到经费问题，boss 刘决定只让找到 partner 的人去坐过山车，其他的人，嘿嘿，就站在下面看着吧。聪明的 Acmer，你可以帮忙算算最多有多少对组合可以坐上过山车吗？

3．分析: 将男生看成是左集合，将女生看成是右集合，直接套用模版求最大匹配即可。

4．代码:

```
#include<stdio.h>
#include<string.h>
#define clr(x)memset(x,0,sizeof(x))
bool g[505][505];
bool v[505];
int l[505];
int n,m;
int find(int k)
{
   int i;
   for(i=1;i<=m;i++)
   {
      if(g[k][i]&&!v[i])
      {
         v[i]=1;   /* 男生 k 与女生 i 配对(i 未与别的男生配对);
                    * 女生 i 与别的男生(l[i])配对了,
                    * 但从与女生 i 配对的男生开始找, 可以找到另外一个可以匹配的 */
         if(l[i]==0||find(l[i]))
         {
            l[i]=k;
            return 1;
         }
      }
   }
   return 0;
}
int main()
{
   int i,k,p,q,tot;
   while(scanf("%d",&k),k)
   {
      scanf("%d%d",&n,&m);
      clr(g);  clr(l);
      for(i=0;i<k;i++)
      {
         scanf("%d%d",&p,&q);
         g[p][q]=1;
      }
      tot=0;
      for(i=1;i<=n;i++)  //每个男的找女友
      {
         clr(v);
         if(find(i))
         tot++;
      }
      printf("%d\n",tot);
```

```
    }
    return 0;
}
```

3.14.4.2 题目 2

1．题目出处: POJ 2771 Guardian of Decency

2．题目描述: 给你 n 个人，和四个条件，两个人只要满足其中任意一个条件就不能成为夫妻，问从中最多能找出多少人使得他们任意两个人都不可能成为夫妻。

3．分析: 该题的模型属于二分图最大独立集，把男的放到一个集合 ，女的放到另一个集合，如果两个人能成为夫妻，则他们构成一个匹配，找出最大匹配，

这里有个定理：

最大独立集 = 总权 - 最大匹配

4．代码：

```
#include<stdio.h>
#include<string.h>
#define clr(x)memset(x,0,sizeof(x))
struct node
{
    int to,next;
}q[200005];
int head[505];
int tot;
void add(int s,int u) // 前向星实现
{
    q[tot].to=u;
    q[tot].next=head[s];
    head[s]=tot++;
}
struct per
{
    int h;
    char x[3];
    char mu[103];
    char pe[103];
}p[505];
int link[505];
int v[505];
int find(int x)
{
    int i,k;
    for(i=head[x];i;i=q[i].next)
    {
        k=q[i].to;
        if(!v[k])
        {
            v[k]=1;
            if(link[k]==0||find(link[k]))
            {
                link[k]=x;
                return 1;
            }
        }
    }
```

```
    return 0;
}
int abs(int x)
{ return x>0?x:-x;  }
int ok(per a,per b)
{
    if(abs(a.h-b.h)>40)
        return 0;
    if(strcmp(a.mu,b.mu))
        return 0;
    if(strcmp(a.pe,b.pe)==0)
        return 0;
    return 1;
}
int main()
{
    int n,t,i,j,sum;
    scanf("%d",&t);
    while(t--)
    {
        scanf("%d",&n);
        tot=1;
        clr(head);
        clr(link);
        for(i=1;i<=n;i++)
            scanf("%d%s%s%s",&p[i].h,p[i].x,p[i].mu,p[i].pe);
        for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
                if(p[i].x[0]=='F'&&p[j].x[0]=='M'&&ok(p[i],p[j]))
                    add(i,j);
        sum=0;
        for(i=1;i<=n;i++)


        {
            clr(v);
            if(p[i].x[0]=='F'&&find(i))
                sum++;
        }
        printf("%d\n",n-sum);
    }
    return 0;
}
```

3.15 强连通

参考文献：

《图论及应用》冯林、金博、于瑞云 哈尔滨工业大学出版社 2012 年 3 月第一版
《图论算法理论、实现及应用》王桂平、王衍 任嘉辰 北京大学出版社 2011 年 1 月第一版

编写：周洲　　校核：程宪庆

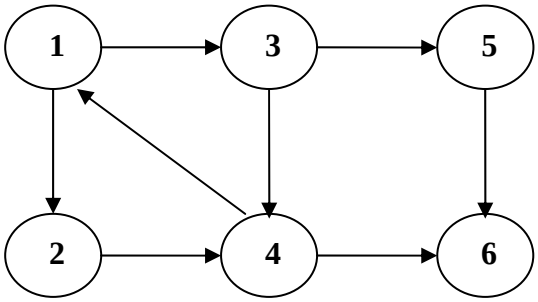
3.15.1 基本原理

定义 1：如果有向图 G 的任何两顶点都互相柯达，则称图 G 是强连通图。

定义 2：如果有向图 G 不是强连通图，它的子图 G′是强连通图，点 v 属于 G′，任意包含 v 的强连通子图也是 G′的子图，则称 G′是有向图 G 的极大强连通子图，也称强

连通分量。

下图中，子图{1,2,3,4}为一个强连通分量，因为顶点 1,2,3,4 两两可达。{5},{6}也分别是两个强连通分量。



直接根据定义，用双向遍历取交集的方法求强连通分量，时间复杂度为 O(N^2+M)。更好的方法是 Kosaraju 算法或 Tarjan 算法，两者的时间复杂度都是 O(N+M)。本章将介绍的是 Tarjan 算法，关于 Kosaraju 算法可以查阅相关资料。

3.15.2 解题思路

下面介绍 Tarjan 算法：

Tarjan 算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一棵子树。搜索时，把当前搜索树中未处理的节点加入一个堆栈，回溯时可以判断栈顶到栈中的节点是否为一个强连通分量。

定义 DFN(u)为节点 u 搜索的次序编号(时间戳)，Low(u)为 u 或 u 的子树能够追溯到的最早的栈中节点的次序号。由定义可以得出

```
Low(u)=Min
{
   DFN(u),
   Low(v),(u,v)为树枝边，u为v的父节点
   DFN(v),(u,v)为指向栈中节点的后向边(非横叉边)
}
```

