第 7 章 图论提高

上一个章节中介绍了图论的一些基本知识和基础算法，例如图与树的一些基本概念、图的四种存储方法与两种遍历方法以及最短路径算法、三种最小生成树算法、二分图判定等等，本章节将在此基础上进一步介绍最小生成树、连通分量、二分图、差分约束系统等相关的算法。

7.1 最小生成树

7.1.1 树和生成树

定义无向连通图的最小生成树（Minimum Spanning Tree，MST）为边权和最小的生成树。注意：只有连通图才有生成树，而对于非连通图，只存在生成森林。

定理：

任意一棵最小生成树一定包含无向图中权值最小的边。

证明:

反证法：假设无向图$G=(V,E)^*$ 存在一棵最小生成树不包含权值最小的边。设$e=(x,y,z)$ 是无向图中权值最小的边。把e添加到树中，e会和树上从x到y的路径一起构成一个环，并且环上其他边的权值都比z大。因此用e代替环上的其他任意一条边，会形成一棵权值和更小的生成树，与假设矛盾。故假设不成立，原命题成立。

证毕。

推论：给定一张无向图$G=(V,E)$，$n=|V|$，$m=|E|$。从E中选出$k<n-1$条边构成G的一个生成森林。若再从剩余的$m-k$条边中选$n-1-k$条添加到生成森林中，使其成为G的生成树，并且选出的边的权值之和最小，则该生成树一定包含这$m-k$条边中连接生成森林的两个不连通节点的权值最小的边。

对于求解最小生成树问题，以下介绍三个主要的算法。

7.1.2 Kruskal 算法

Kruskal 算法就是基于上述推论的。Kruskal 算法总是维护无向图的最小生成森林。最初，可认为生成森林由零条边构成，每个节点各自构成一棵仅包含一个点的树。

在学习 Kruskal 算法之前，请读者确保已经理解并查集的原理，并能够熟练应用并查集，并查集是理解整个 Kruskal 算法的核心和关键。

在任意时刻，Kruskal 算法从剩余的边中选出一条权值最小的，并且这条边的两个端点属于生成森林中两棵不同的树(不连通)，把该边加入生成森林。图中节点的连通情况可以用并查集维护。

详细来说，Kruskal 算法的流程如下:

1. 建立并查集，每个点各自构成一个集合。

2. 把所有边按照权值从小到大排序，依次扫描每条边(x,y,z)。

3. 若x,y属于同一集合(连通)，则忽略这条边，继续扫描下一条。

4. 若x,y属于不同集合，合并x,y所在的集合，并把z累加到答案中。

5. 所有边扫描完成后，第 4 步中处理过的边就构成最小生成树。

时间复杂度为$0(m\ log\ m)$。

关键代码：

```
int n, m;
struct node{
   int u;
   int v;
   int w;
}e[maxn];

int fa[maxn], cnt, sum, num;

void add(int x, int y, int w)
{
   e[++ cnt].u = x;
   e[cnt].v = y;
   e[cnt].w = w;
}

bool cmp(node x, node y)
{
   return x.w < y.w;
}

int find(int x)
{
   return fa[x] == x ? fa[x] : fa[x] = find(fa[x]);//路径压缩
}

/*
int find(int x)
```

```
{
  if(fa[x] == x) return x;
  else
  {
                        fa[x] = find(fa[x]);//路径压缩
                        return fa[x];
            }
}
*/

void kruskal()
{
  for(int i = 1; i <= cnt; i ++)
  {
            int x = find(e[i].u);
            int y = find(e[i].v);
            if(x == y) continue;
            fa[x] = y;
            sum += e[i].w;
            if(++ num == n - 1) break;//如果构成了一颗树
  }
}

int main()
{
  cin>>n>>m;
  for(int i = 1; i <= n; i ++) fa[i] = i;
  while(m--)
  {
            int x, y, w;
            cin>>x>>y>>w;
            add(x, y, w);
  }
  std:: sort(e + 1, e + 1 + cnt, cmp);
  kruskal();
  printf("%d",sum);
  return 0;
}
```

例题 局域网（洛谷 P2820）

某个局域网内有 n 台计算机，由于搭建局域网时工作人员的疏忽，现在局域网内的连接形成了回路，我们知道如果局域网形成回路那么数据将不停的在回路内传输，造成网络卡的现象。因为连接计算机的网线本身不同，所以有一些连线不是很畅通，我们用$f(i,j)$

表示i,j之间连接的畅通程度，$f(i,j)$值越小表示i,j之间连接越通畅，$f(i,j)$为0示i,j之间无网线连接。

现在需要解决回路问题，我们将除去一些连线，使得网络中没有回路，不改变原图节点的连通性，并且被除去网线的$\sum f(i,j)$最大，请求出这个最大值。

【输入格式】

第一行两个正整数n,k。

接下来的k行每行三个正整数i,j,m表示i,j两台计算机之间有网线联通，通畅程度为m。

【输出格式】

一个正整数，$\sum f(i,j)$的最大值。

【分析】

题目要求使删去的边最大，等价于使剩下的边长度最小。那么就可以使用最小生成树求解此题，只需要在输入时算出边长总和，再减去最小生成树的长度即可。

很显然是一道典型的最小生成树问题。一般来说，大部分生成树问题都采用Kruskal算法解决。

由于上文已经提供了Kruskal算法的关键代码，在此不提供解题代码。

7.1.3 Prim算法

Prim算法同样基于上述推论，但思路略有改变。Prim算法总是维护最小生成树的一部分。最初，Prim算法仅确定1号节点属于最小生成树。

在任意时刻，设已经确定属于最小生成树的节点集合为T，剩余节点集合为S。Prim算法找到$min_{x\in S,y\in E}\{z\}$，即两个端点分别属于集合S,T的权值最小的边，然后把点x从集合S中删除，加入到集合T，并把z累加到答案中。

具体来说，可以维护数组d;若$x\in S$,则$d[x]$表示节点x与集合T中的节点之间权值最小的边的权值。若$x\in T$,则$d[x]$就等于x被加入T时选出的最小边的权值。

可以类比Dijkstra 算法，用一个数组标记节点是否属于T。每次从未标记的节点中选出d值最小的，把它标记(新加入T)，同时扫描所有出边，更新另一个端点的d值。最后最小生成树的权值总和就是$\sum_{x=2}^{n} d[x]$。

Prim 算法的时间复杂度为$0(n^2)$， 可以用二叉堆优化到$0(mlogn)$。 但用二叉堆优化不如直接使用 Kruskal 算法更加方便。因此，Prim 主要用于稠密图，尤其是完全图的最小生成树的求解。

关键代码：

```
struct edge{
    int v,w,next;
}e[maxm<<1];
int head[maxn],dis[maxn],cnt,n,m,tot,now=1,ans;
bool vis[maxn];

void add(int u,int v,int w){
    e[++cnt].v=v;
    e[cnt].w=w;
    e[cnt].next=head[u];
    head[u]=cnt;
}

void init(){
    cin>>n>>m;
    for(int i=1,u,v,w;i<=m;++i)
    {
        cin>>u>>v>>w;
        add(u,v,w),add(v,u,w);
    }
}

int prim()
{
    for(int i=2;i<=n;++i){
        dis[i]=inf;
    }
    for(int i=head[1];i;i=e[i].next){
        dis[e[i].v]=min(dis[e[i].v],e[i].w);
    }
    while(++tot<n){
        int minn=inf;
        vis[now]=1;
        for(int i=1;i<=n;++i)
        {
            if(!vis[i]&&minn>dis[i])
            {
                minn=dis[i];
```

```
39.                 now=i;
40.             }
41.         }
42.         ans+=minn;
43.
44.         for(int i=head[now];i;i=e[i].next){
45.             int v=e[i].v;
46.             if(dis[v]>e[i].w&&!vis[v]){
47.                 dis[v]=e[i].w;
48.             }
49.         }
50.     }
51.     return ans;
52. }
53.
54. int main(){
55.     init();
56.     printf("%d",prim());
57.     return 0;
58. }
```

