目录

[单源最短路算法 1](#_Toc119173802)

[多源汇最短路 3](#_Toc119173803)

[最短路计数 5](#_Toc119173804)

[次短路 6](#_Toc119173805)

[K短路 9](#_Toc119173806)

[最短路算法的一些应用 10](#_Toc119173807)

[最小生成树 14](#_Toc119173808)

[次小生成树 15](#_Toc119173809)

[LCA 20](#_Toc119173810)

[LCA应用 23](#_Toc119173811)

[Tarjan求强连通分量 26](#_Toc119173812)

[Tarjan求双连通分量 27](#_Toc119173813)

[对强连通和双连通的总结 31](#_Toc119173814)

[欧拉回路与欧拉路径 31](#_Toc119173815)

[拓扑排序 34](#_Toc119173816)

[关键路径 35](#_Toc119173817)

[二分图 35](#_Toc119173818)

[网络流 38](#_Toc119173819)

[树 56](#_Toc119173820)

[图 63](#_Toc119173821)

### 单源最短路算法

#### dijkstra

##### 朴素版

时间复杂度

int dijkstra()  
{  
 memset(dist,0x3f,sizeof dist);  
 dist[1]=0;  
 for(int i=0;i<n-1;i++)  
 {  
 int t=-1;  
 for(int j=1;j<=n;j++)  
 if(!st[j]&&(t==-1||dist[t]>dist[j]))  
 t=j;  
 st[t]=1;  
 for(int j=1;j<=n;j++)  
 dist[j]=min(dist[j],dist[t]+g[t][j]);  
 }  
 if(dist[n]==0x3f3f3f3f) return -1;  
 return dist[n];  
}

##### 堆优化版

时间复杂度

int dijkstra()  
{  
 memset(dist,0x3f,sizeof dist);  
 dist[1]=0;  
 priority\_queue<PII,vector<PII>,greater<PII>> heap;  
 heap.push({0,1});  
 while(heap.size())  
 {  
 auto t=heap.top();  
 heap.pop();  
 int ver=t.second,distance= t.first;  
 if(st[ver]) continue;  
 st[ver]=1;  
 for(int i=h[ver];i!=-1;i=ne[i])  
 {  
 int j=e[i];  
 if(dist[j]>distance+w[i])  
 {  
 dist[j]=distance+w[i];  
 heap.push({dist[j],j});  
 }  
 }  
 }  
 if(dist[n]==0x3f3f3f3f) return -1;  
 return dist[n];  
}

#### Bellman-Ford

时间复杂度

#include<bits/stdc++.h>  
using namespace std;  
const int N=510,M=1e5+5;  
struct Edge  
{  
 int a,b,c;  
}edges[M];  
int dist[N];  
int pre[N];  
int n,m,k;  
void bellman\_ford()  
{  
 memset(dist,0x3f,sizeof dist);  
 dist[1]=0;  
 for(int i=0;i<k;i++)  
 {  
 memcpy(pre,dist,sizeof dist);  
 for(int j=0;j<m;j++)  
 {  
 auto e=edges[j];  
 dist[e.b]=min(dist[e.b],pre[e.a]+e.c);  
 }  
 }  
}  
int main()  
{  
 scanf("%d%d%d",&n,&m,&k);  
 for(int i=0;i<m;i++)  
 {  
 int a,b,c;  
 scanf("%d%d%d",&a,&b,&c);  
 edges[i]={a,b,c};  
 }  
 bellman\_ford();  
 if(dist[n]>0x3f3f3f3f/2) puts("impossible");  
 else cout<<dist[n]<<endl;  
 return 0;  
}

一般是用作去求有边数限制的最短路。

#### SPFA

时间复杂度最优，最坏会退化为

int spfa()  
{  
 memset(dist,0x3f,sizeof dist);  
 dist[1]=0;  
 queue<int>q;  
 q.push(1);  
 st[1]=1;  
 while(q.size())  
 {  
 int t=q.front();  
 q.pop();  
 st[t]=0;  
 for(int i=h[t];i!=-1;i=ne[i])  
 {  
 int j=e[i];  
 if(dist[j]>dist[t]+w[i]){  
 dist[j]=dist[t]+w[i];  
 if(!st[j])  
 {  
 q.push(j);  
 st[j]=1;  
 }  
 }  
 }  
 }  
 if(dist[n]==0x3f3f3f3f) return -1;  
 else dist[n];  
}

### 多源汇最短路

#### Floyd

时间复杂度

void Floyd()  
{  
 for(int k=1;k<=n;k++)   
 for(int i=1;i<=n;i++)  
 for(int j=1;j<=n;j++)  
 g[i][j]=min(g[i][j],g[i][k]+g[k][j]);  
}

#### Johnson 全源最短路

时间复杂度

直接用了一个洛谷的模板题

[Johnson全源最短路](https://www.luogu.com.cn/problem/P5905)