当 DFN(u)=Low(u)时，以 u 为根的搜索子树上所有节点是一个强连通分量。

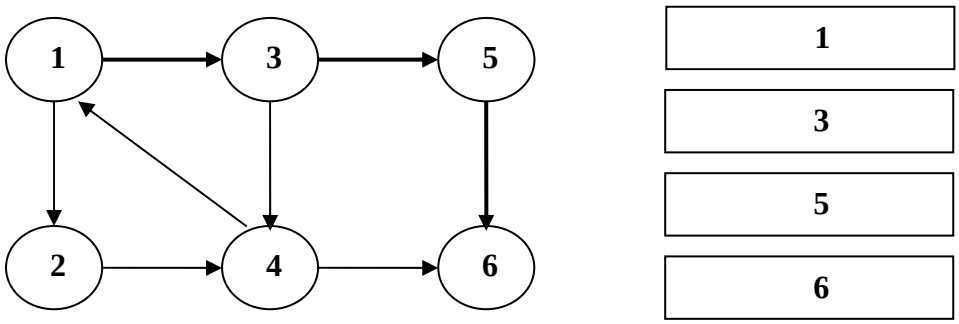
算法伪代码如下：

```
tarjan(u)
{
    DFN[u]=Low[u]=++Index              // 为节点u设定次序编号和Low初值
    Stack.push(u)                      // 将节点u压入栈中
    for each (u, v) in E                 // 枚举每一条边
        if (v is not visted)               // 如果节点v未被访问过
            tarjan(v)                   // 继续向下找
            Low[u] = min(Low[u], Low[v])
        else if (v in S)                   // 如果节点v还在栈内
            Low[u] = min(Low[u], DFN[v])
    if (DFN[u] == Low[u])               // 如果节点u是强连通分量的根
        repeat
            v = S.pop                  // 将v退栈，为该强连通分量中一个顶点
            print v
        until (u== v)
}
```

接下来是对算法流程的演示。

从节点 1 开始 DFS，把遍历到的节点加入栈中。搜索到节点 u=6 时，DFN[6]=LOW[6]，找到了一个强连通分量。退栈到 u=v 为止，{6}为一个强连通分量。

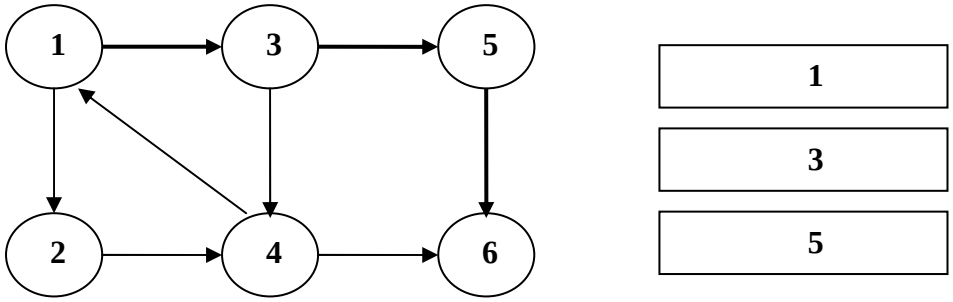
DFN: 1　　DFN:2　　DFN:3



DFN:4 LOW:4

返回节点 5，发现 DFN[5]=LOW[5]，退栈后{5}为一个强连通分量。

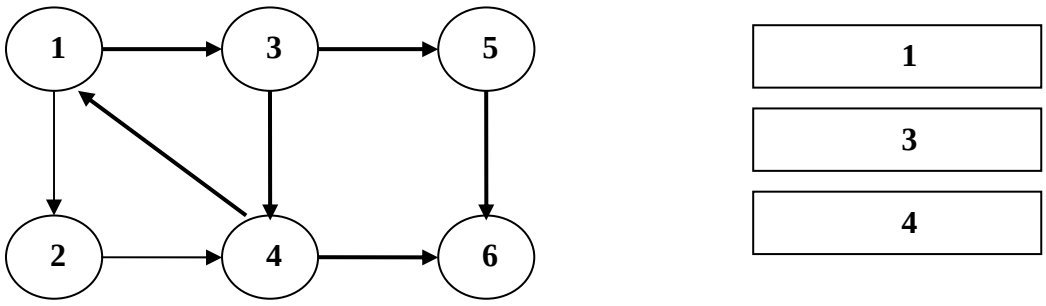
DFN:1　　DFN:2　　DFN:3 LOW:3



DFN:4 LOW:4

返回节点 3，继续搜索到节点 4，把 4 加入堆栈。发现节点 4 向节点 1 有后向边，节点 1 还在栈中，所以 LOW[4]=1。节点 6 已经出栈，(4,6)是横叉边，返回 3，(3,4)为树枝边，所以 LOW[3]=LOW[4]=1。

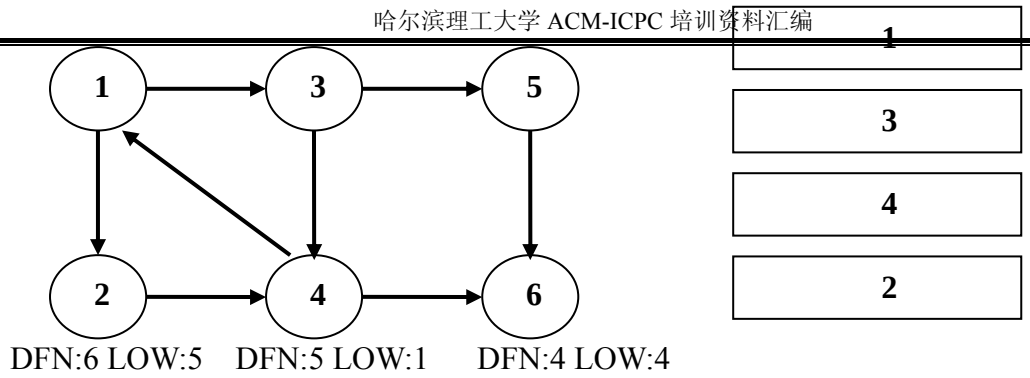
DFN:1　　DFN:2 LOW:1　　DFN:3 LOW:3



DFN:5 LOW:1　　DFN:4 LOW:4

继续回到节点 1，最后访问节点 2。访问边(2,4)，4 还在栈中，所以 LOW[2]=DFN[4]=5。返回 1 后，发现 DFN[1]=LOW[1]，把栈中节点全部取出，组成一个连通分量{1,3,4,2}。

