图论

图的遍历

双向广搜

当题目的解空间过大时，但但只从起点开始搜索的话会导致需要的内存空间过大，以及时间复杂度过大，而同时从起点和终点开始搜索的话则会过滤到一些不必要的解空间，使得复杂度大大降低（需要注意的是，在双向广搜中，一次拓展出来的是一层）

例题：子串变换

题目描述：给定两个字符串和一系列变换的规则（最多六个），问从字符串A到字符串B所需要的最小步数，大于10步的即可认为是无解，输出$NO\ ANSWER$

代码：

```
const int N = 10;

string a[N],b[N];
string A,B;
int n;

int extend(queue<string>& q, unordered_map<string, int>&da,
unordered_map<string, int>& db,
    string a[N], string b[N])
{
    int d = da[q.front()];// 这里就保证了一次扩展出来的是一层
    while(q.size() && da[q.front()] == d){
        auto t = q.front();
        q.pop();

        for(int i=0;i<n;i++){// 枚举规则
            for(int j=0;j<t.size();j++){ // 枚举是否匹配规则
                if(t.substr(j,a[i].size()) == a[i]){
                    string r = t.substr(0,j)+b[i]+t.substr(j+a[i].size());
                    if(db.count(r))//如果在b中已经存在了说明搜到了，直接返回即可
                        return da[t]+db[r]+1;
                    if(da.count(r))
                        continue;
                    da[r] = da[t]+1;
                    q.push(r);
                }
            }
        }
    }

    return 11; // 一直没搜到就返回11
}

int bfs()
{
    if(A == B)
        return 0;
```

```
    queue<string> qa,qb;
    unordered_map<string,int> da,db;

    qa.push(A),qb.push(B);
    da[A]=0,db[B]=0;

    int step = 0;
    // 优先考虑对队列里面剩余元素较小的队列进行扩展。
    while(qa.size() && qb.size()){
        int t;
        if(qa.size()<qb.size())
            t = extend(qa,da,db,a,b);
        else
            t = extend(qb,db,da,b,a);
        if(t<=10)
            return t;
        if(++step == 10)
            return -1;
    }

    return -1;
}

int main()
{
    cin >> A >> B;
    while(cin >> a[n] >> b[n])
        n++;

    int t = bfs();
    if(t==-1)
        puts("NO ANSWER!");
    else
        cout << t << endl;

    return 0;
}
```

传递闭包

$floyd$

是用来求任意两个结点之间的最短路的。

复杂度比较高，但是常数小，容易实现。

适用于任何图，不管有向无向，边权正负，但是最短路必须存在。（不能有个负环）

应用：传递闭包，最小环，恰好经过k条边的最短路

代码：

初始化：

```
    for (int i = 1; i <= n; i ++ ){
        for (int j = 1; j <= n; j ++ ){
            if (i == j)
                d[i][j] = 0;
            else
```

```
                d[i][j] = INF;
        }
    }

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    for (int k = 1; k <= n; k ++ ){
        for (int i = 1; i <= n; i ++ ){
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

bitset优化版本

```
const int N = 2010,M = N*N;

int a[N][N];
bitset<N> dp[N];
int n;


int main()
{
    IOS;
    cin >> n;
    char x;
    rep(i,1,n){
        dp[i][i] = 1;
        rep(j,1,n){
            cin >> x;
            a[i][j] = x-'0';
            if(a[i][j] == 1)
                dp[i][j] = 1;
        }
    }

    int ans = 0;
    rep(k,1,n){
        rep(i,1,n){
            if(dp[i][k]) dp[i] |= dp[k];
        }
    }

    rep(i,1,n)
        ans += dp[i].count();
    cout << ans << endl;

    return 0;
}
```

扩展应用：求最小环，恰好经过 K 条边的最短路

最小环

题目描述：给定一张无向图，求图中一个至少包含 3 个点的环，环上的节点不重复，并且环上的边的长度之和最小。

该问题称为无向图的最小环问题。

你需要输出最小环的方案，若最小环不唯一，输出任意一个均可。

题目思路：我们根据这一条环上其中最大节点的编号，将其分为 n 种，我们思考每一种如何计算，考虑枚举点对 (i,j)，其表示 $i->k->j->i$ 这一条环，那么$i->k$ 和 $k->j$ 就是边的长度，那么 i 到 j 的距离该怎么计算呢，可以发现在第 k 轮循环前，$d[i][j]$ 表示的是经过节点编号不超过 $k-1$，从 i 走到 j 的最小距离，这就是我们所需要的。因此在第 k 轮循环中，先统计 k 的答案，再计算 $floyd$ 即可得到答案。

代码：

```
const int N = 110;

int n,m;
int g[N][N],d[N][N];
int pos[N][N];
int path[N],cnt;

void get_path(int i,int j)
{
    if(pos[i][j] == 0)
        return ;
    int k = pos[i][j];
    get_path(i,k);
    path[cnt++] = k;
    get_path(k,j);
}

int main()
{
    IOS;
    cin >> n >> m;
    rep(i,1,n){
        rep(j,1,n){
            if(i == j)
                g[i][j] = 0;
            else
                g[i][j] = inf;
        }
    }
    rep(i,1,m){
        int a,b,c;
        cin >> a >> b >> c;
        g[a][b] = g[b][a] = min(g[a][b],c);
    }

    int res = inf;
    memcpy(d,g,sizeof d);
    for(int k=1;k<=n;k++){
        for(int i=1;i<k;i++){
            for(int j=i+1;j<k;j++){
                if((ll)d[i][j]+g[i][k]+g[k][j] < res){
```

```
                    res = d[i][j]+g[i][k]+g[k][j];
                    cnt = 0;
                    path[cnt++] = k;
                    path[cnt++] = i;
                    get_path(i,j);
                    path[cnt++] = j;
                }
            }
        }

        for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++){
                if(d[i][j] > d[i][k]+d[k][j]){
                    d[i][j] = d[i][k] + d[k][j];
                    pos[i][j] = k;
                }
            }
        }
    }

    if(res == inf)
        puts("No solution.");
    else{
        for(int i=0;i<cnt;i++)
            cout << path[i] << " ";
        cout << endl;
    }

    return 0;
}
```

恰好经过 K 条边的最短路

题目描述：给定一张由 T 条边构成的无向图，点的编号为 $1 \sim 1000$ 之间的整数。

求从起点 S 到终点 E 恰好经过 N 条边（可以重复经过）的最短路

思路：线代与离散数学相关知识

本质上是计算图的邻接矩阵的 k 次幂，k 可能过大，利用快速幂加速。

我们仍构造这个图的邻接矩阵 G，$G[i,j]$ 表示从 i 到 j 的边权。如果 i,j 两点之间没有边，那么 $G[i,j]=\infty$。（有重边的情况取边权的最小值）

显然上述矩阵对应 $k=1$ 时问题的答案。我们仍假设我们知道 k 的答案，记为矩阵 L_k。现在我们想求 $k+1$ 的答案。显然有转移方程

$$L_{k+1}[i,j]=\min_{1\le p\le n}\{L_k[i,p]+G[p,j]\}$$

事实上我们可以类比矩阵乘法，你发现上述转移只是把矩阵乘法的乘积求和变成相加取最小值，于是我们定义这个运算为 $\odot$，即

$$A\odot B=C\iff C[i,j]=\min_{1\le p\le n}\{A[i,p]+B[p,j]\}$$

于是得到

$$L_{k+1}=L_k\odot G$$

展开递推式得到

$$L_k=\underbrace{G\odot\ldots\odot G}_{k\text{次}}=G^{\odot k}$$

我们仍然可以用矩阵快速幂的方法计算上式，因为它显然是具有结合律的。时间复杂度 $O(n^3\log k)$。

代码：

```
const int N = 210;

int k,n,m,S,E;
int ans[N][N];
int g[N][N];

void mul(int c[][N],int a[][N],int b[][N])
{
    static int temp[N][N];
    memset(temp,0x3f,sizeof temp);
    for(int k=1;k<=n;k++){
        for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++){
                temp[i][j] = min(temp[i][j],a[i][k]+b[k][j]);
            }
        }
    }

    memcpy(c,temp,sizeof temp);
}

void qmi()
{
    memset(ans,0x3f,sizeof ans);
    for(int i=1;i<=n;i++)
        ans[i][i] = 0;
    while(k){
        if(k&1)
            mul(ans,ans,g);
        mul(g,g,g);
        k >>= 1;
    }
```

```
}

int main()
{

    cin >> k >> m >> S >> E;
    memset(g,0x3f,sizeof g);
    map<int,int> ids;
    if(!ids.count(S))
        ids[S] = ++n;
    if(!ids.count(E))
        ids[E] = ++n;
    S = ids[S];
    E = ids[E];

    while(m--){
        int a,b,c;
        cin >> c >> a >> b;
        if(!ids.count(a))
            ids[a] = ++n;
        if(!ids.count(b))
            ids[b] = ++n;
        a = ids[a];
        b = ids[b];
        g[a][b] = g[b][a] = min(g[a][b],c);
    }

    qmi();

    cout << ans[S][E] << endl;

    return 0;
}
```

最短路

$bellman-ford$

朴素版 $O(nm)$

过程：对于边(u,v)，松弛操作对应下面的式子:$dis(v)=min(dis(v),dis(u)+w(u,v))$。

这么做的含义是显然的：我们尝试用$S->u->v$其中$(S->u)$的路径取最短路)这条路径去更新点u最短路的长度，如果这条路径更优，就进行更新。

$Bellman-Ford$算法所做的，就是不断尝试对图上每一条边进行松弛。我们每进行一轮循环，就对图上所有的边都尝试进行一次松弛操作，当一次循环中没有成功的松弛操作时，算法停止。

每次循环是的$O(m)$，那么最多会循环多少次呢?

在最短路存在的情况下，由于一次松弛操作会使最短路的边数至少$+1$，而最短路的边数最多为$(n-1)$，因此整个算法最多执行$(n-1)$轮松弛操作。故总时间复杂度为$O(nm)$。

但还有一种情况，如果从点S出发，抵达一个负环时，松弛操作会无休止地进行下去。注意到前面的论证中已经说明了，对于最短路存在的图，松弛操作最多只会执行$(n-1)$轮，因此如果第n轮循环时仍然存在能松弛的边，说明从S点出发，能够抵达一个负环。

例题：有边数限制的最短路