7.1.4 Boruvka 算法

Boruvka 算法的思想是前两种算法的结合。它可以用于求解边权互不相同的无向图的最小生成森林。(无向连通图就是最小生成树。)

为了描述该算法，我们需要引入一些定义:

- 定义E'为我们当前找到的最小生成森林的边。在算法执行过程中，我们逐步向E'加边，定义连通块表示一个点集$V' \subseteq V$，且这个点集中的任意两个点u，v在E'中的边构成的子图上是连通的(互相可达)。
- 定义一个连通块的最小边为它连向其它连通块的边中权值最小的那一条。

初始时，$E' = \emptyset$,每个点各自是一个连通块:

1. 计算每个点分别属于哪个连通块。将每个连通块都设为“没有最小边”。

2. 遍历每条边(u, v)，如果u和v不在同一个连通块，就用这条边的边权分别更新u和v所在连通块的最小边。

3. 如果所有连通块都没有最小边，退出程序，此时的E'就是原图最小生成森林的边集。否则,将每个有最小边的连通块的最小边加入E',返回第一步。

当原图连通时，每次迭代连通块数量至少减半，算法只会迭代不超过$O(logV)$次，而原图不连通时相当于多个子问题，因此算法复杂度时$O(ElogV)$的。

例题 【模板】最小生成树（洛谷 P3366）

给出一个无向图，求出最小生成树，如果该图不连通，则输出 orz。

【输入格式】

第一行包含两个整数N, M，表示该图共有 N 个结点和 M 条无向边。

接下来 M 行，每行包含三个整数X_i, Y_i, Z_i，表示有一条长度为Z_i的无向边连接了结点X_i, Y_i。

【输出格式】

如果该图连通，则输出一个整数表示最小生成树的各边的长度之和。如果该图不连通则输出 orz。

【分析】

使用 Boruvka 算法解决最小生成树问题。

关键代码：

```
const int MaxN = 5000 + 5, MaxM = 200000 + 5;
int N, M;
int U[MaxM], V[MaxM], W[MaxM];
bool used[MaxM];
int par[MaxN], Best[MaxN];

void init() {
    scanf("%d %d", &N, &M);
    for (int i = 1; i <= M; ++i)
        scanf("%d %d %d", &U[i], &V[i], &W[i]);
}

void init_dsu() {
    for (int i = 1; i <= N; ++i)
        par[i] = i;
}

int get_par(int x) {
    if (x == par[x]) return x;
    else return par[x] = get_par(par[x]);
}

bool Better(int x, int y) {
    if (y == 0) return true;
    if (W[x] != W[y]) return W[x] < W[y];
    return x < y;
}
```

```
28.
29. void Boruvka() {
30.     init_dsu();
31.
32.     int merged = 0, sum = 0;
33.
34.     bool update = true;
35.     while (update) {
36.         update = false;
37.         memset(Best, 0, sizeof Best);
38.
39.         for (int i = 1; i <= M; ++i) {
40.             if (used[i] == true) continue;
41.             int p = get_par(U[i]), q = get_par(V[i]);
42.             if (p == q) continue;
43.
44.             if (Better(i, Best[p]) == true) Best[p] = i;
45.             if (Better(i, Best[q]) == true) Best[q] = i;
46.         }
47.
48.         for (int i = 1; i <= N; ++i)
49.             if (Best[i] != 0 && used[Best[i]] == false) {
50.                 update = true;
51.                 merged++; sum += W[Best[i]];
52.                 used[Best[i]] = true;
53.                 par[get_par(U[Best[i]])] = get_par(V[Best[i]]);
54.             }
55.     }
56.
57.     if (merged == N - 1) printf("%d\n", sum);
58.     else puts("orz");
59. }
60.
61. int main() {
62.     init();
63.     Boruvka();
64.     return 0;
65. }
```

7.2 点(边)双联通分量

7.2.1 割的概念

对于连通图$G=(V,E)$，若$V'\subseteq V$且$G[V/V']$（即从G中删去V'中的点）不是连通图，则V'是图G的一个点割集。大小为 1 的点割集又被称作割点。

对于连通图$G=(V,E)$和整数k，若$|V|\geq k+1$且G不存在大小为$k-1$的点割集，则称图G是k-点连通的，而使得上式成立的最大的k被称作图G的点连通度。（对于非完全图，点连通度即为最小点割集的大小，而完全图K_n的点连通度为$n-1$。）

对于图$G=(V,E)$以及$u,v\in V$满足$u\neq v$，u和v不相邻，u可达v，若$V'\subseteq V$，$u,v\notin V'$且$G[V/V']$中u和v不连通，则V'被称作u到v的点割集。u到v的最小点割集的大小被称作u到v的局部点连通度。

还可以在边上作类似的定义：

对于连通图$G=(V,E)$，若$E'\subseteq E$且$G'[V,E/E']$（即从G中删去E'中的边）不是连通图，则E'是图G的一个边割集。大小为 1 的边割集又被称作桥。

对于连通图$G=(V,E)$和整数k，若G不存在大小为$k-1$的边割集，则称图G是k-边连通的，而使得上式成立的最大的k被称作图G的边连通度。（对于任何图，边连通度即为最小边割集的大小。）

对于图$G=(V,E)$以及$u,v\in V$满足$u\neq v$，u可达v，若$E'\subseteq E$，且$G[V,E/E']$中u和v不连通，则E'被称作u到v的边割集。u到v的最小边割集的大小被称作u到v的局部边连通度。

点双连通几乎与k-点连通完全一致，除了一条边连接两个点构成的图，它是点双连通的，但不是k-点连通的。换句话说，没有割点的连通图是点双连通的。

边双连通与k-边双连通完全一致。换句话说，没有桥的连通图是边双连通的。

与连通分量类似，也有点双连通分量（极大点双连通子图）和边双连通分量（极大边双连通子图）。

7.2.2 割点和桥

割点：对于一个无向图，如果把一个点删除后这个图的极大连通分量数增加了，那么这个点就是这个图的割点（又称割顶）。

割边（和割点差不多，叫做桥）。

对于一个无向图，如果删掉一条边后图中的连通分量数增加了，则称这条边为桥或者割边。严谨来说，就是：假设有连通图$G=(V,E)$，e是其中一条边（即$e\subseteq E$），如果$G-e$是不连通的，则边e是图G的一条割边（桥）。