DFN:1 LOW:1　　DFN:2 LOW:1　　DFN:3 LOW:3



至此，算法结束。经过该算法，求出了图中全部的三个强连通分量{1,3,4,2},{5},{6}。

可以发现，运行 Tarjan 算法的过程中，每个顶点都被访问了一次，且只进出了一次堆栈，每条边也只被访问了一次，所以该算法的时间复杂度为 O(N+M)。

3.15.3 模板代码

```
void tarjan(int i)
{
    int j;
    DFN[i]=LOW[i]=++Dindex;
    instack[i]=true;
    Stap[++Stop]=i;
    for (edge *e=V[i];e;e=e->next)
    {
        j=e->t;
        if (!DFN[j])
        {
            tarjan(j);
            if (LOW[j]<LOW[i])
                LOW[i]=LOW[j];
        }
        else if (instack[j] && DFN[j]<LOW[i])
            LOW[i]=DFN[j];
    }

    if (DFN[i]==LOW[i])
    {
        Bcnt++;
        do
        {
            j=Stap[Stop--];
            instack[j]=false;
            Belong[j]=Bcnt;
        }
        while (j!=i);
    }
}
void solve()
{
    int i;
    Stop=Bcnt=Dindex=0;
    memset(DFN,0,sizeof(DFN));
    for (i=1;i<=N;i++)
        if (!DFN[i])
            tarjan(i);
}
```

3.15.4 经典题目

3.15.4.1 题目 1

1．题目出处: HDU 1269 迷宫城堡

2．题目描述: 给一个有 n 个节点的有向图，和 m 条有向边，问这个图中是否任意两个点都能互达。

3．分析: 任意两个点能够互达，即要求图的强连通分量的个数为一，求出图的强连通子图的个数即可。

4．代码;

```
#include<stdio.h>
#include<string.h>
#define clr(x)memset(x,0,sizeof(x))
const int maxn=100010;
struct node //前向星实现
{
   int to;
   int next;
}q[100010];
int head[100010];
int tot;
int n,m;
void add(int s,int u)
{
   q[tot].to = u;
   q[tot].next = head[s];
   head[s] = tot++;
}
int dfn[maxn];
int low[maxn];
int stack[maxn];
int ti,sn,top;
bool instack[maxn];
void tarjan(int u)
{
   dfn[u] = low[u] = ++ti;  // 取时间戳
   stack[++top] = u;     // 当前节点入栈
   instack[u] = true;
   int k, i;
   for (i=head[u]; i; i=q[i].next)
   {
      k = q[i].to;
      if (dfn[k] == 0)    // k 没有访问过,k是 u 的子女,（u,k）为树枝边
      {
         tarjan(k);
         if(low[u] > low[k])   // 取子女可以达到的最早时间戳
            low[u] = low[k];
      }
      else if(instack[k] && low[u]>dfn[k])  // k 访问过，k是u的祖先，（u,k）是一条回边
         low[u] = dfn[k];                // 取回边中可以达到的最早时间戳
   }
   if(low[u] == dfn[u])        // 以u 为根的强连通分量已经找到
   {
```

```
        sn++;                  // 累计强连通分量的个数
        do
        {
            k = stack[top--];
            instack[k] = false;
        }
        while(k!=u);
    }
}
int main()
{
    int a,b,i;
    while(scanf("%d %d",&n,&m),n)
    {
        clr(head);
        clr(instack);
        clr(dfn);
        ti = sn = 0;  // ti: 时间戳序号，sn: 强连通分量的序号
        tot = 1;
        top = -1;
        while (m--)
        {
            scanf("%d %d",&a,&b);
            add(a,b);
        }
        for (i=1; i<=n; i++)
            if (dfn[i] == 0) // 以没遍历到的点为搜索树的根进行搜索
                tarjan(i);
        if (sn>1)  // 如果强连通分量的个数超过 1
            printf("No\n");
        else printf("Yes\n");
    }
    return 0;
}
```

5．思考与扩展：可以借鉴北大培训教材中做法。

3.15.4.2 题目 2

1．题目出处: HDU 2767 Proving Equivalences

2．题目描述: 在有 n 个节点的有向图中 有 m 条边，问最少需要增加多少条边可以将原图变成一个强连通图。

3．分析: 因为图中每个环都满足强连通的性质，可以用强连通分量的算法将原图中的环进行染色缩点。

缩点的最大好处在于把一个杂乱无章的有向图变成一个有向无环图，而在有向无环图中，有两种点比较特殊：一种是入度为 0 的点，另一种是 出度为 0 的点。

题目问要增加多少条边使得原图变成强连通图，其实只要知道在树根和叶子之间加多少条边，假如 r 为根的个数， g 为叶子的个数，答案即为 max(r,g)。

特殊情况是当缩点之后只有一个点时，答案为 0。

4．代码：

```
#include<stdio.h>
#include<string.h>
```

```
#define clr(x)memset(x,0,sizeof(x))
const int maxn=20002;

struct node    // 前向星实现
{
   int to;
   int next;
}q[60000];
int head[maxn];
int tot;
void add(int s,int u)
{
   q[tot].to = u;
   q[tot].next = head[s];
   head[s] = tot++;
}
bool ins[maxn];
int color[maxn];
int dfn[maxn],low[maxn],stack[maxn];
int ti, sn, top;
void tarjan(int u)
{
   dfn[u] = low[u] = ++ti;
   stack[++top] = u;
   ins[u] = true;
   int i, k;
   for (i=head[u]; i; i=q[i].next)
   {
      k = q[i].to;
      if (dfn[k] == 0)
      {
         tarjan(k);
         if (low[k] < low[u])
            low[u] = low[k];
      }
      else if (ins[k] && dfn[k]<low[u])
         low[u] = dfn[k];
   }
   if(dfn[u] == low[u])
   {
      sn++;
      do
      {
         k = stack[top--];
         ins[k] = false;
         color[k] = sn;    // 强连通分量染色
      }
      while (k!=u);
   }
}
int id[maxn], od[maxn];   // 统计入度和出度
int main()
{
   int t, i, j, k;
   int a, b, n, m;
   scanf("%d",&t);
   while (t--)
   {
      scanf("%d %d",&n,&m);
```

```
        tot = 1;
        top = -1;
        sn = ti = 0;
        clr(low);
        clr(dfn);
        clr(ins);
        clr(head);

        while (m--)
        {
            scanf("%d %d",&a,&b);
            add(a,b);
        }
        for (i=1; i<=n; i++)
            if (dfn[i] == 0)
                tarjan(i);
        if (sn == 1)
            printf("0\n");
        else
        {
            clr(id);
            clr(od);
            int in=0,out=0;
            for (i=1; i<=n; i++)
                for(j=head[i]; j; j=q[j].next)
                {
                    k = q[j].to;
                    if (color[i] != color[k])
                    {
                        id[color[k]]++;
                        od[color[i]]++;
                    }
                }
            for (i=1; i<=sn; i++)
            {
                if (id[i]==0)
                    in++;
                if (od[i]==0)
                    out++;
            }
            printf("%d\n",in>out?in:out);
        }
    }
    return 0;
}
```