```
struct edge{
    int a,b,w;
}edges[N];

int d[510],back[510]; // 备份数组 用的是上一轮的距离来更新 防止因本轮距离被更新影响下轮
int n,m,k;

int bellman_ford()
{
    memset(d,0x3f,sizeof d);
    d[1] = 0;

    for(int i=0;i<k;i++){ // 迭代k次就是不超过k次的边
        memcpy(back,d,sizeof d); // 复制上一轮
        for(int j=0;j<m;j++){
            int a=edges[j].a,b=edges[j].b,w=edges[j].w;
            if(d[b]>back[a]+w)
                d[b] = back[a] + w;
        }
    }

    /*
    有可能是5和4都不连通，但4和5之间存在一条负权边在更新的时候会更新5的距离
    */
    if(d[n]>inf/2)
        return inf; // 标记不可达到
    else
        return d[n];
}
```

$spfa$ （队列优化）平均$O(n)$,最坏$O(nm)$

思路：在松弛操作$dis(v) = min(dis(v), dis(u) + w(u, v))$中只有$dis(u)$发生变化了的话$dis(v)$才会被更新，因此维护一个队列，每次更新时如果他的邻边变小了的话就将邻边的点入队并标记，已经被标记过的不需要重复入队，当从队列里面拿出来的时候就去除标记。

代码：

```
int e[N],ne[N],w[N],h[N],idx;
int d[N];
bool st[N];
int n,m;

void add(int a,int b,int c)
{
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

int spfa()
{
    memset(d,0x3f,sizeof d);
    d[1] = 0;
```

```
    queue<int> q;
    q.push(1);
    st[1] = true;

    while(q.size()){
        int t = q.front();
        q.pop();
        st[t] = false;

        for(int i=h[t];i!=-1;i=ne[i]){
            int j=e[i];
            if(d[j]>d[t]+w[i]){
                d[j] = d[t] + w[i];
                if(!st[j]){
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return d[n]; // 如果d[n]==0x3f3f3f3f说明没有被更新也就说明没有连通 不可达
}
```

应用：判断负环(队列版)

原理：1 - n的最短路最多中间经过$n-1$个点(不包含起点) 如果发现中间经过$n-1$以上的点的话就说明存在负权回路（本质上是统计最短路所经过的边数）。

缺点：当比较大的$spfa$会tle 可以选择一个$trick$的做法 当$spfa$更新好多次(例如)$5*N$次仍在更新时，我们可以认为这个图里面存在负环

代码：

```
int h[N],e[N],w[N],ne[N],idx;
int d[N],cnt[N]; // cnt数组记录所经历过的点
int n,m;
bool st[N];

void add(int a,int b,int c)
{
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

int spfa()
{

    queue<int> q;
    // 图可能不连通 需要建立虚拟源点 其到所有点的距离为0 相当于将所有结点入队 如果存在负环的话距离会一直缩小 因此事先将d数组设置为0的话不会导致错误
    for(int i=1;i<=n;i++){
        q.push(i);
        st[i] = true;
    }
```

```
    while(q.size()){
        int t = q.front();
        q.pop();
        st[t] = false;

        for(int i=h[t];i!=-1;i=ne[i]){
            int j = e[i];
            if(d[j] > d[t] + w[i]){
                d[j] = d[t] + w[i];
                cnt[j] = cnt[t]+1;
                if(cnt[j]>=n)
                    return true;
                if(!st[j]){
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

另一种$trick$办法，将$spfa$中的队列换为栈（从经验上说表现不错，只在判断负环时这么使用，在寻找最短路时还是要用队列）

代码：STL中的栈复杂度较大，建议手写

```
const int N = 700, M = 100010;

int n;
int h[N], e[M], w[M], ne[M], idx;
double dist[N];
int q[N], cnt[N];
bool st[N];

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
}

bool check(double mid)
{
    memset(st, 0, sizeof st);
    memset(cnt, 0, sizeof cnt);

    int hh = 0, tt = 0;
    for (int i = 0; i < 676; i ++ )
    {
        q[tt ++ ] = i;
        st[i] = true;
    }

    // int count = 0;
    while (hh != tt)
    {
```

```
        int t = q[--tt];
        if (hh == N) hh = 0;
        st[t] = false;

        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (dist[j] < dist[t] + w[i] - mid)
            {
                dist[j] = dist[t] + w[i] - mid;
                cnt[j] = cnt[t] + 1;
                // if ( ++ count > 10000) return true; // 经验上的trick
                if (cnt[j] >= N) return true;
                if (!st[j])
                {
                    q[tt ++ ] = j;
                    st[j] = true;
                }
            }
        }
    }

    return false;
}

int main()
{
    char str[1010];
    while (scanf("%d", &n), n)
    {
        memset(h, -1, sizeof h);
        idx = 0;
        for (int i = 0; i < n; i ++ )
        {
            scanf("%s", str);
            int len = strlen(str);
            if (len >= 2)
            {
                int left = (str[0] - 'a') * 26 + str[1] - 'a';
                int right = (str[len - 2] - 'a') * 26 + str[len - 1] - 'a';
                add(left, right, len);
            }
        }

        if (!check(0)) puts("No solution");
        else
        {
            double l = 0, r = 1000;
            while (r - l > 1e-4)
            {
                double mid = (l + r) / 2;
                if (check(mid)) l = mid;
                else r = mid;
            }

            printf("%lf\n", r);
        }
    }
```

```
    return 0;
}
```

$dijkstra$

只用于非负权的图

过程：将结点分成两个集合：已确定最短路长度的点集（记为集合S）的和未确定最短路长度的点集（记为T集合）。一开始所有的点都属于T集合。

初始化$d[起点]=0$，其他点的均为INF。

然后重复这些操作：

1. 从T集合中，选取一个最短路长度最小的结点，移到S集合中。
2. 对那些刚刚被加入S集合的结点的所有出边执行松弛操作。

直到集合为空，算法结束。

朴素版 $O(n^2+m)$

暴力枚举T集合中到S的最短长度最小的结点

相关代码：

```
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0; // 1是起点

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f)
        return -1;
    return dist[n];
}
```

堆优化版$O(mlogn)$

思路：用堆来维护到S集合中最短距离的点，每次取出堆顶就可以找到最小的点

代码：

```
int n,m;
int h[N],e[N],ne[N],w[N],idx;
int d[N];
bool st[N];

void add(int a,int b,int c)
{
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

int dijkstra()
{
    memset(d,0x3f,sizeof d);
    d[1] = 0;
    priority_queue<PII,vector<PII>,greater<PII>> heap; // 小根堆
    heap.push({0,1});

    while(heap.size()){
        auto t=heap.top();
        heap.pop();

        int v=t.second,dist=t.first;
        if(st[v])
            continue;
        st[v] = true;
        for(int i=h[v];i!=-1;i=ne[i]){
            int j=e[i];
            if(d[j]>dist+w[i]){
                d[j] = dist+w[i];
                heap.push({d[j],j});
            }
        }
    }
    if(d[n] == inf)
        return -1;
    else
        return d[n];
}
```

记录最短路条数

` 要求最短路计数首先满足条件是不能存在值为0的环，因为存在的话那么被更新的点的条数就为INF了。要把图抽象成一种最短路树（拓扑图）(要求为拓扑图的原因是在得到答案时要求划分的子集此时已经被计算过了，所以要求为拓扑图)。
求最短路的算法有以下几种
BFS 只入队一次，出队一次。可以抽象成拓扑图， 因为它可以保证被更新的点的父节点一定已经是

最短距离了，并且这个点的条数已经被完全更新过了。这个性质是核心性质。

$dijkstra$ 每个点只出队一次。也可以抽象成拓扑图， 同理由于每一个出队的点一定已经是最短距离，并且它出队的时候是队列中距离最小的点，这就代表他的最短距离条数已经被完全更新了，所以构成拓扑性质。

$bellman_ford$算法和$spfa$ 本身不具备拓扑序，因为更新它的点不一定是最短距离，所以会出错。

bfs相关代码：

```
const int inf = 0x3f3f3f3f;

int n,m;
const int N = 1e5+10,M=2*2e5+10;
int d[N],cnt[N];
int h[N],e[M],ne[M],idx;

void add(int a,int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

void bfs()
{
    memset(d,0x3f,sizeof d);
    d[1] = 0;
    cnt[1] = 1;

    queue<int> q;
    q.push(1);

    while(q.size()){
        int t=q.front();
        q.pop();

        for(int i=h[t];~i;i=ne[i]){
            int j=e[i];
            if(d[j]>d[t]+1){
                d[j] = d[t]+1;
                cnt[j] = cnt[t]%100003;
                q.push(j);
            }
            else if(d[j] == d[t]+1){
                cnt[j] = (cnt[j]+cnt[t])%100003;
            }
        }
    }
}

int main()
{
    scanf("%d%d",&n,&m);
    memset(h,-1,sizeof h);
    while(m--){
        int a,b;
        scanf("%d%d",&a,&b);
```

```
        add(a,b);
        add(b,a);
    }

    bfs();
    rep(i,1,n){
        cout << cnt[i] << endl;
    }

    return 0;
}
```

应用：求最短路及次短路的距离和各自的条数

代码：

```
const int inf = 0x3f3f3f3f;

const int N = 1010,M = 20010;

struct node{
    int ver,type,dist;
    bool operator>(const node &w) const{
        return dist>w.dist;
    }
};

int n,m,s,f;
int h[N],e[M],ne[M],w[M],idx;
int d[N][2],cnt[N][2];
bool st[N][2];

void add(int a,int b,int c)
{
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

int dijkstra()
{
    memset(st,0,sizeof st);
    memset(cnt,0,sizeof cnt);
    memset(d,0x3f,sizeof d);
    d[s][0] = 0;
    cnt[s][0] = 1;

    priority_queue<node,vector<node>,greater<node>> heap;
    heap.push({s,0,0});

    while(heap.size()){
        auto t = heap.top();
        heap.pop();
```

```
        int ver=t.ver,type=t.type,distance=t.dist,count=cnt[ver][type];
        if(st[ver][type])
            continue;
        st[ver][type] = true;
        for(int i=h[ver];i!=-1;i=ne[i]){
            int j=e[i];
            if(d[j][0] > distance+w[i]){
                d[j][1] = d[j][0];
                cnt[j][1] = cnt[j][0];
                heap.push({j,1,d[j][1]});
                d[j][0] = distance+w[i];
                cnt[j][0] = count;
                heap.push({j,0,d[j][0]});
            }
            else if(d[j][0] == distance+w[i])
                cnt[j][0] += count;
            else if(d[j][1] > distance + w[i]){
                d[j][1] = distance + w[i];
                cnt[j][1] = count;
                heap.push({j,1,d[j][1]});
            }
            else if(d[j][1] == distance + w[i])
                cnt[j][1] += count;
        }
    }

    int ans = cnt[f][0];
    if(d[f][0] + 1 == d[f][1])
        ans += cnt[f][1];
    return ans;
}

int main()
{
    int T;
    cin >> T;
    while(T--){
        scanf("%d%d",&n,&m);
        memset(h,-1,sizeof h);
        idx = 0;
        while(m--){
            int a,b,c;
            scanf("%d%d%d",&a,&b,&c);
            add(a,b,c);
        }
        scanf("%d%d",&s,&f);
        printf("%d\n",dijkstra());
    }

    return 0;
}
```