7.2.3 Tarjan

对于求割点的方法，暴力的做法如下：

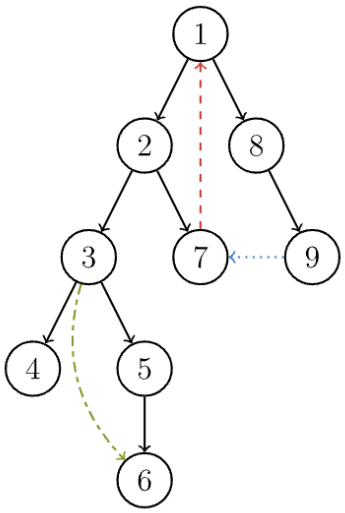
- 依次删除每一个节点v；

- 用 DFS（或 BFS）判断还是否连通；
- 再把节点v加入图中；

若用邻接表，需要做V次 DFS，时间复杂度为$O(V*(V+E))$。

显然暴力的效率极低，所以目前应用最多的便是 Tarjan 算法。Tarjan 算法是一个比较经典的求强连通分量、LCA、割点和割边等多种量的一个算法，应用十分广泛。

因为 tarjan 是以 DFS 为基础的算法，在学习之前，请读者确保能够熟悉并且熟练掌握 DFS 算法。在介绍 tarjan 之前，需要先介绍 DFS 生成树，以下面的有向图为例：



有向图的 DFS 生成树主要有 4 种边（不一定全部出现）：

- 树边：示意图中以黑色边表示，每次搜索找到一个还没有访问过的结点的时候就形成了一条树边。
- 反祖边：示意图中以红色边表示（即$7\to1$），也被叫做回边，即指向祖先结点的边。
- 横叉边：示意图中以蓝色边表示（即$9\to7$），它主要是在搜索的时候遇到了一个已经访问过的结点，但是这个结点并不是当前结点的祖先。
- 前向边：示意图中以绿色边表示（即$3\to6$），它是在搜索的时候遇到子树中的结点的时候形成的。

我们考虑 DFS 生成树与强连通分量之间的关系。

如果结点u是某个强连通分量在搜索树中遇到的第一个结点，那么这个强连通分量的其余结点肯定是在搜索树中以u为根的子树中。结点u被称为这个强连通分量的根。

反证法：假设有个结点v在该强连通分量中但是不在以u为根的子树中，那么u到v的路径中肯定有一条离开子树的边。但是这样的边只可能是横叉边或者反祖边，然而这两条边都要求指向的结点已经被访问过了，这就和u是第一个访问的结点矛盾了。得证。

观察 DFS 搜索树，我们可以发现有两类节点可以成为割点：

对根节点u，若其有两棵或两棵以上的子树，则该根结点u为割点；对非叶子节点u（非根节点），若其子树的节点均没有指向u的祖先节点的回边，说明删除u之后，根结点与u的子树的节点不再连通；则节点u为割点。

用$dfn[u]$记录节点u在 DFS 过程中被遍历到的次序号，$low[u]$记录节点u或u的子树通过非父子边追溯到最早的祖先节点（即 DFS 次序号最小），那么$low[u]$的计算过程如下：

$$\mathrm{low[u]}=\begin{cases}\min\{\mathrm{low[u]},\ \mathrm{low[v]}\},(\mathrm{u},\mathrm{v})\text{为树边}\\ \min\{\mathrm{low[u]},\ \mathrm{dfn[v]}\},(\mathrm{u},\mathrm{v})\text{为回边且 v 不为 u 的父亲节}\end{cases}$$

例题 【模板】最小生成树（洛谷 P3366）

一些学校连入一个电脑网络。那些学校已订立了协议：每个学校都会给其它的一些学校分发软件（称作“接受学校”）。注意即使B在A学校的分发列表中，A也不一定在B学校的列表中。

你要写一个程序计算，根据协议，为了让网络中所有的学校都用上新软件，必须接受新软件副本的最少学校数目（子任务 A）。更进一步，我们想要确定通过给任意一个学校发送新软件，这个软件就会分发到网络中的所有学校。为了完成这个任务，我们可能必须扩展接收学校列表，使其加入新成员。计算最少需要增加几个扩展，使得不论我们给哪个学校发送新软件，它都会到达其余所有的学校（子任务 B）。一个扩展就是在一个学校的接收学校列表中引入一个新成员。

【输入格式】

输入文件的第一行包括一个正整数N，表示网络中的学校数目。学校用前N个正整数标识。

接下来N行中每行都表示一个接收学校列表（分发列表），第$i+1$行包括学校 i 的接收学校的标识符。每个列表用0结束，空列表只用一个 0 表示。

【输出格式】

你的程序应该在输出文件中输出两行。

第一行应该包括一个正整数，表示子任务 A 的解。

第二行应该包括一个非负整数，表示子任务 B 的解。

【分析】

一个比较简单的 tarjan 缩点，因为一个强连通分块里面可以互相到达，那么可以当成一个点处理。

任务 A：需要求最多在多少学校发送新软件，其实就是求缩点后入度为 0 的个数(如果入度不为 0 就可以从其他学校传过来)

任务 B:求入度为 0 的点数与出度为 0 的点的较大值。

【代码】

```
const int N = 1e4 + 10;
int n, head[N], cnt = 0, col = 0, sum;
int top = 1, sta[N], color[N], tim[N], low[N], ins[N];
int in[N], out[N];
struct net
{
    int to, next;
} e[1000000];

void add(int x, int y)
{
    cnt++;
    e[cnt].to = y;
    e[cnt].next = head[x];
    head[x] = cnt;
}

void tarjian(int x)
{
    sum++;
    tim[x] = low[x] = sum;
    sta[top] = x;
    top++;
    ins[x] = 1;
    for (int w = head[x]; w != 0; w = e[w].next)
    {
        if (ins[e[w].to] == 0)
        {
            tarjian(e[w].to);
            low[x] = min(low[x], low[e[w].to]);
        }
        else if (ins[e[w].to] == 1)
            low[x] = min(low[x], tim[e[w].to]);
    }
```

```
35.
36.     if (tim[x] == low[x])
37.     {
38.         col++;
39.         do
40.         {
41.             top--;
42.             color[sta[top]] = col;
43.             ins[sta[top]] = -1;
44.         } while (sta[top] != x);
45.     }
46.     return;
47. }
48.
49. int hym[1000000][3];
50. int main()
51. {
52.     int k = 0;
53.     cin >> n;
54.     for (int i = 1; i <= n; i++)
55.     {
56.         int x;
57.         cin >> x;
58.         while (x != 0)
59.         {
60.             k++;
61.             add(i, x);
62.             hym[k][1] = i;
63.             hym[k][2] = x;
64.             scanf("%d", &x);
65.         }
66.     }
67.     for (int i = 1; i <= n; i++)
68.         if (!ins[i])
69.             tarjian(i);
70.     for (int i = 1; i <= k; i++)
71.         if (color[hym[i][1]] != color[hym[i][2]])
72.         {
73.             out[color[hym[i][1]]]++;
74.             in[color[hym[i][2]]]++;
75.         }
76.     int ans1 = 0, ans2 = 0;
77.     for (int i = 1; i <= col; i++)
78.     {
```

```
        if (in[i] == 0)
            ans1++;
        if (out[i] == 0)
            ans2++;
    }
    if (col == 1)
        cout << 1 << endl
             << 0;
    else
        cout << ans1 << endl
             << max(ans1, ans2);
    return 0;
}
```