3.16 重连通

参考文献：

《图论及应用》冯林、金博、于瑞云 哈尔滨工业大学出版社 2012 年 3 月第一版

《图论算法理论、实现及应用》王桂平、王衍 任嘉辰 北京大学出版社 2011 年 1 月第一版

编写：周洲　　校核：程宪庆

3.16.1 基本原理

定义 1：在一个无向连通图中，如果有一个顶点集合，删除这个顶点集合，以及这个集合中所有顶点相关联的边以后，原图变成多个连通块，就称这个点集为**割点集合**。一个图的**点连通度**的定义为，最小割点集合中的顶点数。

类似的，如果有一个边集合，删除这个边集合以后，原图变成多个连通块，就称这个点集为**割边集合**。一个图的**边连通度**的定义为，最小割边集合中的边数。

定义 2: 如果一个无向连通图的点连通度大于 1，则称该图是**点双连通的(point biconnected)**，简称**双连通**或**重连通**。一个图有割点，当且仅当这个图的点连通度为 1，则割点集合的唯一元素被称为**割点(cut point)**，又叫**关节点(articulation point)**。

如果一个无向连通图的边连通度大于 1，则称该图是**边双连通的(edge biconnected)**，简称双连通或重连通。一个图有桥，当且仅当这个图的边连通度为 1，则割边集合的唯一元素被称为**桥(bridge)**，又叫**关节边(articulation edge)**。

可以看出，点双连通与边双连通都可以简称为双连通，它们之间是有着某种联系的，下文中提到的双连通，均既可指点双连通，又可指边双连通。

定义 3:在图 G 的所有子图 G’中，如果 G’是双连通的，则称 G’为**双连通子图**。如果一个双连通子图 G’它不是任何一个双连通子图的真子集，则 G’为**极大双连通子图**。**双连通分支(biconnected component)**，或重连通分支，就是图的极大双连通子图。特殊的，点双连通分支又叫做**块**。

本节主要介绍求桥和割点的 Tarjan 算法。

3.16.2 解题思路

3.16.2.1 点双连通求解

思路和有向图求强连通分量的 Tarjan 算法类似。

顶点 u 是关节点的充要条件如下：

在深搜树中，时间戳为 dfn[k] ，当 k 满足① ②中一个时，k 为割点

① k 为深搜树的根，当且仅当 k 的儿子个数>=2 时 k 为割点

② k 为深搜树的中间节点（k 既不是根也不是叶），那么 k 必然有父亲和儿子；

对于①是显然的，根结点 k 一旦有 2 个以上的分支，那么删除 k 必然出现森林；

对于②首先注意 low[son]>= dfn[k]这个条件，意思就是“k 的儿子 son 的辈分最高的祖先（暂且设其为 w）的深度，比 k 的深度要深（或者等于 k 的深度，此 时 k 就是 w），就是说 k 的辈分比 w 更高（深度更浅），那么一旦删除 k，son 所在的网络势必和 k 的 father 所在的网络断开”，那么 k 就是割点。

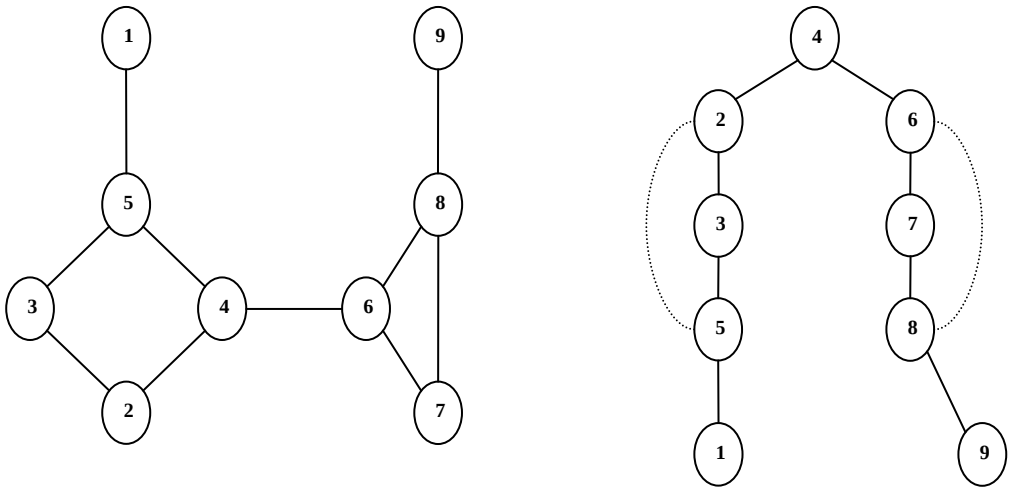
因此，可以对图 G 的每个顶点 u 定义一个 low 值：low[u]是从 u 或 u 的子孙出发通过回边可以达到的最低深度优先数。其中 Low[u]的定义如下：

```
Low(u)=Min
{
   dfn(u)
   low[v]  v是u 的一个子女
   dfn[v]  v与 u邻接，且边（u,v）是一条回边
}
```

3.16.2.2 边双连通求解

割边的求解过程和求割点的方法类似，判断方法是： 无向图中的一条边（u,v）是桥，当且仅当（u,v）是生成树中的边，且满足 dfn[u]<low[v]。

例如，图 a 所示的无向图，如果从顶点 4 开始 DFS 搜索，各个顶点的 dfn[]值和 low[]值如图 b 所示。根据以上判断方法，可判断出边（1，5）（4，6）（8，9）为割边。