最短路径树

最小生成树

*prim*算法

```
const int inf = 0x3f3f3f3f;

const int N = 510;

int n,m;
int g[N][N];
int d[N];
bool st[N];

int prim()
{
    memset(d,0x3f,sizeof d);
    d[1] = 0; // 任何一点都均可

    int res=0;
    for(int i=1;i<=n;i++){
        int t=-1;
        for(int j=1;j<=n;j++){
            if(!st[j] && (t==-1 || d[t]>d[j]))
                t=j;
        }
        if(d[t] == inf) // 说明不连通
            return inf;
        if(i)
            res += d[t];
        st[t] = true;
        rep(j,1,n){
            d[j] = min(d[j],d[t]+g[t][j]);
        }
    }
    return res;
}

int main()
{
    cin >> n >> m;
    memset(g,0x3f,sizeof g);
    while(m--){
        int a,b,c;
        cin >> a >> b >> c;
        g[a][b] = g[b][a] = min(g[a][b],c);
    }

    int t = prim();
    if(t == inf)
        cout << "impossible" << endl;
    else
        cout << t << endl;

    return 0;
```

```
}
```

*boruvka*算法

次小生成树

次小生成树分为严格次小生成树和非严格次小生成树

非严格次小生成树

严格次小生成树

代码：

```
using namespace std;

typedef long long LL;

const int N = 100010, M = 300010, INF = 0x3f3f3f3f;

int n, m;
struct Edge
{
    int a, b, w;
    bool used;
    bool operator< (const Edge &t) const
    {
        return w < t.w;
    }
}edge[M];

int p[N];
int h[N], e[M], w[M], ne[M], idx;
int depth[N], fa[N][17], d1[N][17], d2[N][17];
int q[N];

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
}

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

LL kruskal()
{
    for (int i = 1; i <= n; i ++ ) p[i] = i;
    sort(edge, edge + m);
    LL res = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = find(edge[i].a), b = find(edge[i].b), w = edge[i].w;
```

```
        if (a != b)
        {
            p[a] = b;
            res += w;
            edge[i].used = true;
        }
    }

    return res;
}

void build()
{
    memset(h, -1, sizeof h);
    for (int i = 0; i < m; i ++ )
        if (edge[i].used)
        {
            int a = edge[i].a, b = edge[i].b, w = edge[i].w;
            add(a, b, w), add(b, a, w);
        }
}

void bfs()
{
    memset(depth, 0x3f, sizeof depth);
    depth[0] = 0, depth[1] = 1;
    q[0] = 1;
    int hh = 0, tt = 0;
    while (hh <= tt)
    {
        int t = q[hh ++ ];
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (depth[j] > depth[t] + 1)
            {
                depth[j] = depth[t] + 1;
                q[ ++ tt] = j;
                fa[j][0] = t;
                d1[j][0] = w[i], d2[j][0] = -INF;
                for (int k = 1; k <= 16; k ++ )
                {
                    int anc = fa[j][k - 1];
                    fa[j][k] = fa[anc][k - 1];
                    int distance[4] = {d1[j][k - 1], d2[j][k - 1], d1[anc][k -
1], d2[anc][k - 1]};
                    d1[j][k] = d2[j][k] = -INF;
                    for (int u = 0; u < 4; u ++ )
                    {
                        int d = distance[u];
                        if (d > d1[j][k]) d2[j][k] = d1[j][k], d1[j][k] = d;
                        else if (d != d1[j][k] && d > d2[j][k]) d2[j][k] = d;
                    }
                }
            }
        }
    }
}
```

```
int lca(int a, int b, int w)
{
    static int distance[N * 2];
    int cnt = 0;
    if (depth[a] < depth[b]) swap(a, b);
    for (int k = 16; k >= 0; k -- )
        if (depth[fa[a][k]] >= depth[b])
        {
            distance[cnt ++ ] = d1[a][k];
            distance[cnt ++ ] = d2[a][k];
            a = fa[a][k];
        }
    if (a != b)
    {
        for (int k = 16; k >= 0; k -- )
            if (fa[a][k] != fa[b][k])
            {
                distance[cnt ++ ] = d1[a][k];
                distance[cnt ++ ] = d2[a][k];
                distance[cnt ++ ] = d1[b][k];
                distance[cnt ++ ] = d2[b][k];
                a = fa[a][k], b = fa[b][k];
            }
        distance[cnt ++ ] = d1[a][0];
        distance[cnt ++ ] = d1[b][0];
    }

    int dist1 = -INF, dist2 = -INF;
    for (int i = 0; i < cnt; i ++ )
    {
        int d = distance[i];
        if (d > dist1) dist2 = dist1, dist1 = d;
        else if (d != dist1 && d > dist2) dist2 = d;
    }

    if (w > dist1) return w - dist1;
    if (w > dist2) return w - dist2;
    return INF;
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < m; i ++ )
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        edge[i] = {a, b, c};
    }

    LL sum = kruskal();
    build();
    bfs();

    LL res = 1e18;
    for (int i = 0; i < m; i ++ )
        if (!edge[i].used)
```

```
        {
            int a = edge[i].a, b = edge[i].b, w = edge[i].w;
            res = min(res, sum + lca(a, b, w));
        }
    printf("%lld\n", res);

    return 0;
}
```

点（边）双连通分量

$tarjan$

强连通分量

有向图
连通分量:对于分量中任意两点u,v 必然可以从u走到v 且从v走到u
强连通分量:极大连通分量
有向图 → 有向无环图(DAG)
缩点(将所有连通分量缩成一个点)
缩点举例:

```
   o→o→o→o
     ↑ ↓
   o→o→o→o
```

中间的环缩成一个点

```
o     o
  ↘ ↗
   o
  ↗ ↘
o     o
```

应用:
求最短/最长路 递推

求强连通分量:dfs
1 树枝边(x,y)

```
        o
       / \
      o   o
     /   / \
    o   o   o
```

2 前向边(x,y)

```
        o
       / \
      o   x
     /  ↙ \
    o   y   o
```

3 后向边

```
        o
       / \
      o   y
     /  ↗ \
    o   x   o
```

4横插边(往已经搜过的路径上的点继续深搜)
因为我们是从左往右搜的 所以一般是x左边分支上的点

```
        o
       / \
```

```
      o   o
     /   / \
    y ← x   o
```
如果往x右边边分支上的点搜 则属于树枝边

强连通分量：
情况1：
    x存在后向边指向祖先结点y 直接构成环
```
        o
       / \
      o   y
     /  ↗/\
    o   x   o
```
情况2：
    x存在横插边指向的点y有指向x和y的公共祖先节点及以上的点的边
    再通过根节点往下走到x间接构成环
```
        o
   ↗   / \
  ↗   o   o
 ↗   /   / \
    y ← x   o
```

Tarjan 算法求强连通分量
引入 时间戳(按dfs 回溯的顺序标记)
```
        1
   ↗   / \
  ↗   2   4
 ↗   /   / \
    3 ← 5   6
```
标记上时间后：
dfn[u]dfs遍历到u的时间(如上图中的数字)
low[u]从u开始走所能遍历到的最小时间戳(上图中1,2,3,4,5都是一个环/强连通分量中的
                                        即dfn[1]=low[1]=low[2]=low[3]=low[4]=low[5])
    --即u如果在强连通分量,其所指向的层数最高的点

u是其所在的强连通分量的最高点 (上图中dfn[1]=low[1] dfn[6]=low[6])
```
           <=>
     dfn[u] == low[u]
```

树枝边(x,y) 中dfn[y]>dfn[x] low[u]>dfn[u]
前向边(x,y) 中dfn[y]>dfn[x] low[u]>dfn[u]
后向边(x,y) 中dfn[x]>dfn[y] 后向边的终点dfn[u] == low[u]
横插边(x,y) 中dfn[x]>dfn[y]

缩点
```
for i=1;i<=n;i++
 for i的所有邻点j
   if i和j不在同一scc中:
    加一条新边id[i]→id[j]
```

缩点操作后变成有向无环图
就能做topo排序了(此时连通分量编号id[]递减的顺序就是topo序了)
因为我们++scc_cnt是在dfs完节点i的子节点j后才判断low[u]==dfn[u]后才加的
        那么子节点j如果是强连通分量 scc_idx[j]一定小于scc_idx[i]

本题
当一个强连通的出度为0,则该强连通分量中的所有点都被其他强连通分量的牛欢迎
但假如存在两及以上个出度=0的牛(强连通分量) 则必然有一头牛(强连通分量)不被所有牛欢迎

```
见下图最右边两个强连通分量
o→o→o
  ↑
  o→o
*/
```

相关结论：在一个 DAG 图中，要是改图变成一个强连通分量只需加上 $max(p,q)$ 个边即可，其中 p 表示的是起点的数量，而 q 表示的是终点的数量。

```
const int N = 110,M = 10010;

int n;
int h[N],e[M],ne[M],idx;
int dfn[N],low[N],timestamp;
int stk[N],top;
bool in_stk[N];
int id[N],scc_cnt;
int din[N],dout[N];

void tarjan(int u)
{
    dfn[u] = low[u] = ++timestamp;
    stk[++top]=u,in_stk[u]=true;

    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j);
            low[u] = min(low[u],low[j]);
        }
        else if(in_stk[j])
            low[u] = min(low[u],dfn[j]);
    }

    if(dfn[u] == low[u]){
        ++scc_cnt;
        int y;
        do{
            y=stk[top--];
            in_stk[y] = false;
            id[y]=scc_cnt;
        }while(y != u);
    }
}
```

点双连通分量

相关定义：

若一张无向连通图不存在割点，则称它为“**点双连通图**”。

无向图的**极大**点双连通子图被称为“**点双连通分量**”（$verter\ Double\ Connected\ Components\ vDCC$）。

在点双连通分量里面，对割点要裂点（即割点所连的连通分量里面必须含有割点，同时在缩点时，割点自己也要缩点）。

$Tarjan$ 算法求 $vDCC$

用一个栈存点，若遍历回到 x 时，发现割点判定法则 $low[y] >= dfn[x]$ 成立，则从栈中弹出节点，直到 y 被弹出为止。刚才所弹出的点与 x 构成一个 $vDCC$。