例题 【模板】割点（割顶）（洛谷 P3388）

给出一个n个点,m条边的无向图，求图的割点。

【输入格式】

第一行输入两个正整数n,m。

下面m行每行输入两个正整数x,y表示 x 到 y 有一条边。

【输出格式】

第一行输出割点个数。

第二行按照节点编号从小到大输出节点，用空格隔开。

【分析】

选定一个根节点，从该根节点开始遍历整个图（使用 DFS）。

对于根节点，判断是不是割点只需要计算其子树数量，如果有 2 棵及以上的子树，就是割点。因为如果去掉这个点，这两棵子树就不能互相到达。

对于非根节点，维护两个数组$dfn[]$和$low[]$，$dfn[u]$表示顶点u第几个被（首次）访问，$low[u]$表示顶点u及其子树中的点，通过非父子边（回边），能够回溯到的最早的点（dfn最小）的dfn值（但不能通过连接u与其父节点的边）。对于边(u,v)，如果$low[v] >= dfn[u]$，此时u就是割点。

【代码】

```
const int N = 1e6 + 10;
int a[N], nxt[N], head[N], dfn[N], low[N], cnt, k;
bool cut[N], bst[N];

void add(int x, int y)
{
    a[++k] = y;
```

```
    nxt[k] = head[x];
    head[x] = k;
}

void tarjan(int u, int mr)
{
    int rc = 0;
    dfn[u] = low[u] = ++cnt;
    for (int p = head[u]; p; p = nxt[p])
    {
        int v = a[p];
        if (!dfn[v])
        {
            tarjan(v, mr);
            low[u] = min(low[u], low[v]);
            if (low[v] >= dfn[u] && u != mr)
                cut[u] = true;
            if (u == mr)
                rc++;
        }
        low[u] = min(low[u], dfn[v]);
    }
    if (u == mr && rc >= 2)
        cut[mr] = true;
}

int main()
{
    int n, m, ans = 0;
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++)
    {
        int x, y;
        scanf("%d%d", &x, &y);
        add(x, y);
        add(y, x);
    }
    for (int i = 1; i <= n; i++)
    {
        if (!dfn[i])
            tarjan(i, i);
    }
    for (int i = 1; i <= n; i++)
        if (cut[i])
```

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

例题 【模板】最近公共祖先（LCA）（洛谷 P3379）

如题，给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

【输入格式】

第一行包含三个正整数N,M,S，分别表示树的结点个数、询问的个数和树根结点的序号。

接下来$N-1$行每行包含两个正整数x,y，表示x结点和y结点之间有一条直接连接的边（数据保证可以构成树）。

接下来M行每行包含两个正整数a,b，表示询问a结点和b结点的最近公共祖先。

【输出格式】

输出包含M行，每行包含一个正整数，依次为每一个询问的结果。

【分析】

LCA 有许多种算法，常见的如用倍增加速 LCA，RMQ 求 LCA，树链剖分等等。在这里介绍的是 Tarjan 算法，其他解决方法感兴趣的读者可自行探索。

与其他做法不同的是，tarjan 求 LCA 是一个离线的求法。先读入完所有的数据，再用一次 DFS 遍历之后全部输出。

【代码】

```
const int N = 1e6 + 10;
struct edge
{
    int to, next;
} e[N];
int head[N], cnt;
int fa[N];
bool vis[N];
int ans[N];
struct node
{
    int v, id;
};
vector<node> ask[N];
```

```
15. 
16. void add(int u, int v)
17. {
18.     e[++cnt] = (edge){v, head[u]};
19.     head[u] = cnt;
20. }
21. 
22. int get(int x)
23. {
24.     return fa[x] == x ? x : fa[x] = get(fa[x]);
25. }
26. 
27. void dfs(int u, int f)
28. {
29.     for (int i = head[u]; i; i = e[i].next)
30.         if (e[i].to != f)
31.             dfs(e[i].to, u), fa[e[i].to] = u;
32.     int len = ask[u].size();
33.     for (int i = 0; i < len; i++)
34.         if (vis[ask[u][i].v])
35.             ans[ask[u][i].id] = get(ask[u][i].v);
36.     vis[u] = 1;
37. }
38. 
39. int main()
40. {
41.     int n, m, s;
42.     scanf("%d%d%d", &n, &m, &s);
43.     for (int i = 1; i <= 500000; i++)
44.         fa[i] = i;
45.     for (int i = 1; i < n; i++)
46.     {
47.         int u, v;
48.         scanf("%d%d", &u, &v);
49.         add(u, v);
50.         add(v, u);
51.     }
52.     for (int i = 1; i <= m; i++)
53.     {
54.         int u, v;
55.         scanf("%d%d", &u, &v);
56.         ask[u].push_back((node){v, i});
57.         ask[v].push_back((node){u, i});
58.     }
```

```
    dfs(s, 0);
    for (int i = 1; i <= m; i++)
        printf("%d\n", ans[i]);
    return 0;
}

```

7.3 二分图

在上一章节中已经介绍了二分图的基础知识，并提供了二分图的判定方法。在本章节中，将更加深入的了解二分图的应用及相关算法。

7.3.1 图匹配

匹配或是独立边集是一张图中没有公共边的集合。 图匹配算法是信息学竞赛中常用的算法，总体分为最大匹配以及最大权匹配，先从二分图开始介绍，在进一步提出一般图的作法。

图的匹配：

对于一个给定的图$G = (V, E)$，V是点集， E是边集。一组两两不相邻（没有公共顶点）的边集$M \subseteq E$称为这张图的匹配。

如果一个顶点是匹配中的边的顶点，则称这个顶点是被匹配的/匹配点（饱和的）。匹配中的边称为匹配边，反之称为未匹配边。定义匹配的大小为边的数量$| M |$。

- 极大匹配：匹配M不是G任何其他匹配的真子集，换句话说不能再往匹配中加匹配边。
- 最大匹配：边数最多的匹配。
- 最大权匹配：最大匹配中匹配点的数量称为图的配对数，记作$\nu(G)$，当图中的边带权的时候，边权和最大的为最大权匹配。
- 完美匹配：一个包括图中所有顶点的匹配，是最大匹配的一种。
- 近似完美匹配/准完美匹配：只有一个点不是匹配点，扣掉此点以后的图称为近似完美匹配。
- 交错路：始于非匹配点且由匹配边与非匹配边交错而成。
- 增广路：始于非匹配点且终于非匹配点的交错路。

7.3.2 匈牙利算法

二分图的最大匹配问题是二分图中比较基础但是非常重要的问题，二分图的其他问题基本都是在最大匹配的基础上做些改动。

增广路显然拥有以下性质：

1. 长度len是奇数。

2. 路径上第$1,3,\ldots,len$ 条边是非匹配边，第$2,4,6,\ldots,len-1$条边是匹配边。正因为以上性质，如果我们把路径上所有边的状态取反，原来的匹配边变成非匹配的，原来的非匹配边变成匹配的，那么得到的新的边集S'仍然是一组匹配，并且匹配边数增加了 1。进一步可以得到推论:二分图的一组匹配S是最大匹配，当且仅当图中不存在S的增广路。

匈牙利算法又称增广路算法，用于计算二分图最大匹配。它的主要过程为:

1）设$S=0$，即所有边都是非匹配边。

2）寻找增广路$path$，把路径上所有边的匹配状态取反，得到一个更大的匹配s'。

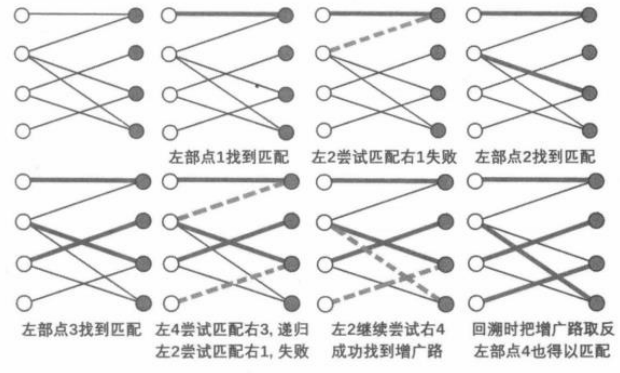
3）重复第 2 步，直至图中不存在增广路。

该算法的关键在于如何找到一条增广路。匈牙利算法依次尝试给每一个左部节点x寻找一个匹配的右部节点y。右部点y能与左部点 x 匹配，需要满足以下两个条件之一:

1. y本身就是非匹配点，此时无向边(x,y) 本身就是非匹配边，自己构成一条长度为 1 的增广路。

2. y已经与左部点x'匹配，但从x出发能找到另一个右部点y' 与之匹配。此时路径$x-y-x'-y'$为一条增广路。

在实际的程序实现中，我们采用深度优先搜索的框架，递归地从x出发寻找增广路。若找到，则在深搜回溯时，正好把路径上的匹配状态取反。另外，可以用全局$bool$数组标记节点的访问情况，避免重复搜索。



匈牙利算法的正确性基于贪心策略，它的一一个重要特点是:当-一个节点成为匹配点后，至多因为找到增广路而更换匹配对象，但是绝不会再变回非匹配点。对于每个左部节点，寻找增广路最多遍历整张二分图-次。因此，该算法的时间复杂度为$O(NM)$。

【代码】

```
1. bool dfs(int u)
2. {
```

```
    for (int i = 0; i < edge[u].size(); ++i)
    {
        int v = edge[u][i];
        if (visit[v])
            continue;
        visit[v] = true;
        if (pb[v] == -1 || dfs(pb[v]))
        {
            pb[v] = u;
            return true;
        }
    }
    return false;
}
```

例题 【模板】二分图最大匹配（洛谷 P3386）

给定一个二分图，其左部点的个数为n，右部点的个数为m，边数为e，求其最大匹配的边数。

左部点从1至 n 编号，右部点从 1 至 m 编号。

【输入格式】

输入的第一行是三个整数，分别代表 n，m 和 e。

接下来 e 行，每行两个整数u,v，表示存在一条连接左部点 u 和右部点 v 的边。

【输出格式】

输出一行一个整数，代表二分图最大匹配的边数。

【代码】

```
const int N = 505;
const int M = 505;

int n, m, res;
vector<int> edge[N];
bool vis[M];
int pb[M];

bool dfs(int u)
{
    for (int i = 0; i < edge[u].size(); ++i)
    {
        int v = edge[u][i];
        if (vis[v])
            continue;
        vis[v] = true;
```

```
        if (pb[v] == -1 || dfs(pb[v]))
        {
            pb[v] = u;
            return true;
        }
    }
    return false;
}

void solve()
{
    res = 0;
    memset(pb, -1, sizeof(pb));
    for (int i = 1; i <= n; ++i)
    {
        memset(vis, false, sizeof(vis));
        if (dfs(i))
        {
            ++res;
        }
    }
}

int main()
{
    int e, u, v;
    scanf("%d%d%d", &n, &m, &e);
    while (e--)
    {
        scanf("%d%d", &u, &v);
        edge[u].push_back(v);
    }
    solve();
    printf("%d\n", res);
    return 0;
}

```

给定一张二分图，二分图的每条边都带有一个权值。求出该二分图的一组最大匹配，使得匹配边的权值总和最大。这个问题称为二分图的带权最大匹配，也称二分图最优匹配。注意，二分图带权最大匹配的前提是匹配数最大，然后再最大化匹配边的权值总和。

二分图带权最大匹配有两种解法:费用流和 KM 算法。本节将针对 KM 算法进行介绍，费用流将在图论进阶中进行讲解。KM 算法程序实现简单，在稠密图上的效率一般高于费用流。不过，KM 算法有很大的局限性，只能在满足“带权最大匹配一定是完备匹配”的图中正确求解。在接下来的讨论中，我们设二分图左、右两部的节点数均为 N。

交错树:

在匈牙利算法中，如果从某个左部节点出发寻找匹配失败，那么在 DFS 的过程中，所有访问过的节点，以及为了访问这些节点而经过的边，共同构成一棵树。

这棵树的根是一个左部节点，所有叶子节点也都是左部节点(因为最终匹配失败),并且树上第 1,3,5,…层的边都是非匹配边，第2,4,6,…层的边 都是匹配边。因此这棵树被称为交错树。

顶标(顶点标记值):

在二分图中,给第$i(1 \le i \le N)$个左部节点一个整数值 A,给第$j(1 \le j \le N)$个右部节点一个整数值B_j。同时， 必须满足$\forall i,j,A_i + B_j \ge w(i,j)$,其中$w(i,j)$表示第$i$个左部点与第$j$个右部点之间的边权(没有边时设为负无穷)。这些整数值A,B;称为节点的顶标。

二分图中所有节点和满足$A_i + B_j \ge w(i,j)$的边构成的子图称为二分图的相等子图。有如下定理：若相等子图中存在完备匹配，则这个完备匹配就是二分图的带权最大匹配，这很容易证明，我们可以直接使用此定理。

KM 算法的基本思想就是，先在满足$\forall i,j,A_i + B_j \ge w(i,j)$的前提下，给每个节点随意赋值一个顶标，然后采取适当的策略不断扩大相等子图的规模，直至相等子图存在完备匹配。例如，我们可以赋值$A_i = max_{1 \le j \le N}\{w(i,j)\}, B_j = 0$。对于一个相等子图，我们用匈牙利算法求它的最大匹配。若最大匹配不完备，则说明一定有一个左部节点匹配失败。该节点匹配失败的那次 DFS 形成了一棵交错树，记为T。

考虑匈牙利算法的流程，容易发现以下两条结论:

1）除了根节点以外，T中其他的左部点都是从右部点沿着匹配边访问到的，即在程序中调用了$dfs(match[y])$, 其中y是一个右部节点。

2）T中所有的右部点都是从左部点沿着非匹配边访问到的。

在寻找到增广路以前，我们不会改变已有的匹配，所以一个右部点沿着匹配边能访问到的左部点是固定的。为了让匹配数增加，我们只能从第 2 条结论入手，考虑“怎样能让左部点沿着非匹配边访问到更多的右部点”。

假如我们把交错树 T 中的所有左部节点顶标$A_i(i \in T)$减小一个整数值A, 把T中的所有右部节点顶标$B_j(j \in T)$增大一个整数值0节点的, 可能发生如下状况：

1）右部点j沿着匹配边，递归访问$i = match[j]$ 的情形。对于一条匹配边，显然要么$i,j \in T$ (被访问到)，要么$i,j \notin T$ (没被访问到)。故$A_i + B_j$不变，匹配边仍然属于相等子图。

2）左部点i沿着非匹配边，访问右部点j，尝试与之匹配的情形。因为左部点的访问是被动的(被右部点沿着匹配边递归)，所以只需考虑$i \in T$。若$i,j \in T$, 则$A_i + B_j$不变。即以前能从i访问到的点j, 现在仍能访问；若$i \in T, j \notin T$, 则$A_i + B_j$减小。即以前从i访问不到的点j，现在有可能访问到。