(a) 各个顶点的 dfn[]值和 low[]值 （b）深度优先搜索树

求解边双连通分量的时候，我们正常的做法是求桥，删桥，再求连通分支，而我们有一种更为简便的方法，在求桥的时候，只要对 Tarjan 算法做一些变化即可。

我们之前规定 low[u]是其子孙通过一条返祖边直接到达的点，把这个改成是其子孙可以连续通过多条返祖边所能到达的点。那么 low[u]=min(low[v],dfn[u]);这样做的缺陷是，不能求割点了，多次返祖会导致求割点的错误，在多环两两以单个点相连排成一条线，且每两个连接点间只有一条边的情况中，那些连接点本应是割点，但是在 dfs 过程中，这些连接点之间的边又恰好不是树枝边的话，low[u]可能会通过多次返祖,从一个割点不断的经过这些割点到达最上边的割点才记录下 low[u]。这样中间的割点就都不符合 dfn(u)<=low[v]了。

但是这样做有一个好处，就是所有的对于边的双连通分支都以 low 标记出来了，即属于同一双连通分支的所有点的 low 都等于同一个值。因为在不遇到桥的情况下，low 可以返祖到该连同分支在遍历树中的最高点（dfn 最小的点）。这样就相当于整理出了所有的对于边的双连通分支,

3.16.3 模板代码

关节点求解的算法实现：

```
void tarjan(int p,int u)
{
    dfn[u] = low[u] = ++ti;   // 时间戳
    int i, k;
    int son = 0;     //  当前节点儿子的个数
    for (i=head[u]; i!=-1; i=edge[i].next)   //前向星实现
    {
        k = edge[i].to;
        if (k == p)
            continue;
```

```
        if (dfn[k] == 0)
        {
            son++;
            tarjan(u,k);
            if (low[k] < low[u])
                low[u] = low[k];
            if((u!=1 && dfn[u]<=low[k]) || (u==1&&son>=2))
                istcc[u]=1;   // 条件符合，u是关节点
        }
        else
            low[u] = min(low[u],dfn[k]);
    }
}
```

桥求解的算法实现：

```
void tarjan(int p,int u)
{
   low[u] = ++ti;
   int k, i;
   for (i=head[u]; i!=-1; i=edge[i].next)
   {
      k = edge[i].to;
      if (k == p) continue;
      if (low[k] == 0)
         tarjan(u,k);
      if (low[k] < low[u])
         low[u] = low[k];
   }
}
```

3.16.4 经典题目

3.16.4.1 题目 1

1．题目出处: POJ 1523 SPF

2．题目描述: 找出无向图的割点，并判断每个割点去掉后能形成多少个双连通分量。

3．分析:直接套用求割点的模版，不过需要做一点小小的变化。

4．代码:

```
#include<stdio.h>
#include<string.h>
#define min(a,b)(a)<(b)?(a):(b)
#define max(a,b)(a)>(b)?(a):(b)
#define clr(x)memset(x,0,sizeof(x))
const int maxn=1010;

struct node  // 前向星实现
{
   int to,next;
}e[10010];
int tot;
int head[maxn];
void add(int s,int u)
{
   e[tot].to = u;
   e[tot].next = head[s];
```

```
    head[s] = tot++;
}
int st,en;
int ti;
int dfn[maxn];
int low[maxn];
int num[maxn];
void dfs(int p, int u)
{
    dfn[u] = low[u] = ++ti;
    int i, k;
    int son = 0;
    for (i=head[u]; i; i=e[i].next)
    {
        k = e[i].to;
        if (k!=p && dfn[k]>0)
            low[u] = min(low[u],dfn[k]);
        else if(dfn[k] == 0)
        {
            son++;
            dfs(u,k);
            if (low[k] < low[u])
                low[u] = low[k];
            if ((u==st && son>=2) || (u!=st && dfn[u]<=low[k]))
                num[u]++;  //统计删除该点之后形成的连通块的个数
        }
    }
}
int main()
{
    int a,b,i;
    int ca=1;
    while (scanf("%d",&a),a)
    {
        scanf("%d",&b);
        ti = 0;
        tot = 1;
        clr(head);
        st = 1005;
        en = 0;
        st = min(st,a);
        st = min(st,b);
        en = max(en,a);
        en = max(en,b);
        add(a,b);
        add(b,a);
        while (scanf("%d",&a),a)
        {
            scanf("%d",&b);
            st = min(st,a);
            st = min(st,b);
            en = max(en,a);
            en = max(en,b);
            add(a,b);
            add(b,a);
        }
        clr(dfn); clr(low);
        clr(num);
        dfs(0,st);
```

```
        printf("Network #%d\n",ca++);
        bool flag = true;
        for (i=st; i<=en; i++)
            if(num[i])
            {
                flag=false;
                printf("  SPF node %d leaves %d subnets\n",i,num[i]+1);
            }
        if (flag)
        printf("  No SPF nodes\n");
        printf("\n");
    }
    return 0;
}
```

3.16.4.2 题目 2

1．题目出处: POJ 3352 Road Construction

2．题目描述: 一个有 N 个景点的岛，任意两个景点都有道路相连，当道路施工时，游客便不能在该道路上通行，问至少再增加几条道路可以使得在任一条道路维修的情况下，游客都能从岛上任意一个景点到达另一个景点。

3．分析: 当原图中存在桥的时候，即原图不是双连通图的时候，目的就无法达到，所以这题的关键在于 **需要增加几条边可以使得原图中不存在桥**。

可以先找出原图中所有的边双连通分量，对其进行缩点，缩点具体做法可以用上面讲解的那种求法，将图中 low 值相同的节点看作一个点，缩点后，原图可以看成是一颗树，而要使得一棵树变为一个双连通图，这里有一个定理：