$vDCC->$ 缩点

将所有的点双连通分量都缩成点，把缩点和对应的割点连边，构成一个树（或森林）。然后观察树，构造答案。

相关代码：

```
const int N = 1010;

int n,m,root;
int h[N],e[N],ne[N],idx;
int dfn[N],low[N],tot;
int stk[N],top;
int dcc_cnt;
vector<int> dcc[N];
bool cut[N];

void add(int a,int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
}

void tarjan(int u)
{
    dfn[u]=low[u]=++tot;
    stk[++top] = u;

    if(u==root && h[u] == -1){
        dcc_cnt++;
        dcc[dcc_cnt].pb(u);
        return ;
    }

    int child=0;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j);
            low[u] = min(low[u],low[j]);
            if(low[j]>=dfn[u]){
                child++;
                if(u != root || child>1)
                    cut[u] = true;
                ++dcc_cnt;
                int y;
                do{
                    y = stk[top--];
                    dcc[dcc_cnt].pb(y);
                }while(y != j);
                dcc[dcc_cnt].pb(u);
            }
```

```
        }
        else
            low[u] = min(low[u],dfn[j]);
    }
}

int main()
{
    int T=1;
    while(cin >> m,m){
        for(int i=1;i<=dcc_cnt;i++)
            dcc[i].clear();
        memset(h,-1,sizeof h);
        memset(dfn,0,sizeof dfn);
        memset(cut,0,sizeof cut);
        idx = n = tot = top = dcc_cnt = 0;

        while(m--){
            int a,b;
            cin >> a >> b;
            n = max(n,a);
            n = max(n,b);
            add(a,b);
            add(b,a);
        }

        for(root=1;root<=n;root++){
            if(!dfn[root])
                tarjan(root);
        }

        // 要是需要缩点的话 则在这里对每个点进行编号然后连边即可。

        int res = 0;
        ull num = 1;

        for(int i=1;i<=dcc_cnt;i++){
            int cnt=0;
            for(int j=0;j<dcc[i].size();j++){
                if(cut[dcc[i][j]])
                    cnt++;
            }

            if(cnt == 0){
                if(dcc[i].size() > 1){
                    res += 2;
                    num *= dcc[i].size()*(dcc[i].size()-1)/2;
                }
                else
                    res++;
            }
            else if(cnt == 1){
                res++;
                num *= dcc[i].size()-1;
            }
        }
        printf("Case %d: %d %llu\n",T++,res,num);
    }
```

```
    return 0;
}
```

边双连通分量

定义：无向图中极大的不包含割边的连通块被称为“边双连通分量”（$edge\ Double\ Connected\ Components, eDCC$）。

$Tarjan$ 算法求 $eDCC$

```
无向图
双连通分量
1 边的双连通分量   e-dcc   极大的不包含桥的连通块
2 点的双连通分量   v-dcc   极大的不包含割点的连通块

删除后不连通 的 两个定义

桥
 o-o 桥o-o
 | | ↓ | |
 o-o - o-o
1 如何找桥？ x
             /
            y
x和y之间是桥 <=> dfn[x] < low[y] y无论如何往上走不到x
                 +y能到的最高的点low[y] = dfn[y]

2 如何找所有边的双连通分量？
2.1 将所有桥删掉
2.2 stack
dfn[x] == low[x]
<=> x无论如何走不到x的上面
<=> 从x开始的子树可以用一个栈存

每个割点至少属于两个连通分量

树里的每条边都是桥
     o
    / \
   o   o
  /\   /\
 o  o o  o

1 边双连通分量
tarjan回顾
dfn[x] dfs序时间戳
low[x] x能达到的时间戳最小的点

无向图不存在横插边
     o
    / \
   o   o
  /   / \
 y ← x   o
   →
```

```
x能往左的话 由于无向边 所以y也能往右，
那么在之前dfs的时候就把x先于x的父节点加进来了


新建一条道路 使得每两个草场之间都有一个分离的路径
给定一个无向连通图，问最少加几条边，可以将其变为一个边双连通分量

结论：一个边的双连通分量 <=> 任何两个点之间至少存在两个不相交路径
充分性：对于每两个点都有互相分离的路径的话，则必然为强连通分量
反证 假设有桥(非双连通) x,y必然经过中间的桥 则只x→y的路径必在桥上相交
 o-x 桥y-o
 | | ↓ | |
 o-o - o-o
必要性：图是一个边双连通分量 <=> 不包含桥
         则一定对任意两点x,y
         x,y之间至少存在两条互相分离(不相交)的路径
反证:假设存在两条相交路径
      那么x→y中间必然有桥
//  o-o-o-o-o 蓝色路径 (从x出发到y经过边数最少的路径)
//   - - - -  绿色路径
//  x       y

对双连通分量做完缩点后 只剩桥和点
          o
         / \
        o   o
       /\   /\
      o  o o  o
     / \
    o   o

          o
         / \
        o   o
       /\   /\
      o  o-o  o
     / \   |  |
    o   o__|  |
    |_________|
    可以发现对左右两个叶子节点连通后，根节点连向左右叶子节点的边就可以删去了
    同理 再把第2个和第4个叶子节点连通后，根节点连向第2个和第4个叶子节点的边也可以删去
    第3个叶子节点随便连
  给叶子节点按对称性加上边后就没有桥 <=> 变成边的双连通分量
  这里cnt= 5 加了ans=(cnt+1)/2=3条
  ans >= 下界[cnt/2]取整 == [(cnt+1)/2]取整
         其中 cnt为缩完点后度数==1的点的个数
```

相关结论：

给定一个无向图，在该无向图上加入一些边，使其变为一个边双连通分量，求最少需要加多少边，思路是先对原图进行缩点，在一个边的连通分量里面加边是没有意义的，只需加 $(cnt+1)/2$ 条边即可，其中 cnt 表示的是缩完点后叶子节点的数量。

相关代码：

```
const int N = 5010,M = 20010;
```

```cpp
int n,m;
int h[N],e[M],ne[M],idx;
int dfn[N],low[N],tot;
int stk[N],top;
int id[N],dcc_cnt;
bool is_bridge[M];
int d[N];

void add(int a,int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
}

void tarjan(int u,int edg)
{
    dfn[u]=low[u]=++tot;
    stk[++top]=u;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j,i);
            low[u]=min(low[u],low[j]);
            if(dfn[u] < low[j])
                is_bridge[i] = is_bridge[i^1] = true;
        }
        else if(i != (edg^1))
            low[u] = min(low[u],dfn[j]);
    }

    if(dfn[u] == low[u]){
        ++dcc_cnt;
        int y;
        do{
            y = stk[top--];
            id[y] = dcc_cnt;
        }while(y != u);
    }
}

int main()
{
    IOS;
    cin >> n >> m;
    memset(h,-1,sizeof h);
    while(m--){
        int a,b;
        cin >> a >> b;
        add(a,b);
        add(b,a);
    }

    tarjan(1,-1);

    for(int i=0;i<idx;i++){
        if(is_bridge[i])
```

```
            d[id[e[i]]]++;
    }

    int cnt=0;
    for(int i=1;i<=dcc_cnt;i++){
        if(d[i] == 1)
            cnt++;
    }

    cout << (cnt+1)/2 << endl;

    return 0;
}
```

割点、割边

割点

定义：对于一个无项图，如果把一个点删除后，连通块的个数增加了，那么这个点就是割点（又称割顶）。

割点判定法则：

如果 x 不是根节点，当搜索树上存在 x 的一个子节点 y，满足 $low[y] >= dfn[x]$，那么 x 就是割点。

如果 x 是根节点，当搜索树上存在至少两个子节点 $y1, y2$，满足上述条件，那么 x 就是割点。

相关证明：

$low[y] >= dfn[x]$，说明从 y 出发，在不通过 x 点的前提下，不管走哪条边，都无法到达比 x 更早访问的节点，故删除 x 点后，以 y 为根的子树 $subtree(y)$ 也就断开了。即环顶的点割得掉。

反之，若 $low[y] < dfn[x]$，则说明 y 能绕行其他边到达比 x 更早访问的节点，x 就不是割点了，即环内的点割不掉。

相关代码：

```
int n,m,root;
int h[N],e[M],ne[M],idx;
int dfn[N],low[N],tot;
bool st[N];

void add(int a,int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
}

void tarjan(int u)
{
    dfn[u]=low[u]=++tot;
    int child=0;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j);
```

```
            low[u] = min(low[u],low[j]);
            if(low[j]>=dfn[u]){
                child++;
                if(u!=root || child>1)
                    st[u] = true;
            }
        }
        else
            low[u] = min(low[u],dfn[j]);
    }
}

int main()
{
    IOS;
    cin >> n >> m;
    memset(h,-1,sizeof h);
    while(m--){
        int a,b;
        cin >> a >> b;
        add(a,b);
        add(b,a);
    }

    for(int i=1;i<=n;i++){
        if(!dfn[i]){
            root = i;
            tarjan(i);
        }
    }

    vector<int> ans;
    for(int i=1;i<=n;i++){
        if(st[i])
            ans.pb(i);
    }

    cout << ans.size() << endl;
    for(auto x:ans)
        cout << x << " ";
    cout << endl;

    return 0;
}
```

割边

定义：对于一个无向图，如果删掉一条边后图中的连通块个数增加了，则称这条边为桥或者割边。

割边判定法则：

当搜索树上存在 x 的一个子节点 y，满足 $low[y] > dfn[x]$，则 (x, y) 这条边就是割边。

相关证明：

$low[y] > dfn[x]$，说明从 y 出发，在不经过 (x,y)这条边的前提下，不管走哪条边，都无法到达 x 或更早的访问的节点。故删除 (x,y) 这条边，以 y 为根的子树 $subtree(y)$ 也就断开了，即**环外的边割得断。**

反之，若 $low[y] <= dfn[x]$ ，则说明 y 能绕行其他边到达 x 或 更早访问的节点，(x,y) 就不是割边了。即环内的边割不断。（判断时要判断是不是反边）