#include<bits/stdc++.h>  
using namespace std;  
typedef long long LL;  
typedef pair<LL,int> PII;  
const int N = 3e3 + 10,M = N\*3;  
int h[N],e[M],ne[M],w[M],idx;  
bool st[N];  
LL dist[N],d[N],cnt[N];  
int n,m;  
  
void add(int a,int b,int c)  
{  
 e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx++;  
}  
  
bool spfa()  
{  
 queue<int> q;  
 for(int i=1;i<=n;i++)  
 st[i]=1,q.push(i);  
 while(q.size())  
 {  
 auto t = q.front();q.pop();  
 st[t] = 0;  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(dist[j]>dist[t]+w[i])  
 {  
 dist[j] = dist[t] + w[i];  
 cnt[j] = cnt[t] + 1;  
 if(cnt[j]>=n) return 1;  
 if(!st[j])  
 {  
 st[j] = 1;  
 q.push(j);  
 }  
 }  
 }  
 }  
 return 0;  
}  
  
void dijkstra(int s)  
{  
 priority\_queue<PII,vector<PII>,greater<PII>> q;  
 for(int i=1;i<=n;i++) d[i] = 1e9;  
 memset(st,0,sizeof st);  
 d[s] = 0;  
 q.push({0,s});  
 while(q.size())  
 {  
 auto t = q.top();  
 q.pop();  
 int ver = t.second;  
 if(st[ver]) continue;  
 st[ver] = 1;  
 for(int i=h[ver];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(d[j]>d[ver]+w[i])  
 {  
 d[j] = d[ver] + w[i];  
 q.push({d[j],j});  
 }  
 }  
 }  
}  
  
int main()  
{  
 cin>>n>>m;  
 memset(h,-1,sizeof h);  
 for(int i=0;i<m;i++)  
 {  
 int a,b,c;cin>>a>>b>>c;  
 add(a,b,c);  
 }  
 if(spfa())   
 {  
 puts("-1");  
 return 0;  
 }  
 for(int i=1;i<=n;i++){  
 for(int j=h[i];~j;j=ne[j])  
 {  
 int k = e[j];  
 w[j] -= dist[k] - dist[i];  
 }  
 }  
 for(int i=1;i<=n;i++){  
 dijkstra(i);  
 LL res = 0;  
 for(int j=1;j<=n;j++){  
 if(d[j]==1e9) res += 1ll\*j\*d[j];  
 else res += 1ll\*j\*(d[j]+(dist[j]-dist[i]));  
 }  
 cout<<res<<endl;  
 }  
 return 0;  
}

**具体步骤**

1. 先从虚拟源点S向每个点连接一条边权为0的边(不用真的建出来)，再从S做一遍SPFA

* 实际操作，我们只需要将每个点都放入队列，并且最短距离设置为0即可。

1. 在遍历图中的每一条边，若从**i -> j,边权为z**，则将边权变为**z - (dist[j] - dist[i])**
2. 接着从n个顶点分别做一次Dijkstra，我们就可以得到每个顶点到其余顶点的最短距离

* 实际的最短距离需要还原回去

### 最短路计数

#include<bits/stdc++.h>  
using namespace std;  
typedef long long LL;  
typedef pair<int, int> PII;  
const int N = 1e5 + 10,M = 4e5 + 10,mod=100003;  
int h[N],e[M],ne[M],idx;  
int f[N];  
int dist[N];  
bool st[N];  
int n,m;  
  
void add(int a,int b)  
{  
 e[idx]=b,ne[idx]=h[a],h[a]=idx++;  
}  
  
void dijkstra()  
{  
 memset(dist,0x3f,sizeof dist);  
 memset(st, 0, sizeof st);  
 dist[1]=0;  
 f[1]=1;  
 priority\_queue<PII,vector<PII>,greater<PII>> q;  
 q.push({0,1});  
 while(q.size())  
 {  
 auto t = q.top();  
 q.pop();  
 int ver = t.second;  
 if(st[ver]) continue;  
 st[ver]=1;  
 for(int i=h[ver];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(dist[j]>dist[ver]+1)  
 {  
 dist[j]=dist[ver]+1;  
 f[j]=f[ver];  
 q.push({dist[j],j});  
 }  
 else if(dist[j]==dist[ver]+1) f[j]=(LL)(f[j]+f[ver])%mod;  
 }  
 }  
}  
  