增加的边数 = （树中总度数为 1 的节点数+1）/ 2

4．代码：

```
#include<stdio.h>
#include<string.h>
#define clr(x)memset(x,0,sizeof(x))
const int maxn=5005;

struct node   //前向星实现
{
    int to,next;
}e[10005];
int tot;
int head[maxn];
void add(int s,int u)
{
    e[tot].to=u;
    e[tot].next=head[s];
    head[s]=tot++;
}

int ti,top,sn;
int low[maxn];
void tarjan(int p,int u)
{
    low[u] = ++ti;
    int k,i;
    for (i=head[u]; i; i=e[i].next)
```

```
    {
        k = e[i].to;
        if (k == p)
            continue;
        if (low[k] == 0)
            tarjan(u,k);
        if (low[k] < low[u])
            low[u] = low[k];
    }
}

int main()
{
    int n, m, re;
    int i, j, k, a, b;
    int degree[maxn];
    scanf("%d %d",&n,&m);
    ti = 0;
    tot = 1;
    clr(head);
    clr(low);
    while (m--)
    {
        scanf("%d %d",&a,&b);
        a--; b--;
        add(a, b);
        add(b, a);
    }
    tarjan(0,0);
    clr(degree);
    for (i=0; i<n; i++)
        for (j=head[i]; j; j=e[j].next)
        {
            k = e[j].to;
            if (low[k] != low[i])
                degree[low[i]]++;
        }
    re = 0;
    for (i=1; i<=ti; i++)
        if (degree[i] == 1)
            re++;
    printf("%d\n",(re+1)/2);
    return 0;
}
```

3.17 2-SAT

参考文献：

《图论及应用》冯林、金博、于瑞云 哈尔滨工业大学出版社 2012 年 3 月第一版
《图论算法理论、实现及应用》王桂平、王衍 任嘉辰 北京大学出版社 2011 年 1 月第一版

编写：周洲　　校核：程宪庆

3.17.1 基本原理

定义 1：布尔变量 x，假如逻辑运算“或”和“与”分别用“∨”和“∧ ”来表示，¬x 表示 x 的非，布尔表达式是用算术运算符号连接起来的变量所构成的代数表达式。给定每个变量 x 的一个值 p(x)，可以像计算代数表达式一样计算布尔表达式的值。如果存在一个真值分配，使得布尔表达式的取值为真，则这个布尔表达式称为可适定性的，简称 SAT。

例如（x1∨x2）∧(¬x1∨¬x2) 这个布尔表达式，如果 p(x1)=真，p(x2)=假，则表达式的值为真，则这个表达式是适定性的。不是所有的布尔表达式都是可适定的。例如 x1∧¬x2∧(¬x1∨x2)，则不管 p(x1),p(x2)取何值，表达式都不可能为真，因此这个布尔表达式是不可适定的。

适定性问题的一般形式 X=｛x1,x2..,xn｝为一个有限的布尔变量集，包含 x1,x2，..,xn 的“或”和“与”，运算的 m 个句子 C1,C2,..,Cm,布尔表达式 C1∧C2∧,..,∧Cm 是否可适定。

布尔表达式由用“与”连接起来的一些句子构成，则称这个表达式为“合取范式”。

定义 2:对于给定的句子 C1,C2,..,Cm,如果 max{|Ci|}=k(1≦i≦m),则称此适定性问题为 k 适定性问题，简称 k-SAT。

当 k>2 时，k-SAT 是 NP 完全的，本节主要讨论 k=2 时的情况。

3.17.2 解题思路

下面我们从一道例题来认识 2-SAT 问题，并提出对一类 2-SAT 问题通用的解法。

Poi 0106 Peaceful Commission [和平委员会]

某国有 n 个党派，每个党派在议会中恰有 2 个代表。现在要成立和平委员会 ，该会满足：

每个党派在和平委员会中有且只有一个代表

如果某两个代表不和，则他们不能都属于委员会

代表的编号从 1 到 2n，编号为 2a-1、2a 的代表属于第 a 个党派

输入 n（党派数），m（不友好对数）及 m 对两两不和的代表编号

其中 1≤n≤8000，0≤m ≤20000

求和平委员会是否能创立。

若能，求一种构成方式。

例：输入：3　2　　输出：1

1　3　　4

2　4　　5

分析：原题可描述为：

有 n 个组，第 i 个组里有两个节点 Ai, Ai' 。需要从每个组中选出一个。而某些点不可以同时选出（称之为不相容）。任务是保证选出的 n 个点都能两两相容。（在这里把 Ai, Ai' 的定义稍稍放宽一些，它们同时表示属于同一个组的两个节点。也就是说，如果我们描述 Ai，那么描述这个组的另一个节点就可以用 Ai'）