相关代码：

```
int n,m;
int h[N],e[M],ne[M],idx;
int dfn[N],low[N],tot,cnt;
PII b[M];

void add(int a,int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
}

void tarjan(int u,int edg)
{
    dfn[u]=low[u]=++tot;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j,i);
            low[u] = min(low[u],low[j]);
            if(low[j] > dfn[u])
                b[++cnt] = {u,j};
        }
        else if(i != (edg^1))
            low[u] = min(low[u],dfn[j]);
    }
}

int main()
{
    cin >> n >> m;

    memset(h,-1,sizeof h);
    while(m--){
        int a,b;
        cin >> a >> b;
        add(a,b);
        add(b,a);
    }

    tarjan(1,-1);

    sort(b+1,b+cnt+1);

    for(int i=1;i<=cnt;i++)
        cout << b[i].fi << " " << b[i].se << endl;

    return 0;
```

```
}
```

圆方树

$kosaraju$

二分图

二分图判定

染色法

染色法

将所有点分成两个集合，使得所有边只出现在集合之间，而集合内部不会出现边，这就是二分图
二分图：

一定不含有奇数环，可能包含长度为偶数的环， 不一定是连通图

dfs版本

代码思路：

染色可以使用 1 和 2 区分不同颜色，用 0 表示未染色

遍历所有点，每次将未染色的点进行dfs,默认染成1或者2,接着遍历临边，将其颜色染成另一个，接着dfs临点，如果在临边中出现已经被遍历过的点且其颜色与当前点颜色相同，则染色失败，说明不是二分图。

染色失败相当于存在相邻的2个点染了相同的颜色

代码：

```
const int N = 1e5+10,M=N*2;

int n,m;
int color[N];
int h[N],e[M],ne[M],idx;

void add(int a,int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

bool dfs(int u,int c)
{
    color[u] = c;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(!color[j]){
            if(!dfs(j,3-c))
                return false; // 在dfs里面已经赋值了，因此不需要赋值，直接dfs就行
        }
        else if(color[j] == c)
            return false;
```

```
    }

    return true;
}

int main()
{
    cin >> n >> m;
    memset(h,-1,sizeof h);
    while(m--){
        int a,b;
        cin >> a >> b;
        add(a,b);
        add(b,a);
    }

    rep(i,1,n){
        if(!color[i]){
            if(!dfs(i,1)){
                cout << "No" << endl;
                return 0;
            }
        }
    }

    cout << "Yes" << endl;

    return 0;
}
```

匈牙利算法

一些概念：

匹配：在图论中，一个「匹配」是一个边的集合，其中任意两条边都没有公共顶点。

最大匹配：一个图所有匹配中，所含匹配边数最多的匹配，称为这个图的最大匹配。

最小点覆盖：在一个图中，选择最少的顶点使得至少每条边其中的一个顶点被选中。

最大独立集：选出一些顶点使得这些顶点两两不相邻，则这些点构成的集合称为独立集。找出一个包含顶点数最多的独立集称为最大独立集。

最小路径点覆盖：用最少的互不相交的路径 将所有点覆盖

最小路径重复点覆盖：用最少的可以相交的路径 将所有点覆盖

完美匹配：如果一个图的某个匹配中，所有的顶点都是匹配点，那么它就是一个完美匹配。

交替路：从一个未匹配点出发，依次经过非匹配边、匹配边、非匹配边...形成的路径叫交替路。

增广路：从一个未匹配点出发，走交替路，如果途径另一个未匹配点（出发的点不算），则这条交替 路称为增广路（$agumenting\ path$）。

> **(König定理)**
>
> 一个二分图中的最大匹配数**等于**这个图中的最小点覆盖数。

在二分图中，最大独立集等价于寻找最少的点将边破坏掉，就等于总点数减去最小点覆盖。

应用：求一个二分图中的最大匹配数，最小点覆盖数，最大独立集。

最大匹配数 = 最小点覆盖 = 总点数-最大独立集 = 总点数-最小路径覆盖

```
int n1,n2,m;
int h[N],e[M],ne[M],idx;
int match[N];
bool st[N];

void add(int a,int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

int find(int x)
{
    for(int i=h[x];i!=-1;i=ne[i]){
        int j=e[i];
        if(!st[j]){
            st[j] = true; // 此处标记是为了 find(match[j]) 不能再找回来
            if(match[j] == 0 || find(match[j])){
                match[j] = x;
                return true;
            }
        }
    }

    return false;
}


int main()
{
    scanf("%d%d%d",&n1,&n2,&m);
    memset(h,-1,sizeof h);

    while(m--){
        int a,b;
        scanf("%d%d",&a,&b);
        add(a,b);
    }

    int res = 0;
    for(int i=1;i<=n1;i++){
        memset(st,false,sizeof st); // 每次寻找的时候都要初始化
        if(find(i))
            res++;
    }

    printf("%d\n",res);

    return 0;
}
```

应用二：求一个图的最小路径点覆盖

```
/*
3 最小点覆盖、最大独立集、最小路径点覆盖、最小路径重复点覆盖
  最大匹配数 = 最小点覆盖 = 总点数-最大独立集 = 总点数-最小路径覆盖

最小路径覆盖
DAG(有向无环图)
用最少的互不相交的路径 将所有点覆盖
1
2
...
n
拆成两个点
1    1'
2    2'
...  ...
n    n'
1→2→3
1    1'
  ↘
2    2'
  ↘
3    3'
...  ...
n    n'
出点 入点
原图有边i→j
    则有    i→j'
        出点 入点
原图变为从左边(出点)连向右边(入点)的二分图
则最少互不相交的路径=n-m(最大点覆盖数量)

原图中的每条路径 转化到新图中
每个点最多只有一个出度一个入度
<=> 新图中的任意两条边之间不相交
<=> 新图中的边都是匹配边

每个路径终点 对应 一个左侧非匹配点(3作为终点 在新图中没出边)
<=> 让左侧非匹配点最少 n-m
<=> 让左侧匹配点最多 m
<=> 找最大匹配边数 m
*/

/*
扩展:取消对路径互不相交的约束后
问题-最小路径重复点覆盖
1 先求传递闭包
     <=>
o→o→o => o→o→o
          ↘↗
  G        G'
2 原图G最小路径重复点覆盖==新图G'最小路径覆盖
证:
原图G中多条路径点重复(例如第一条路径和第二条路径都经过A) 把重复点跳过
        →
      ↑   ↓
```

```
o→o→o→o→o→o
  ↓ A ↑ A
    →
对第3到第n条边
如果当前边中点在前面出现过 则跳过

做完后 原图G转化为新图G'

新图G'转化为原图G(直接把外跳边取消)
        →
      ↑   ↓
o→o→o→o→o→o
  ↓ A ↑ A
    →
*/
```

代码:

```
const int N = 210;

int n,m;
bool d[N][N];
bool st[N];
int match[N];


bool find(int x)
{
    for(int i=1;i<=n;i++){
        if(d[x][i] && !st[i]){
            st[i] = true;
            int t=match[i];
            if(t==0 || find(t)){
                match[i] = x;
                return true;
            }
        }
    }

    return false;
}


int main()
{
    IOS;
    cin >> n >> m;
    while(m--){
        int a,b;
        cin >> a >> b;
        d[a][b] = true;
    }

    rep(k,1,n){
        rep(i,1,n){
            rep(j,1,n)
```

```
                d[i][j] |= d[i][k]&d[k][j];
        }
    }

    int res = 0;
    for(int i=1;i<=n;i++){
        memset(st,0,sizeof st);
        if(find(i))
            res++;
    }

    cout << n-res << endl;

    return 0;
}
```

km

代码用处：用来求二分图的最大权完美匹配

代码模板：

```
const ll inf = 1e18;
const ll mod = 1e9+7;
ll gcd(ll a,ll b){return b?gcd(b,a%b):a;}
ll qmi(ll a,ll k,ll p){ll res=1%p;a%=p;while(k)
{if(k&1)res=res*a%p;k>>=1;a=a*a%p;}return res;}

const ll Maxn=505;
ll n,m,w[Maxn][Maxn],matched[Maxn];
ll slack[Maxn],pre[Maxn],ex[Maxn],ey[Maxn];//ex,ey顶标
bool visx[Maxn],visy[Maxn];

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]-w[x][i])
            {
                slack[i]=ex[x]+ey[i]-w[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+=w[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++)
            w[i][j]=-inf;
    for(ll i=1;i<=m;i++)
    {
        ll a,b,c;
        scanf("%lld%lld%lld",&a,&b,&c);
        w[a][b]=max(w[a][b],c);
    }

    printf("%lld\n",KM());
    for(ll i=1;i<=n;i++)
        printf("%lld ",matched[i]);
    printf("\n");

    return 0;
}
```

$hopcraft-karp$

算法用处：求一个二分图的最大匹配 在一定特殊的图上比 $dicnic$ 要快得多。

时间复杂度： $O(m\sqrt{n})$,其中 m 为边数，n 为顶点数，比匈牙利算法更好一些。当图顶点数很多，但是边很少的时候，HK 算法比匈牙利算法好很多

代码模板：

```
template <class T> inline T read()
{
    T x=0,w=1;char ch=0;
    while(ch<'0' || ch>'9'){ if (ch=='-') {w=-1;} ch=getchar();}
    while(ch>='0' && ch<='9'){ x=(x<<1)+(x<<3)+(ch^48);ch=getchar();}
```

```
    return x*w;
}

template <class T> inline void write(T x)
{
    if(x<0){ x=-x; putchar('-');}
    if(x>9) write(x/10);
    putchar(x%10+'0');
}

const int N = 3010;

vector<int> v[N];
int n,m,e,dis;
int match_x[N],match_y[N];
int dis_x[N],dis_y[N];
bool st[N];

bool bfs()
{
    queue<int> q;
    dis = inf;
    memset(dis_x,-1,sizeof dis_x);
    memset(dis_y,-1,sizeof dis_y);
    for(int i=1;i<=n;i++){
        if(match_x[i] == -1){
            q.push(i);
            dis_x[i] = 0;
        }
    }

    while(!q.empty()){
        int t = q.front();
        q.pop();
        if(dis_x[t] > dis)
            break;
        int sz = v[t].size();
        for(int i = 0;i<sz;++i){
            int j = v[t][i];
            if(dis_y[j] == -1) {
                dis_y[j] = dis_x[t] + 1;
                if(match_y[j] == -1)
                    dis = dis_y[j];
                else
                    dis_x[match_y[j]] = dis_y[j] + 1, q.push(match_y[j]);
            }
        }
    }

    return dis != inf;
}

bool dfs(int u)
{
    int sz=v[u].size();
    for(int i=0;i<sz;i++){
        int j=v[u][i];
        if(!st[j] && dis_y[j] == dis_x[u]+1){
```

```
            st[j] = true;
            if(match_y[j] != -1 && dis_y[j] == dis)
                continue;
            if(match_y[j] == -1 || dfs(match_y[j])){
                match_y[j] = u;
                match_x[u] = j;
                return true;
            }
        }
    }
    return false;
}

int hk()
{
    int res=0;
    memset(match_x,-1,sizeof match_x);
    memset(match_y,-1,sizeof match_y);
    while(bfs()){
        memset(st,false,sizeof st);
        for(int i=1;i<=n;i++){
            if(match_x[i] == -1 && dfs(i))
                res++;
        }
    }
    return res;
}

int main()
{
    n = read<int>();
    rep(i,1,n){
        rep(j,1,n){
            int x;
            x = read<int>();
            if(x == 1)
                v[i].pb(j);
        }
    }
    write(hk());

    return 0;
}
```

dinic 求最大匹配

代码：

```
const int N = 210,M = 30010;

int n,m,S,T;
int h[N],e[M],ne[M],f[M],idx;
int q[N],d[N],cur[N];

void add(int a,int b,int c)
{
```