为了保证项标符合前提条件$\forall i,j, A_i + B_j \geq w(i,j)$, 我们就在所有$i \in T, j \notin T$的边$(i,j)$之中，找出最小的$A_i + B_j - w(i,j)$, 作为Δ的值。只要原图存在完备匹配，这样的边一定存在。上述方法既不会破坏前提条件，又能保证至少有一条新的边会加入相等子图，使交错树中至少一个左部点能访问到的右部点增多。

不断重复以上过程，直到每一个左部点都匹配成功，就得到了相等子图的完备匹配，即原图的带权最大匹配。具体实现时，可以在 DFS 的过程中维护一个数组记录可能更新Δ的值，以便快速求出新的Δ。时间复杂度为$O(N^4)$，随机数据$O(N^3)$。

```
// Data
const ll N = 510;
ll n, m, e[N][N];

// KM
ll mat[N], d[N], va[N], vb[N], ak[N], bk[N];
ll Dfs(ll u)
{
    va[u] = 1;
    for (ll v = 1; v <= n; v++)
        if (!va[v])
        {
            if (ak[u] + bk[v] - e[u][v] == 0)
            {
                vb[v] = 1;
                if (!mat[v] || Dfs(mat[v]))
                    return mat[v] = u, 1;
            }
            else
                d[v] = min(d[v], ak[u] + bk[v] - e[u][v]);
```

```
        }
    return 0;
}

ll KM()
{
    fill(ak + 1, ak + n + 1, -INF);
    for (ll u = 1; u <= n; u++)
        for (ll v = 1; v <= n; v++)
            ak[u] = max(ak[u], e[u][v]);
    for (ll u = 1; u <= n; u++)
    {
        while (true)
        {
            memset(va, 0, sizeof(va));
            memset(vb, 0, sizeof(vb));
            memset(d, 0x3f, sizeof(d));
            if (Dfs(u))
                break;
            ll c = INF;
            for (ll v = 1; v <= n; v++)
                if (!vb[v])
                    c = min(c, d[v]);
            for (ll v = 1; v <= n; v++)
                if (va[v])
                    ak[v] -= c;
            for (ll v = 1; v <= n; v++)
                if (vb[v])
                    bk[v] += c;
        }
    }
    ll res = 0;
    for (ll v = 1; v <= n; v++)
        res += e[mat[v]][v];
    return res;
}

int main()
{
    scanf("%lld%lld", &n, &m);
    for (ll u = 1; u <= n; u++)
        for (ll v = 1; v <= n; v++)
            e[u][v] = -INF;
    for (ll i = 1; i <= m; i++)
```

```
    {
        ll u, v, w;
        scanf("%lld%lld%lld", &u, &v, &w);
        e[u][v] = max(e[u][v], w);
    }
    printf("%lld\n", KM());
    for (ll u = 1; u <= n; u++)
        printf("%lld ", mat[u]);
    puts("");
    return 0;
}

```

例题 【模板】二分图最大权完美匹配（洛谷 P6577）

给定一张二分图，左右部均有n个点，共有 m 条带权边，且保证有完美匹配。求一种完美匹配的方案，使得最终匹配边的边权之和最大。

【输入格式】

第一行两个整数n,m，含义见题目描述。

第$2\sim m+1$ 行，每行三个整数y,c,h 描述了图中的一条从左部的y号结点到右部的c号节点，边权为h的边。

【输出格式】

本题存在 Special Judge。

第一行一个整数ans 表示答案。

第二行共n个整数$a_1,a_2,a_3\cdots a_n$，其中a_i表示完美匹配下与右部第i个点相匹配的左部点的编号。如果存在多种方案，请输出任意一种。

【分析】

上面的算法一般对于二分图来说已经足够用，但将上述代码提交并不能完全通过，实际上，KM 算法还有$O(n^3)$的写法。将 DFS 换成 BFS 即可，本质并无不同。

【代码】

```
const ll N = 505;
const ll inf = 1e18;
ll n, m, map[N][N], matched[N];
ll slack[N], pre[N], ex[N], ey[N];
bool visx[N], visy[N];

void match(ll u)
{
    ll x, y = 0, yy = 0, delta;
```

```
    memset(pre, 0, sizeof(pre));
    for (ll i = 1; i <= n; i++)
        slack[i] = inf;
    matched[y] = u;
    while (1)
    {
        x = matched[y];
        delta = inf;
        visy[y] = 1;
        for (ll i = 1; i <= n; i++)
        {
            if (visy[i])
                continue;
            if (slack[i] > ex[x] + ey[i] - map[x][i])
            {
                slack[i] = ex[x] + ey[i] - map[x][i];
                pre[i] = y;
            }
            if (slack[i] < delta)
            {
                delta = slack[i];
                yy = i;
            }
        }
        for (ll i = 0; i <= n; i++)
        {
            if (visy[i])
                ex[matched[i]] -= delta, ey[i] += delta;
            else
                slack[i] -= delta;
        }
        y = yy;
        if (matched[y] == -1)
            break;
    }
    while (y)
    {
        matched[y] = matched[pre[y]];
        y = pre[y];
    }
}

ll KM()
{
```

```
    memset(matched, -1, sizeof(matched));
    memset(ex, 0, sizeof(ex));
    memset(ey, 0, sizeof(ey));
    for (ll i = 1; i <= n; i++)
    {
        memset(visy, 0, sizeof(visy));
        match(i);
    }
    ll res = 0;
    for (ll i = 1; i <= n; i++)
        if (matched[i] != -1)
            res += map[matched[i]][i];
    return res;
}

int main()
{
    scanf("%lld%lld", &n, &m);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            map[i][j] = -inf;
    for (ll i = 1; i <= m; i++)
    {
        ll u, v, w;
        scanf("%lld%lld%lld", &u, &v, &w);
        map[u][v] = w;
    }
    printf("%lld\n", KM());
    for (ll i = 1; i <= n; i++)
        printf("%lld ", matched[i]);
    printf("\n");
    return 0;
}
```

二分图的问题主要分为：最大匹配、最小点覆盖、最少边覆盖、最大独立集、多重匹配、最大权匹配等，除此之外还有 DAG 的二分图问题，主要包括最少不相交覆盖路径问题和最少可相交覆盖问题，我们已经知晓了最大匹配和最大权匹配的做法，对于其他的问题，有如下几种转换。

1、最小点覆盖数=最大匹配数（点覆盖：点集合使得任意一条边至少有一个端点在集合中）。

2、最少路径覆盖=点数-最大匹配数（路径覆盖：任何一个点都属于且仅属于一条路径）。

3、最大独立集=点数-最大匹配数（独立集：点集合中任何两个顶点都不互相连接）。

以上定理都可证明，这里不详细给出，只需要知晓最大匹配数目，其他的都可以很方便的得到。

7.4 其他

7.4.1 竞赛图

竞赛图是一定义在有向图上的概念，图中每对不同的顶点通过单个有向边连接，即每对顶点间都有一条有向边。

设 D 为 n 阶有向简单图，若 D 的基图为 n 阶无向完全图，则 D 为 n 阶竞赛图。

简单来说，竞赛图就是将完全无向图的无向边给定了方向。

竞赛图有许多性质，比如在哈密顿问题中，对于 n 阶竞赛图，当$n \geq 2$时一定存在哈密顿通路。

将一个竞赛图的每一个点的出度从小到大排序后得到的序列称为竞赛图的比分序列，那么，兰道定理的内容为：

对于一个长度为 n 的序列$S = (s_1 \leq S_2 \leq \cdots \leq S_n), n > 1$是合法的比分序列，当且仅当：$\forall 1 \leq \mathrm{k} \leq \mathrm{n}, \sum_{i=1}^{k} s_i \geq \binom{k}{2}$，且 $k = n$ 时，式子必须要取等。