初步构图

如果 Ai 与 Aj 不相容，那么如果选择了 Ai，必须选择 Aj` ；同样，如果选择了 Aj，就必须选择 Ai`。

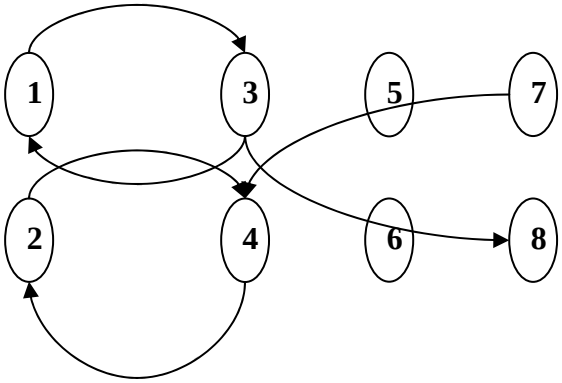
Ai ⟶ Aj`

Aj ⟶ Ai`

这样的两条边对称

我们从一个例子来看：

假设 4 个组，不和的代表为：1 和 4，2 和 3，7 和 3，那么构图：



图（1）

假设：

首先选 1

3 必须选，2 不可选

5、6 可以任选一个

8 必须选，4、7 不可选

矛盾的情况为：

存在 Ai，使得 Ai 既必须被选又不可选。

得到算法 1：

枚举每一对尚未确定的 Ai, Ai‘ ，任选 1 个，推导出相关的组，若不矛盾，则可选择；否则选另 1 个，同样推导。若矛盾，问题必定无解。

此算法正确性简要说明：

由于 Ai,Ai' 都是尚未确定的，它们不与之前的组相关联，前面的选择不会影响 Ai, Ai' 。

算法的时间复杂度在最坏的情况下为 O(nm)。

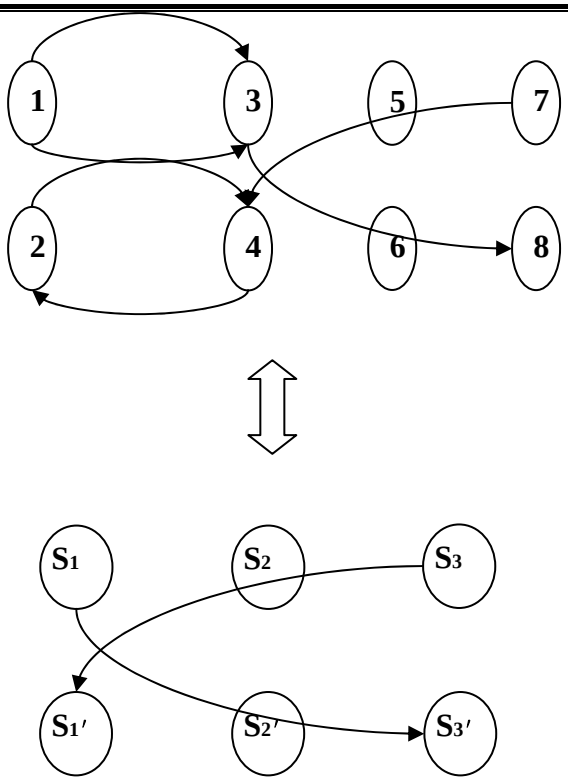
在这个算法中，并没有很好的利用图中边的对称性

观察图（1）可以发现，1 和 3 构成一个环，这样 1 和 3 要么都被选中，要么都不选。2 和 4 也同样如此。

在每个一个环里，任意一个点的选择代表将要选择此环里的每一个点。不妨把环收缩成一个子节点。新节点的选择表示选择这个节点所对应的环中的每一个节点。

对于原图中的每条边 Ai⟶Aj（设 Ai 属于环 Si，Aj 属于环 Sj）如果 Si≠Sj，则在新图中连边：

Si⟶ Sj



这样构造的有向无环图和原图是等价的，这样我们就可以用之前介绍过的强连通分量的算法把图转化成有向无环图，在这个基础上，如果存在一对 Ai, Ai'属于同一个环，则判无解，否则将采用拓扑排序，以自底向上的顺序进行推导，一定能找到可行解。

下面给出 2-SAT 问题中常用的建边方式：

2-SAT 中元素关系常见有以下 11 种

A[x]
NOT A[x]
A[x] AND A[y]
A[x] AND NOT A[y]
A[x] OR A[y]
A[x] OR NOT A[y]
NOT (A[x] AND A[y])
NOT (A[x] OR A[y])
A[x] XOR A[y]
NOT (A[x] XOR A[y])
A[x] XOR NOT A[y]
And 结果为 1：建边 ~x->y, ~y->x (两个数都为 1)
And 结果为 0：建边 y->~x , x->~y(两个数至少有一个为 0)
OR 结果为 1：建边 ~x->y , ~y->x(两个数至少有一个为 1)
OR 结果为 0：建边 x->~x , y->~y(两个数都为 0)
XOR 结果为 1：建边 x->~y , ~x->y , ~y->x , y -> ~x (两个数一个为 0，一个为 1)
XOR 结果为 0：建边 x->y , ~x->~y , y->x ~y->~x(两个数同为 1 或者同为 0)

对于一般判定是不是有解的情况，我们可以直接采用 tarjan 算法求强联通，然后缩点，如果 x 与~x 染色相同，说明无解，否则有解。有的时候，可能需要用二分+tarjan 算法

3.17.3 模板代码

在求强连通分量的时候可以参考之前讲解的 Tarjan 算法，其他的代码实现部分读者可以通过下面的一些题目归纳出来。

3.17.4 经典题目

3.17.4.1 题目 1

1．题目出处：POJ 3207 Ikki's Story IV - Panda's Trick

2．题目描述：已知一个圆上顺时针放着 n 个点，这 n 个点中有 m 对顶点之间有连线，连线要么在园外要么在圆内，每个点最多连接一条边，问是否存在一种连接情况满足所有的边都不相交。

3．分析: 将每条边看成两个点 i , i+m 分别表示边在内部和在外部，如果两条边 i, j 的端点存在序号上的交叉，则这两对点之间的连线一个在外部一个在内部即如果存在 i，则必然存在 j+m ，如果存在 j ，则必然存在 i+m，如果存在 i+m，则必然存在 j ，如果存在 j+m ，则必然存在 i，

建图的时候连的边为

i -> j+m
j -> i+m
i+m -> j
j+m -> i

求出强连通分量并染色，判断是否存在冲突的情况即可。

4．代码：

```
#include<stdio.h>
#include<string.h>
#define maxn 1100
#define clr(x)memset(x,0,sizeof(x))
#define min(a,b)(a)<(b)?(a):(b)
#define max(a,b)(a)>(b)?(a):(b)
struct Edg   //前向星实现
{
   int u, v;
}e[1100];
struct node
{
   int to, next;
}edge[1000000];
int tot;
int head[maxn];
void add(int s,int t)
{
   edge[tot].to = t;
   edge[tot].next = head[s];
   head[s] = tot++;
}

int dfn[maxn];
int low[maxn];
int sta[maxn];
int ins[maxn];
int col[maxn];  // 染色标记
int top,sn, ti;
```

```
void tarjan(int u)
{
    dfn[u] = low[u] = ++ti;
    sta[++top] = u;
    ins[u] = 1;
    int i, k;
    for (i=head[u]; i; i=edge[i].next)
    {
        k = edge[i].to;
        if (dfn[k] == 0)
        {
            tarjan(k);
            low[u] = min(low[u], low[k]);
        }
        else if (ins[k])
            low[u] = min(low[u],dfn[k]);
    }
    if (dfn[u] == low[u])
    {
        sn++;
        do
        {
            k = sta[top--];
            ins[k] = 0;
            col[k] = sn;
        }while (k!=u);
    }
}
int main()
{
    int n,m;
    int i,j;
    while(scanf("%d %d",&n,&m)!=EOF)
    {
        for (i=0; i<m; i++)
        {
            scanf("%d %d",&e[i].u,&e[i].v);
            if (e[i].u > e[i].v)
            {
                int tmp = e[i].u;
                    e[i].u = e[i].v;
                    e[i].v = tmp;
            }
        }
        tot=1;
        clr(head);
        for (i=0; i<m; i++)
            for (j=i+1; j<m; j++)
                if (e[i].u<e[j].v && e[i].u>e[j].u && e[i].v>e[j].v
                  ||e[i].v>e[j].u && e[i].v<e[j].v && e[i].u<e[j].u)
                {
                    add(i, j+m);
                    add(j, i+m);
                    add(j+m, i);
                    add(i+m, j);
                }
        sn = top = ti = 0;
        clr(dfn);  clr(low);
        clr(ins);  clr(col);
```

```
        for (i=0; i<2*m; i++)
            if (dfn[i] == 0)
                tarjan(i);
        for (i=0; i<m; i++)
            if (col[i] == col[i+m])  // 一个强连通分量里面同时出现了某个元素的两种状态，有冲突
                break;
        if(i == m)
            printf("panda is telling the truth...\n");
        else
            printf("the evil panda is lying again\n");
    }
    return 0;
}
```