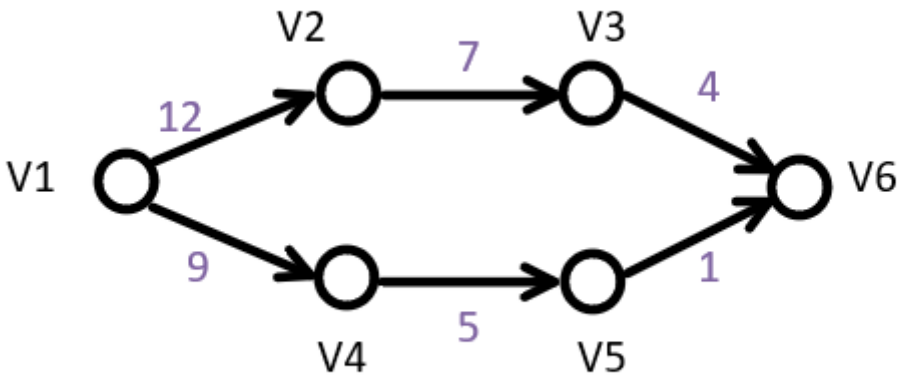
```
    e[idx] = b,f[idx] = c,ne[idx] = h[a],h[a] = idx++;
    e[idx] = a,f[idx] = 0,ne[idx] = h[b],h[b] = idx++;
}

bool bfs()
{
    int hh=0,tt=0;
    memset(d,-1,sizeof d);
    d[S]=0,q[0]=S,cur[S]=h[S];
    while(hh<=tt){
        int t=q[hh++];
        for(int i=h[t];i!=-1;i=ne[i]){
            int ver=e[i];
            if(d[ver]==-1 && f[i]){
                cur[ver] = h[ver];
                d[ver] = d[t]+1;
                if(ver == T)
                    return true;
                q[++tt] = ver;
            }
        }
    }

    return false;
}

int find(int u,int limit)
{
    if(u == T)
        return limit;
    int flow = 0;
    for(int i=cur[u];i!=-1 && flow<limit;i=ne[i]){
        cur[u] = i;
        int ver=e[i];
        if(d[ver] == d[u]+1 && f[i]){
            int t=find(ver,min(f[i],limit-flow));
            if(!t)
                d[ver] = -1;
            f[i] -= t;
            f[i^1] += t;
            flow += t;
        }
    }
    return flow;
}

int dinic()
{
    int r=0,flow=0;
    while(bfs()){
        while(flow = find(S,inf))
            r += flow;
    }
    return r;
}

int main()
{
```

```
    IOS;
    cin >> m >> n;
    S = 0;
    T = n+1;
    memset(h,-1,sizeof h);
    rep(i,1,m)
        add(S,i,1);
    rep(i,m+1,n)
        add(i,T,1);
    int a,b;
    while(cin >> a >> b && !(a == -1 && b == -1)){
        add(a,b,1);
    }

    int t = dinic();
    cout << t << endl;
    for(int i=0;i<idx;i+=2){
        if(e[i]>m && e[i]<=n && !f[i]){
            cout << e[i^1] << " " << e[i] << endl;
        }
    }

    return 0;
}
```

网络流

基本概念：**网络流**（network-flows）是一种类比水流的解决问题方法，与线性规划密切相关。在一个**有向图**中，一个 **源点S** 到一个 **汇点T** 之间有连边，边的权值是该边的最大容量，网络流就是从 S 点到 T 点的一个可行流。
网络流最初的问题就是研究**最大流**，就是指所有可行流里面最大的流。

举个简单的例子，如下图：



若 v_1 到 v_6六个点表示的是六个城市，每条边的权值表示的是城市之间路的长度，选择要运送一批物资从 v_1 到 v_6，要走的路最短这就是最短路问题。
但是如果 v_1 到 v_6 表示的是六个中转站，每条边表示的是水管，权值是水管的最大运输量，现在求从 v_1 运输到 v_6 的最大运输量，这就是最大流问题。
上图的最短路是15（v1-v4-v5-v6），最大流是5。

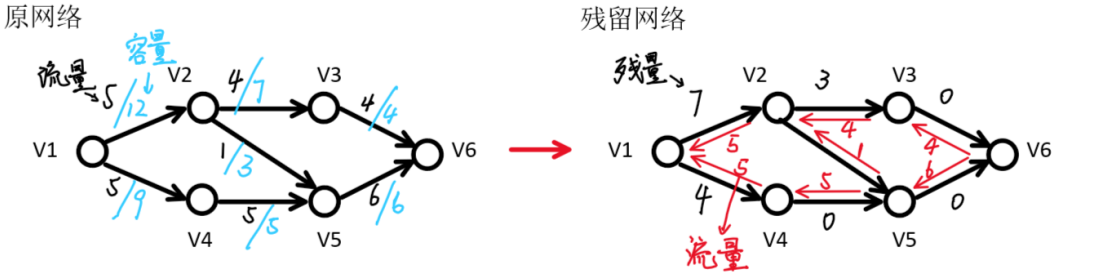
网络流的三个基本性质：

我们先定义C是某条边的最大容量，F是某条边的实际流量，即C(u,v)就是<u,v>这条边的最大容量，F(u,v)就是<u,v>这条边的实际流量。

1、第一个很明显的性质是F(u,v) ≤ C(u,v)，水管的实际流量肯定不会超过最大容量（否则水管就会爆炸...）。
2、第二个性质是对于任意一个节点，其流入的总量一定等于流出的总量，流量守恒，即对于节点x：Σ F(v,x) = Σ F(x,u)。总不能在某个节点蓄水（爆炸危险...）或平白无故的多水（？？？）。当然，对于源点和汇点不用满足流量守恒，我们不关心水从哪来到哪去。
3、第三个不是很明显的性质，斜对称性：F(u,v) = -F(v,u)。但这是完善网络流理论不可缺少的。意思是u向v流了f流量的水，则v向u就流了-f流量的水，就好比a给b给了10块钱，就等于b给a给了-10块钱。

重要定义定理：

首先我们要了解一些定义。
①流量： 一条边上流过的流量。
②容量： 一条边上可供流过的最大流量。
③残量： 容量 - 流量。
④增广路： 从S到T的一条路径，路径上的每条边的残量都为正。
⑤可行流： 每条边的流量任意，但必须满足上面三个基本定理，这样可行的流网络的就称为可行流。
⑥最大流： 最大可行流，顾名思义就是所有可行流中最大的。
⑦残留网络： 残留网络也是个网络，记作f'，其所有点和原网络一样，边是原来的2倍。残留网络包括了原来的所有边，其f'的值是该边的残量（原来就有的正向边），同时残留网络也包含所有边的反向边，f'值是该边正向边的流量。定理：原可行流f+其残留网络f'仍是一个可行流。可以理解残留网络的正向边就是该边还可以加多少流量，反向边就是该边可以减多少流量（一进一退，还可以进的流量就是该边的残量[容量限制]，可以退的流量就是该正向边的流量[最多退到0]）。



最大流（最小割）

EK

最大流最简单的算法是Edmonds-Karp算法，即最短路径增广算法，简称EK算法。EK算法基于一个基本的方法：Ford-Fulkerson方法，即增广路方法，简称FF方法。增广路算法是很多网络流算法的基础，一般都在残留网络中实现。其思路是不断调整流值和残留网络，直到没有增广路为止。FF方法的依据是增广路定理：网络达到最大流当且仅当残留网络中没有增广路。证明略，不过这个定理也不难理解。

那么如何找到一条增广路呢？很简单，我们把所有残量为正的边都看作可行边，然后跑一遍bfs就找到了一条最短的增广路，因为是bfs，所以这是所有增广路里面最小的。当我们找到这条增广路后，所能增广的流值是路径上最小的残留容量边所决定的。

时间复杂度：复杂度较高，为 $O(nm^2)$

代码模板：

```
/*
f数组表示流量
q数组是用来循环的队列
d数组表示这条增广路上流量最小值为多少
```

```
pre数组表示当前的节点是由哪条边转移过来的，之所以用边记录是因为我们要不断更新流量，而如果只记录点的话就无法更新流量。
*/
const int N = 1010,M = 20010;
int n,m,S,T;
int h[N],e[M],ne[M],f[M],idx;
int q[N],d[N],pre[N];
bool st[N];

void add(int a,int b,int c)
{
    e[idx] = b,f[idx] = c,ne[idx] = h[a],h[a] = idx++;
    e[idx] = a,f[idx] = 0,ne[idx] = h[b],h[b] = idx++;
}

bool bfs()
{
    int hh=0,tt=0;
    memset(st,false,sizeof st);
    q[0] = S;
    st[S] = true;
    d[S] = inf;
    while(hh<=tt){
        int t=q[hh++];
        for(int i=h[t];i!=-1;i=ne[i]){
            int ver=e[i];
            if(!st[ver] && f[i]){
                st[ver] = true;
                d[ver] = min(d[t],f[i]);
                pre[ver] = i;
                if(ver == T)
                    return true;
                q[++tt] = ver;
            }
        }
    }

    return false;
}

ll EK()
{
    ll r = 0;
    while(bfs()){
        r += d[T];
        for(int i=T;i!=S;i=e[pre[i]^1]){
            f[pre[i]] -= d[T];
            f[pre[i]^1] += d[T];
        }
    }
    return r;
}

int main()
{
    IOS;
    cin >> n >> m >> S >> T;
    memset(h,-1,sizeof h);
```

```
    while(m--){
        int a,b,c;
        cin >> a >> b >> c;
        add(a,b,c);
    }

    ll ans = EK();
    cout << ans << endl;

    return 0;
}
```