例题 Football Games（HDU5783）

t 组数据，每组给出 m 个队伍的积分情况，每个队伍之间两两进行比赛，赢的队伍获得 2 分，输的队伍不得分，平局的两个队伍各得 1 分，现在给出每支队伍的得分情况，问是否合法。

【分析】

给出的 m 个队伍可以构成一个竞赛图，问得分情况是否合法实质是在问竞赛图是否合法。

根据竞赛图的兰道定理，将得分视为比分序列，将所有得分进行排序，然后依次处理：由于每次比赛胜利都会使得总分+2，那么前 i 只队伍的得分情况必须大于等于 $i * (i-1)$，当判断到最后一只队伍时，有前 $n-1$ 只队伍的得分必须大于 $n * (n-1)$。

【代码】

```
1. const int N = 1e5 + 10;
2.
3. int a[N];
```

```cpp
int main()
{
    int t;
    while (scanf("%d", &t) != EOF)
    {
        while (t--)
        {
            int n;
            scanf("%d", &n);
            for (int i = 1; i <= n; i++)
                scanf("%d", &a[i]);
            sort(a + 1, a + 1 + n);

            int sum = 0;
            bool flag = true;
            for (int i = 1; i <= n; i++)
            {
                sum += a[i];
                if (i <= n - 1)
                {
                    if (sum < i * (i - 1))
                    {
                        flag = false;
                        break;
                    }
                }
                else
                {
                    if (sum != i * (i - 1))
                    {
                        flag = false;
                        break;
                    }
                }
            }

            if (flag)
                printf("T\n");
            else
                printf("F\n");
        }
    }
    return 0;
}
```

48.

7.4.2 2-SAT

SAT 是适定性（Satisfiability）问题的简称。一般形式为 k -适定性问题，简称 $k-SAT$。而当$k>2$时该问题为 NP 完全的。所以我们只研究$k=2$的情况。

2-SAT 简单的说就是给出n个集合，每个集合有两个元素，已知若干个$<a,b>$，表示a与b矛盾（其中a与b属于不同的集合）。然后从每个集合选择一个元素，判断能否一共选n个两两不矛盾的元素。显然可能有多种选择方案，一般题中只需要求出一种即可。

2-SAT 有着一定的现实意义，比如邀请人来吃喜酒，夫妻二人必须去一个，然而某些人之间有矛盾（比如 A 先生与 B 女士有矛盾，C 女士不想和 D 先生在一起），那么我们要确定能否避免来人之间有矛盾，有时需要方案。这是一类生活中常见的问题。

使用布尔方程表示上述问题。设(a)表示 A 先生去参加，那么 B 女士就不能参加$(\neg a)$；(b)表示 C 女士参加，那么$(\neg b)$也一定成立（D 先生不参加）。总结一下，即$(a \vee b)$（变量a,b至少满足一个）。对这些变量关系建有向图，则有：$\neg a \to b \wedge \neg b \to a$（a不成立则b一定成立；同理，b不成立则a一定成立）。建图之后，我们就可以使用缩点算法来求解 2-SAT 问题了。

较为常用的解决方法为 Tarjan 缩点。

算法考究在建图这点，我们举个例子来讲：

假设有a_1,a_2和b_1,b_2两对，已知a_1和b_2间有矛盾，于是为了方案自洽，由于两者中必须选一个，所以我们就要拉两条有向边(a_1,b_1)和(b_2,a_2)表示选了a_1则必须选b_1，选了b_2则必须选a_2才能够自洽。

然后通过这样子建边我们跑一遍 Tarjan SCC 判断是否有一个集合中的两个元素在同一个 SCC 中，若有则输出不可能，否则输出方案。构造方案只需要把几个不矛盾的 SCC 拼起来。

输出方案时可以通过变量在图中的拓扑序确定该变量的取值。如果变量x的拓扑序在$\neg x$之后，那么取x值为真。应用到 Tarjan 算法的缩点，即x所在 SCC 编号在$\neg x$之前时，取x为真。因为 Tarjan 算法求强连通分量时使用了栈，所以 Tarjan 求得的 SCC 编号相当于反拓扑序。

例题 Party（HDU3062）

有 n 对夫妻被邀请参加一个聚会，因为场地的问题，每对夫妻中只有1人可以列席。在$2n$个人中，某些人之间有着很大的矛盾（当然夫妻之间是没有矛盾的），有矛盾的2个人是不会同时出现在聚会上的。有没有可能会有n个人同时列席？

【分析】

裸的 2-SAT 判断是否有方案，按照我们上面的分析，如果a_1中的丈夫和a_2中的妻子不合，我们就把a_1中的丈夫和a_2中的丈夫连边，把a_2中的妻子和a_1中的妻子连边，然后缩点染色判断即可。

【代码】

```
const int N = 3000;
const int M = 4e6 + 100;
int Index, instack[N], DFN[N], LOW[N];
int tot, color[N];
int numedge, head[N];

struct Edge
{
    int nxt, to;
} edge[M];

int sta[N], top;
int n, m;

void add(int x, int y)
{
    edge[++numedge].to = y;
    edge[numedge].nxt = head[x];
    head[x] = numedge;
}

void tarjan(int x)
{
    sta[++top] = x;
    instack[x] = 1;
    DFN[x] = LOW[x] = ++Index;
    for (int i = head[x]; i; i = edge[i].nxt)
    {
        int v = edge[i].to;
        if (!DFN[v])
        {
            tarjan(v);
            LOW[x] = min(LOW[x], LOW[v]);
```

```
        }
        else if (instack[v])
            LOW[x] = min(LOW[x], DFN[v]);
    }
    if (DFN[x] == LOW[x])
    {
        tot++;
        do
        {
            color[sta[top]] = tot;
            instack[sta[top]] = 0;
        } while (sta[top--] != x);
    }
}

bool solve()
{
    for (int i = 0; i < 2 * n; i++)
        if (!DFN[i])
            tarjan(i);
    for (int i = 0; i < 2 * n; i += 2)
        if (color[i] == color[i + 1])
            return 0;
    return 1;
}

void init()
{
    top = 0;
    tot = 0;
    Index = 0;
    numedge = 0;
    memset(sta, 0, sizeof(sta));
    memset(DFN, 0, sizeof(DFN));
    memset(instack, 0, sizeof(instack));
    memset(LOW, 0, sizeof(LOW));
    memset(color, 0, sizeof(color));
    memset(head, 0, sizeof(head));
}

int main()
{
    while (~scanf("%d%d", &n, &m))
    {
```

```
        init();
        for (int i = 1; i <= m; i++)
        {
            int a1, a2, c1, c2;
            scanf("%d%d%d%d", &a1, &a2, &c1, &c2);
            add(2 * a1 + c1, 2 * a2 + 1 - c2);
            add(2 * a2 + c2, 2 * a1 + 1 - c1);
        }
        if (solve())
            printf("YES\n");
        else
            printf("NO\n");
    }
    return 0;
}