3.17.4.2 题目 2

1．题目出处: POJ 3683 Priest John's Busiest Day

2．题目描述:有 n 对新人结婚, 只有一个牧师，知道了每个婚礼的开始时间 s,和结束时间 t 和需要牧师的主持时间 las, 牧师可以选在在[s,s+las]或[t-las,t]两个时间段内主持,问是否存在一个时间安排，使得所有新人都可以得到牧师的主持。

3．分析:由于每个时间段对应着两种状态，二者取其一，不难想到用 2-SAT 来求解。

建图方式如下:

每个婚礼的两个时间段 i = [s,s+las],i +n = [t-las,t]

如果 i 和 j 冲突，建边 i —> j + n

如果 i 和 j + n 冲突，建边 i —> j

如果 i + n 和 j 冲突，建边 i + n —> j + n

如果 i + n 和 j + n 冲突，建边 i + n —> j

求出强连通分量并染色，判断是否有 i 和 i+n 在一个集合的情况，如果有则不存在，否则反向拓扑排序，找出一组解即可。

4．代码:

```
#include<stdio.h>
#include<string.h>
#define min(a,b)(a)<(b)?(a):(b)
#define max(a,b)(a)>(b)?(a):(b)
#define clr(x)memset(x,0,sizeof(x))
#define maxn 2100
#define maxm 3000000

struct node    // 前向星实现
{
    int from, to, next;
}e[maxm],sed[maxm];
int head[maxn];
int sorh[maxn];
int tot;
int tt;

void add(int s,int t)
{
    e[tot].from = s;
    e[tot].to = t;
    e[tot].next = head[s];
    head[s] = tot++;
}
```

```
void add2(int s,int t)
{
    sed[tt].to = t;
    sed[tt].next = sorh[s];
    sorh[s] = tt++;
}
int ti,sn,top,n;
int low[maxn];
int dfn[maxn];
int ins[maxn];
int sta[maxn];
int col[maxn];
int sco[maxn];
int ct[maxn];
int q[maxn];
int ind[maxn];  // 入度
int res[maxn];

void tarjan(int u)
{
    dfn[u] = low[u] = ++ti;
    ins[u] = 1;
    sta[++top] = u;
    int i, k;
    for (i=head[u]; i; i=e[i].next)
    {
        k = e[i].to;
        if (dfn[k] == 0)
        {
            tarjan(k);
            low[u] = min(low[u],low[k]);
        }
        else if (ins[k])
            low[u] = min(low[u],dfn[k]);
    }
    if (dfn[u] == low[u])
    {
        sn++;
        do
        {
            k = sta[top--];
            ins[k] = 0;
            sco[k] = sn; // 染色
        }while(k!=u);
    }
}
struct edge
{
    int s1,e1;
    int s2,e2;
}p[maxn];
char st[22],en[22];
int main()
{
    scanf("%d",&n);
    int i, j, k, las;
    int front, rear;
    for (i=0; i<n; i++)
    {
```

```
        scanf("%s %s %d",st,en,&las);
        p[i].s1 = ((st[0]-'0')*10+st[1]-'0')*60+(st[3]-'0')*10+st[4]-'0';
        p[i].e1 = p[i].s1+las;
        p[i].e2 = ((en[0]-'0')*10+en[1]-'0')*60+(en[3]-'0')*10+en[4]-'0';
        p[i].s2 = p[i].e2-las;
    }
    clr(head);
    tot = 1;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
        {
            if (i == j)
                continue;
            if (p[i].s1<p[j].e1 && p[j].s1<p[i].e1)
                add(i,j+n);
            if (p[i].s1<p[j].e2 && p[j].s2<p[i].e1)
                add(i,j);
            if (p[i].s2<p[j].e1 && p[j].s1<p[i].e2)
                add(i+n,j+n);
            if (p[i].s2<p[j].e2 && p[j].s2<p[i].e2)
                add(i+n,j);
        }
        ti = sn = top = 0;
        clr(sco);  clr(dfn);
        clr(low);  clr(ins);
        for (i=0; i<2*n; i++)
            if (!dfn[i])
                tarjan(i);
        int flag=0;
        for (i=0; i<n; i++)
        {
            if (sco[i] == sco[i+n])
            flag = 1;
            ct[sco[i]] = sco[i+n];
            ct[sco[i+n]] = sco[i];
        }
        if(flag)
            goto loop;
        tt = 1;
        clr(sorh);
        clr(ind);
        clr(col);
        for (i=1; i<tot; i++)
            if (sco[e[i].from] != sco[e[i].to])  // 如果不在同一个强连通分量里面
            {
                add2(sco[e[i].to],sco[e[i].from]);
                ind[sco[e[i].from]]++;
            }
        front=0, rear=0;
        for (i=1; i<=sn; i++) // 反向拓扑排序
            if (ind[i] == 0)
                q[rear++] = i;
        while (front<rear)
        {
            int x = q[front++];
            if (col[x] == 0)
            {
                col[x] = 1;
                col[ct[x]] = -1;
```

```
            }
            for (i=sorh[x]; i; i=sed[i].next)
            {
                k = sed[i].to;
                if (--ind[k] == 0)
                    q[rear++] = k;
            }
        }
        clr(res);
        for (i=0; i<2*n; i++)
            if (col[sco[i]] == 1)
                res[i] = 1;
loop:   if(flag)
            printf("NO\n");
        else
        {
            printf("YES\n");
            for (i=0; i<n; i++)
            {
                if (res[i])
                    printf("%02d:%02d%02d:%02d\n",p[i].s1/60,
                            p[i].s1%60,p[i].e1/60,p[i].e1%60);
                else
                    printf("%02d:%02d%02d:%02d\n",p[i].s2/60,
                            p[i].s2%60,p[i].e2/60,p[i].e2%60);
            }
        }
    return 0;
}
```