dinic

```
const int N = 10010,M = 2e5+10;

int n,m,S,T;
int h[N],e[M],ne[M],f[M],idx;
int q[N],d[N],cur[N];

void add(int a,int b,int c)
{
    e[idx] = b,f[idx] = c,ne[idx] = h[a],h[a] = idx++;
    e[idx] = a,f[idx] = 0,ne[idx] = h[b],h[b] = idx++;
}

bool bfs()
{
    int hh=0,tt=0;
    memset(d,-1,sizeof d);
    q[0] = S,d[S] = 0,cur[S] = h[S];
    while(hh<=tt){
        int t=q[hh++];
        for(int i=h[t];i!=-1;i=ne[i]){
            int ver=e[i];
            if(d[ver]==-1 && f[i]){
                d[ver]=d[t]+1;
                cur[ver] = h[ver];
                if(ver == T)
                    return true;
                q[++tt] = ver;
            }
        }
    }
    return false;
}

int find(int u,int limit)
{
    if(u == T)
        return limit;
    int flow = 0;
    for(int i=cur[u];i!=-1 && flow<limit;i=ne[i]){
        cur[u] = i;
        int ver=e[i];
        if(d[ver] == d[u]+1 && f[i]){
```

```
            int t=find(ver,min(f[i],limit-flow));
            if(!t)
                d[ver] = -1;
            f[i] -= t;
            f[i^1] += t;
            flow += t;
        }
    }
    return flow;
}

int dinic()
{
    ll r=0,flow=0;
    while(bfs()){
        while(flow = find(S,inf))
            r += flow;
    }
    return r;
}

int main()
{
    IOS;
    cin >> n >> m >> S >> T;
    memset(h,-1,sizeof h);
    while(m--){
        int a,b,c;
        cin >> a >> b >> c;
        add(a,b,c);
    }

    cout << dinic() << endl;

    return 0;
}
```

sap

费用流

最大流

思路：本质上是每次增广一个最小费用路，以单位费用为距离跑最短路，然后对这条最短路上的点增流。

```
const int N = 5010,M = 100010;

int n,m,S,T;
int h[N],e[M],ne[M],w[M],f[M],idx;
int q[N],d[N],pre[N],incf[N];
bool st[N];

void add(int a,int b,int c,int d)
{
    e[idx] = b,f[idx] = c,w[idx] = d,ne[idx] = h[a],h[a] = idx++;
```

```
    e[idx] = a,f[idx] = 0,w[idx] = -d,ne[idx] = h[b],h[b] = idx++;
}

bool spfa()
{
    int hh=0,tt=1;
    memset(d,0x3f,sizeof d);
    memset(incf,0,sizeof incf);
    q[0] = S,d[S] = 0,incf[S] = inf;
    while(hh != tt){
        int t=q[hh++];
        if(hh == N)
            hh = 0;
        st[t] = false;
        for(int i=h[t];i!=-1;i=ne[i]){
            int ver=e[i];
            if(f[i] && d[ver]>d[t]+w[i]){
                d[ver] = d[t]+w[i];
                pre[ver] = i;
                incf[ver] = min(f[i],incf[t]);
                if(!st[ver]){
                    q[tt++] = ver;
                    if(tt == N)
                        tt = 0;
                    st[ver] = true;
                }
            }
        }
    }

    return incf[T]>0;
}

void EK(int &flow,int &cost)
{
    flow = cost = 0;
    while(spfa()){
        int t=incf[T];
        flow += t;
        cost += t*d[T];
        for(int i=T;i!=S;i=e[pre[i]^1]){
            f[pre[i]] -= t;
            f[pre[i]^1] += t;
        }
    }
}

int main()
{
    IOS;
    cin >> n >> m >> S >> T;
    memset(h,-1,sizeof h);
    rep(i,1,m){
        int a,b,c,d;
        cin >> a >> b >> c >> d;
        add(a,b,c,d);
    }
    int flow,cost;
```

```
    EK(flow,cost);
    cout << flow << ' ' << cost << endl;

    return 0;
}
```

可行流

zkw费用流

上下界网络流

本质上是对原网络的调整

无源汇上下界最大网络流

```
const int N = 210,M = (10200+N)*2;

int n,m,S,T;
int h[N],e[M],ne[M],l[M],f[M],idx;
int q[N],d[N],cur[N];
int A[N];

void add(int a,int b,int c,int d)
{
    e[idx] = b,f[idx] = d-c,l[idx] = c,ne[idx] = h[a],h[a] = idx++;
    e[idx] = a,f[idx] = 0,ne[idx] = h[b],h[b] = idx++;
}

bool bfs()
{
    int hh=0,tt=0;
    memset(d,-1,sizeof d);
    d[S]=0,q[0]=S,cur[S]=h[S];
    while(hh<=tt){
        int t=q[hh++];
        for(int i=h[t];i!=-1;i=ne[i]){
            int ver=e[i];
            if(d[ver]==-1 && f[i]){
                cur[ver] = h[ver];
                d[ver] = d[t]+1;
                if(ver == T)
                    return true;
                q[++tt] = ver;
            }
        }
    }

    return false;
}

int find(int u,int limit)
{
    if(u == T)
        return limit;
    int flow = 0;
```

```
    for(int i=cur[u];i!=-1 && flow<limit;i=ne[i]){
        cur[u] = i;
        int ver=e[i];
        if(d[ver] == d[u]+1 && f[i]){
            int t=find(ver,min(f[i],limit-flow));
            if(!t)
                d[ver] = -1;
            f[i] -= t;
            f[i^1] += t;
            flow += t;
        }
    }
    return flow;
}

int dinic()
{
    int r=0,flow=0;
    while(bfs()){
        while(flow = find(S,inf))
            r += flow;
    }
    return r;
}

int main()
{
    IOS;
    cin >> n >> m;
    S = 0;
    T = n+1;
    memset(h,-1,sizeof h);
    rep(i,1,m){
        int a,b,c,d;
        cin >> a >> b >> c >> d;
        add(a,b,c,d);
        A[a] -= c;
        A[b] += c;
    }

    int sum = 0;
    for(int i=1;i<=n;i++){
        if(A[i]>0){
            add(S,i,0,A[i]);
            sum += A[i];
        }
        else if(A[i]<0){
            add(i,T,0,-A[i]);
        }
    }

    if(dinic() != sum)
        cout << "NO" << endl;
    else{
        cout << "YES" << endl;
        for(int i=0;i<m*2;i+=2)
            cout << f[i^1]+l[i] << endl;
    }
```

```
    return 0;
}
```

欧拉回路

欧拉路径与欧拉回路
欧拉路径： 对于图G来说，如果存在一条通路包含G中所有的边，则该通路成为欧拉通路，也称欧拉路径。

欧拉回路： 如果欧拉路径是一条**回路**，那么称其为欧拉回路。

欧拉图 ： 含有欧拉回路的图是欧拉图

半欧拉图：具有欧拉通路但不具有欧拉回路的图。

对无向图G和有向图H：
图G存在欧拉路径与欧拉回路的**充要条件**分别是：
欧拉路径： 图中所有奇度点的数量为0或2。（对于起点来说度应该为奇数因为多出一条边出的，对于终点也是如此，因此要么是**两个**表示起点和终点不是同一个点要么是**0个**表示起点和终点重合）
欧拉回路： 图中所有点的度数都是偶数。

图H存在欧拉路径和欧拉回路的**充要条件**分别是：
欧拉路径： 所有点的入度等于出度 或者 存在一点出度比入度大1(起点)，一点入度比出度大1(终点)，其他点的入度均等于出度。
欧拉回路：所有点的入度等于出度。

同时还要确定所有边在一个连通块内。

代码：

```
const int N = 100010,M = N*4;

int type,n,m;
int h[N],e[M],ne[M],idx;
bool used[M];
int ans[M],cnt;
int din[N],dout[N];

void add(int a,int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
}

void dfs(int u)
{
    for(int &i=h[u];i!=-1;){
        if(used[i]){
            i = ne[i];
            continue;
        }
        used[i] = true;
        if(type == 1)
```

```
            used[i^1] = true;
        int t;
        if(type == 1){
            t = i/2+1;
            if(i&1)
                t = -t;
        }
        else
            t = i+1;
        int j = e[i];
        i=ne[i];
        dfs(j);
        ans[++cnt] = t;
    }
}

signed main()
{
    cin >> type >> n >> m;
    memset(h,-1,sizeof h);

    // type = 1 表示的是无向图 type = 2表示的是有向图
    for(int i=0;i<m;i++){
        int a,b;
        cin >> a >> b;
        add(a,b);
        if(type == 1)
            add(b,a);
        din[b]++,dout[a]++;
    }

    if(type == 1){
        for(int i=1;i<=n;i++){
            if(din[i]+dout[i] & 1){
                puts("NO");
                return 0;
            }
        }
    }
    else{
        for(int i=1;i<=n;i++){
            if(din[i] != dout[i]){
                puts("NO");
                return 0;
            }
        }
    }

    for(int i=1;i<=n;i++){
        if(h[i] != -1){
            dfs(i);
            break;
        }
    }

    if(cnt<m){
        puts("NO");
        return 0;
```

```
    }

    puts("YES");
    for(int i=cnt;i>=1;i--)
        cout << ans[i] << " ";
    cout << endl;

    return 0;
}
```

2-sat

$2-SAT$，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。

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，$a \vee b$ 变量至少满足一个）。对这些变量关系建有向图，则有:$\neg a->b \wedge \neg b->\neg a$($a$ 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。

解法：Tarjan scc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 拼起来就好了。

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。

显然地，时间复杂度为$O(n+m)$。

代码：

注意：用 $2*i$ 表示 命题为 0，用 $2*i+1$ 表示命题 为 1

```
const int N = 2e6+10,M = N;
int n,m;
int h[N],e[M],ne[M],idx;
int dfn[N],low[N],tot;
int stk[N],top;
int id[N],cnt;
bool in_stk[N];

void add(int a,int b)
{
    e[idx] = b;
```

```
    ne[idx] = h[a];
    h[a] = idx++;
}

void tarjan(int u)
{
    dfn[u] = low[u] = ++tot;
    stk[++top] = u;
    in_stk[u] = true;

    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j);
            low[u] = min(low[u],low[j]);
        }
        else if(in_stk[j])
            low[u] = min(low[u],dfn[j]);
    }

    if(low[u] == dfn[u]){
        int y;
        cnt++;
        do{
            y = stk[top--];
            in_stk[y] = false;
            id[y] = cnt;
        }while(y != u);
    }
}

int main()
{
    IOS;
    cin >> n >> m;
    memset(h,-1,sizeof h);
    rep(k,1,m){
        int i,a,j,b;
        cin >> i >> a >> j >> b;
        i--,j--;
        add(2*i+!a,2*j+b);
        add(2*j+!b,2*i+a);
    }

    for(int i=0;i<n*2;i++){
        if(!dfn[i])
            tarjan(i);
    }

    for(int i=0;i<n;i++){
        if(id[i*2] == id[2*i+1]){
            cout << "IMPOSSIBLE" << endl;
            return 0;
        }
    }

    cout << "POSSIBLE" << endl;
    for(int i=0;i<n;i++){
```

```
        if(id[i*2]<id[i*2+1])
            cout << "0 ";
        else
            cout << "1 ";
    }

    return 0;
}
```