```

7.4.3 差分约束

差分约束系统是求解关于一组变数的特殊不等式组之方法。如果一个系统由n个变量和m个约束条件组成，其中每个约束条件形如$x_j - x_i \le b_k (i,j \in [1,n], k \in [1,m])$,则称其为差分约束系统。亦即，差分约束系统是求解关于一组变量的特殊不等式组的方法。通俗一点地说，差分约束系统就是一些不等式的组，而我们的目标是通过给定的约束不等式组求出最大值或者最小值或者差分约束系统是否有解。例如如下一组不等式：

$$x_1 - x_0 \le 2$$

$$x_2 - x_0 \le 7$$

$$x_3 - x_0 \le 8$$

$$x_2 - x_1 \le 3$$

$$x_3 - x_2 \le 2$$

差分约束系统可以转化为图论来解决，对应于上面的不等式组，如果要求出$x_3 - x_0$的最大值的话，叠加不等式可以推导出$x_3 - x_0 \le 7$,最大值即为 7，我们可以通过建立一个图，包含 6 个顶点，对每个$x_j - x_i \le b_k$，建立一条i到j的有向边，权值为b_k。通过求出这个图的x_0到x_3的最短路可以知道也为 7。

之所以差分约束系统可以通过图论的最短路来解，是因为对于$x_j - x_i \le b_k$，可以发现它类似最短路中的三角不等式$d[v] <= d[u] + w[u,v]$，即$d[v] - d[u] <= w[u,v]$。而求取最大值的过程类似于最短路算法中的松弛过程。

三角不等式：

$$B - A \leq c \quad (1)$$

$$C - B \leq a \quad (2)$$

$$C - A \leq b \quad (3)$$

如果要求C-A的最大值，可以知道$\max(C - A) = min(b, a + c)$，而这正对应了C到A的最短路。

因此，对三角不等式加以推广，变量n个，不等式m个，要求$x_n - x_1$的最大值，便就是求取建图后的最短路。同样地，如果要求取差分约束系统中$x_n - x_1$的最小值，便是求取建图后的最长路。最长路可以通过SPFA求出来，只需要改下松弛的方向即可，即

$$if(d[v] < d[u] + dist(u, v))\ d[v] = d[u] + dist(u, v)$$

当然我们可以把图中所有的边权取负，求取最短路，两者是等价的。

在这里需要注意的是，建图后不一定存在最短路/最长路，因为可能存在无限减小/增大的负环/正环，题目一般会对应于不同的输出。判断差分约束系统是否存在解一般判环即可。

差分约束系统的应用很广，大部分都会有一定的背景，我们只需要根据题意构造出差分约束系统，然后再根据题目的要求求解即可。一般题目会有三种情况：

1）求取最短路；

2）求取最长路；

3）判断差分约束系统的解是否存在。

当然这三种也可能会相互结合。

差分约束系统的解法如下：

根据条件把题意通过变量组表达出来得到不等式组，注意要发掘出隐含的不等式，比如说前后两个变量之间隐含的不等式关系。之后进行建图；建好图之后使用SPFA算法或bellman-ford算法求解，不能用dijstra算法，因为一般存在负边，同时需要注意初始化的问题。

根据题目的要求进行不等式组的标准化。

1）如果要求取最小值，那么求出最长路，那么将不等式全部化成$x_i - x_j \geq k$的形式，这样建立$j \to i$的边，权值为k的边，如果不等式组中有$x_i - x_j > k$，因为一般题目都是对整形变量的约束，化为$x_i - x_j \geq k + 1$即可；如果不等式组中有$x_i - x_j = k$，那么可以变为如下两个：$x_i - x_j \geq k$与$x_i - x_j \leq k$,进一步变为$x_i - x_j \geq -k$，建立两条边即可。

2） 如果求取的是最大值，那么求取最短路，将不等式全部化成$x_i - x_j \leq k$的形式，这样建立$j \to i$的边，权值为k的边，如果像上面的两种情况，同样地进行标准化。

3） 如果要判断差分约束系统是否存在解，一般都是判断环，选择求最短路或者最长路求解都可以，只是不等式标准化时候不同，判环可以使用 SPFA 算法，n个点中如果同一个点入队超过n次，那么即存在环。

值得注意的一点是：建立的图可能不联通，我们只需要加入一个超级源点，比如说求取最长路时图不联通的话，我们只需要加入一个点S，对其他的每个点建立一条权值为 0 的边，就可以转化为连通图。然后从 S 点开始进行 SPFA 判环。最短路类似。

例题　种树（luoguP1250）

一条街的一边有几座房子，因为环保原因居民想要在路边种些树。路边的地区被分割成块，并被编号成$1,2,\ldots,n$。每个部分为一个单位尺寸大小并最多可种一棵树。

每个居民都想在门前种些树，并指定了三个号码b,e,t。这三个数表示该居民想在地区b和e之间（包括b和e）种至少t棵树。居民们想种树的各自区域可以交叉。你的任务是求出能满足所有要求的最少的树的数量。

【输入格式】

输入的第一行是一个整数，代表区域的个数n。

输入的第二行是一个整数，代表房子个数h。

第 3 到第 $(h+2)$ 行，每行三个整数，第 $(i+2)$ 行的整数依次为b_i, e_i, t_i，代表第i个居民想在b_i和e_i之间种至少t_i棵树。

【输出格式】

输出一行一个整数，代表最少的树木个数。

【分析】

很容易可以列出不等式组，设置超级源点，建图跑最短路即可。

【代码】

```
typedef long long ll;
const int INF = 0x3f3f3f;
const int M = 3e4 + 10;
const int N = 1e5 + 10;
struct node
{
    int next, to, dis;
} e[N];
int cnt, dis[M], vis[M], head[M], n, h;
```

```
10.
11. void add(int u, int v, int w)
12. {
13.     e[++cnt].next = head[u];
14.     e[cnt].to = v;
15.     e[cnt].dis = w;
16.     head[u] = cnt;
17. }
18.
19. void spfa(int s)
20. {
21.     queue<int> q;
22.     for (int i = 1; i <= n; ++i)
23.     {
24.         dis[i] = INF;
25.         vis[i] = 0;
26.     }
27.     q.push(s);
28.     dis[s] = 0;
29.     vis[s] = 1;
30.     while (!q.empty())
31.     {
32.         int u = q.front();
33.         q.pop();
34.         vis[u] = 0;
35.         for (int i = head[u]; i; i = e[i].next)
36.         {
37.             int v = e[i].to;
38.             if (dis[v] > dis[u] + e[i].dis)
39.             {
40.                 dis[v] = dis[u] + e[i].dis;
41.                 if (vis[v] == 0)
42.                 {
43.                     vis[v] = 1;
44.                     q.push(v);
45.                 }
46.             }
47.         }
48.     }
49.     return;
50. }
51.
52. int main()
53. {
```

```
    int x, y, z, s;
    scanf("%d%d", &n, &h);
    s = n + 1;
    for (int i = 0; i <= n; ++i)
    {
        add(s, i, 0);
    }
    for (int i = 1; i <= h; ++i)
    {
        scanf("%d%d%d", &x, &y, &z);
        add(y, x - 1, -z);
    }
    for (int i = 1; i <= n; ++i)
    {
        add(i - 1, i, 1);
        add(i, i - 1, 0);
    }
    spfa(s);
    int minn = INF;
    for (int i = 0; i <= n; ++i)
    {
        minn = min(minn, dis[i]);
    }
    printf("%d\n", dis[n] - minn);
    return 0;
}

```