竞赛图

差分约束

定义：**差分约束系统** 是一种特殊的n元一次不等式组，它包含n个变量$x_1, x_2, \ldots, x_n$以及m个约束条件，每个约束条件是由两个其中的变量做差构成的，形如$x_i <= x_j + c_k$，其中$1 <= i, j <= n, i \neq j, 1 <= k <= m$并且$c_k$是常数（可以是非负数，也可以是负数）。我们要解决的问题是：求一组解$x_1 = a_1, x_2 = a_2, \ldots, x_n = a_n$，使得所有的约束条件得到满足，否则判断出无解。

源点需要满足的条件：从源点出发，一定可以走到所有的边

差分约束系统中的每个约束条件都可以变形成$x_i <= x_j + c_k$，这与单源最短路中的三角形不等式$d[y] <= d[x] + z$非常相似。因此，我们可以把每个变量x_i看做图中的一个结点，对于每个约束条件$x_i <= x_j + c_k$，从结点 j 向结点 x 连一条长度为c_k的有向边。

注意到，如果$a_1, a_2, \ldots, a_n$是该差分约束系统的一组解，那么对于任意的常数$d, a_1 + d, a_2 + d, \ldots, a_n + d$显然也是该差分约束系统的一组解，因为这样做差后$d$刚好被消掉。

步骤：

[1] 先将每个不等式 $x_i <= x_j + c_k$,转化为一条从x_j走到x_i 长度为c_k的一条边。

[2] 找一个超级源点，使得该源点一定可以遍历到所有边，直接建立虚拟源点，从虚拟源点连接所有点即可，一定可以遍历所有边。

[3] 从源点求一遍最短路

结果1：如果存在负环，则原不等式一定无解

```
3.1 假如存在负环
  x[1]→x[2]→x[3]→x[k]
    ↑ c1   c2   c3 ↓
      ← ← ← ← ← ←
          ck
x[2]≤ x[1]+c[1]
     ...
x[k]≤ x[k-1]+c[k-1]
x[1]≤ x[k]+c[k]
对第一个不等式用后面的不等式一直做松弛
x[2] ≤ x[1]+c[1]
     ≤ x[k]+c[k]+c[1]
     ≤ x[k-1]+c[k-1]+c[k]+c[1]
     ...
     ≤ x[2]+c[2]+...+c[k-1]+c[k]+c[1]
     ≤ x[2]+(小于零的Σc[i])
x[2] < x[2]
即矛盾
```

```
得出结论:不等式无解 <=> 存在负环
```

结果2：如果没有负环，则$d[i]$就是原不等式的一个可行解

```
x[i] ≤ x[j] + C[k]
  x1 ≤ x2+1
{ x2 ≤ x3+2
  x3 ≤ x1-5
 x1 =  0
 x2 = -1
 x3 = -2

类比最短路
  i→j    求之前 d[j] > d[i]+c
   c     求完后 d[j] ≤ d[i]+c
一个图里每个点求完最短距离后每个点的最短距离都有第二个不等式满足
即   任何一个最短路问题 可以 转化为一个差分约束问题
同理   一个差分约束问题 可以 转化为一个单源最短路问题

最长路
   i→j    求之前 d[j] < d[i]+c
   c      求完后 d[j] ≥ d[i]+c
```

代码：

```
/*
题目要求 算最小数量，因此采用的是最长路
*/
const int N = 1e5+10,M = N*2+10;

int h[N],e[M],w[M],ne[M],idx;
int n,m;
int d[N],cnt[N];
bool st[N];

void add(int a,int b,int c)
{
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

bool spfa()
{
    memset(d,-0x3f,sizeof d);
    d[0] = 0;

    queue<int> q;
    q.push(0);
    st[0] = true;

    while(q.size()){
        int t = q.front();
        q.pop();
```

```
        st[t] = false;
        for(int i=h[t];i!=-1;i=ne[i]){
            int j = e[i];
            if(d[j] < d[t]+w[i]){
                d[j] = d[t]+w[i];
                cnt[j] = cnt[t]+1;
                if(cnt[j]>=n+1)
                    return false;
                if(!st[j]){
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return true;
}

int main()
{
    memset(h,-1,sizeof h);
    cin >> n >> m;
    while(m--){
        int x,a,b;
        cin >> x >> a >> b;
        if(x == 1){
            add(b,a,0);
            add(a,b,0);
        }
        else if(x == 2)
            add(a,b,1);
        else if(x == 3)
            add(b,a,0);
        else if(x == 4)
            add(b,a,1);
        else
            add(a,b,0);
    }

    rep(i,1,n)
        add(0,i,1);

    if(!spfa())
        cout << -1 << endl;
    else{
        ll res = 0;
        rep(i,1,n)
            res += d[i];
        cout << res << endl;
    }

    return 0;
}
```

应用二：

```
应用二：
2 如何求最大值或者最小值(x[i] for i in range(1,n))
    结论1:如果求的是最小值,则应该求最长路,如果求的是最大值,则应该求最短路
    问题1:如何转化x[i] ≤ c 其中c是一个常数 这类的不等式
    方法:建立一个超级源点,0号点x[0],然后建立0→i 长度是c的边即可
        x[i] ≤ c
            <=>
        x[i] ≤ x[0] + c = 0 + c

    以求x[i]的最大值为例:所有从x[i]出发,构成的不等式链
    x[i] ≤ x[j] + c[j]
        ≤ x[k] + c[k] + c[j]
        ≤ x[0] + c[1]+ c[2]+... + c[j]
        =   0  + c[1]+ ... + c[j]
    所计算出的上界,
    最终x[i]的最大值
    =所有上界的最小值
    举例 x[i] ≤ 5
        x[i] ≤ 2
        x[i] ≤ 3
        max(x[i]) = min(5,2,3) = 2
    0 → 1 → 3 → 5 → ... → i
      c1  c3  c5       ci-1
    x[1] ≤ x[0] + c[1]
    x[3] ≤ x[1] + c[3]
    x[5] ≤ x[3] + c[5]
    ...
    x[i] ≤ x[i-1] + c[i-1]
    则
    x[i] ≤ x[i-1] + c[i]
        ≤ x[i-3] + c[i-3] + c[i]
        ...
        ≤ x[0] + c[1] + c[3] + c[i-3] + c[i-1]
    ★可以发现Σc[i]就是从0→i的一条路径的长度

    那么
        求x[i]最大值
            <=>
        求所有上界的最小值
            <=>
        求所有从0→i的路径和的最小值
            <=>
        最短路求dist[i]

    同理 求x[i]最小值
            <=>
        求所有下界的最大值
            <=>
        求所有从0→i的路径和的最大值
            <=>
        最长路求dist[i]
```

斯坦纳树

仙人掌

最小树形图

一般图匹配

k最短路

描述：给定一张N个点（编号$1,2\dots N$),M条边的有向图，求从起点 S到终点 T的第K短路的长度，路径允许重复经过点或边。

注意： 每条最短路中至少要包含一条边。

做法：$A*$算法

思路：很显然地，我们有一个暴力的BFS做法：第k次搜到点T的就是所求。然而这样太慢了，我们考虑优化方案。

对于搜索中的一个状态x，令$g(x)$为当前状态下该点到点T的路径长。

朴素的 Bfs 就是直接暴力地拓展，但我们可以设计一种方案，使得 **相对接近终点的状态优先拓展**。因为暴力扩展多个可能会产生无用的信息。

设计估价函数：$f(x)$表示x到终点的最短距离，已经是最优的了。$g(x)$表示到起点的距离，$h(x)=f(x)+g(x)$我们起始时建一个反图，由终点跑向各点，即为该点到终点的最短距离，然后利用堆，每次取出h最小的点进行扩展，记录终点被访问的次数（当终点被访问k次时，即可求得k短路的距离）。

代码：

```
const int N = 1010,M = 2e4+10;

int n,m;
int S,T,k;
int h[N],rh[N],e[M],ne[M],w[M],idx;
int dist[N],cnt[N];
bool st[N];

void add(int h[],int a,int b,int c)
{
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

void dijkstra()
{
    memset(dist,0x3f,sizeof dist);
    dist[T] = 0;

    priority_queue<PII,vector<PII>,greater<PII>> heap;
    heap.push({0,T});

    while(heap.size()){
        auto t = heap.top();
        heap.pop();
```

```
        int ver = t.second;
        if(st[ver])
            continue;
        st[ver] = true;
        for(int i=rh[ver];i!=-1;i=ne[i]){
            int j = e[i];
            if(dist[j]>dist[ver]+w[i]){
                dist[j] = dist[ver]+w[i];
                heap.push({dist[j],j});
            }
        }
    }
}

int astar()
{
    priority_queue<PIII, vector<PIII>, greater<PIII>> heap; // 存储三个值，一个是估价函数，一个是到起点的真实距离，一个是节点
    heap.push({dist[S], {0, S}});

    while(heap.size()){
        auto t=heap.top();
        heap.pop();

        int ver=t.se.se,distance=t.se.fi;
        cnt[ver]++;
        if(cnt[T] == k)
            return distance;
        for(int i=h[ver];i!=-1;i=ne[i]){
            int j=e[i];
            if(cnt[j]<k){
                heap.push({distance+w[i]+dist[j],{distance+w[i],j}});
            }
        }
    }

    return -1;
}

int main()
{
    scanf("%d%d",&n,&m);
    memset(h,-1,sizeof h);
    memset(rh,-1,sizeof rh);

    rep(i,1,m){
        int a,b,c;
        scanf("%d%d%d",&a,&b,&c);
        add(h,a,b,c);
        add(rh,b,a,c);
    }

    scanf("%d%d%d",&S,&T,&k);
    if(S == T)
        k++;

    dijkstra();
```

```
    printf("%d\n",astar());

    return 0;
}
```

支配树

全局最小割

弦图