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);  
 }  
  
 dijkstra();  
  
 for(int i=1;i<=n;i++)   
 {  
 printf("%d\n",f[i]);  
 }  
 return 0;  
}

本来觉得没有必要，但是最后想了想，其实还是比较有代表意义的。

所有的在图论上的计数问题都可以以此作为延伸。

### 次短路

#### 可重复过点

这个板子，同时有计数操作。

#include<bits/stdc++.h>  
using namespace std;  
const int N = 1100,M = 11000;  
int h[N],e[M],ne[M],w[M],idx;  
int dist[N][2];  
int cnt[N][2];  
bool st[N][2];  
int n,m,S,T;  
struct Ver  
{  
 int id,type,dist;  
 bool operator>(const Ver& W) const  
 {  
 return dist>W.dist;  
 }  
};  
void add(int a,int b,int c)  
{  
 e[idx]=b,ne[idx]=h[a],w[idx]=c,h[a]=idx++;  
}  
int dijkstra()  
{  
 memset(st, 0, sizeof st);  
 memset(dist,0x3f,sizeof dist);  
 memset(cnt,0,sizeof cnt);//多组测试样例的时候一定要记得这些初始化  
 dist[S][0]=0;  
 cnt[S][0]=1;  
 priority\_queue<Ver,vector<Ver>,greater<Ver>> q;  
 q.push({S,0,0});  
 while(q.size())  
 {  
 auto t = q.top();  
 q.pop();  
 int ver = t.id,type = t.type,count=cnt[ver][type];  
 if(st[ver][type]) continue;  
 st[ver][type]=1;  
 for(int i=h[ver];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(dist[j][0]>dist[ver][type]+w[i])//如果可以更新最短路，那原最短路就变为了次短路；  
 {  
 dist[j][1]=dist[j][0],cnt[j][1]=cnt[j][0];  
 q.push({j,1,dist[j][1]});  
 dist[j][0]=dist[ver][type]+w[i],cnt[j][0]=count;  
 q.push({j,0,dist[j][0]});  
 }  
 else if(dist[j][0]==dist[ver][type]+w[i]) cnt[j][0]+=count;//如果到同一个点的最短路相同，则直接累加条数  
 else if(dist[j][1]>dist[ver][type]+w[i])//如果次短路可以被更新，那就更新次短路的条数和长度；  
 {  
 dist[j][1]=dist[ver][type]+w[i],cnt[j][1]=count;  
 q.push({j,1,dist[j][1]});  
 }  
 else if(dist[j][1]==dist[ver][type]+w[i]) cnt[j][1]+=count;//如果到该点的次短路相同，直接累加  
 }  
 }  
 int res = cnt[T][0];  
 if(dist[T][0]+1==dist[T][1]) res += cnt[T][1];  
 return res;  
}  
int main()  
{  
 int t = 0;  
 cin>>t;  
 while(t--)  
 {  
 int n,m;  
 cin>>n>>m;  
 memset(h, -1, sizeof h);  
 idx=0;//多组测试样例上，记住初始化  
 while (m -- )  
 {  
 int a,b,c;  
 scanf("%d%d%d",&a,&b,&c);  
 add(a,b,c);  
 }  
 scanf("%d%d",&S,&T);  
 cout<<dijkstra()<<endl;  
 }  
 return 0;  
}

#### 不可重复过点

#include<bits/stdc++.h>  
using namespace std;  
typedef pair<int,int> PII;  
typedef pair<double,int> PDI;  
const int N = 210,M = N\*N;  
int h[N],e[M],ne[M],idx;  
PII P[N];  
int pre[N];  
double w[M],dist[M];  
bool st[N];  
int n,m;  
  
double get(int a,int b)  
{  
 double dx = P[a].first - P[b].first,dy = P[a].second - P[b].second;  
 return sqrt(dx\*dx+dy\*dy);  
}  
  
void add(int a,int b,double c)  
{  
 e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx++;  
}  
  
double dijkstra(int x,int y)  
{  
 for(int i=1;i<=n;i++) dist[i] = 1e9,st[i] = 0;  
 priority\_queue<PDI,vector<PDI>,greater<PDI>> q;  
 dist[1] = 0;  
 q.push({0,1});  
 while(q.size())  
 {  
 auto t = q.top();  
 q.pop();  
 int ver = t.second;  
 if(st[ver]) continue;  
 st[ver] = 1;  
 for(int i=h[ver];~i;i=ne[i])  
 {  
 int j = e[i];  
 if((ver==x&&j==y)||(ver==y&&j==x)) continue;  
 if(dist[j]>dist[ver]+w[i])  
 {  
 dist[j] = dist[ver] + w[i];  
 if(x==-1&&y==-1) pre[j] = ver;  
 q.push({dist[j],j});  
 }  
 }  
 }  
 return dist[n];  
}  
  
int main()  
{  
 cin>>n>>m;  
 memset(h,-1,sizeof h);  
 for(int i=1;i<=n;i++) cin>>P[i].first>>P[i].second;  
 while(m--)  
 {  
 int a,b;cin>>a>>b;  
 double c = get(a,b);  
 add(a,b,c),add(b,a,c);  
 }  
 dijkstra(-1,-1);  
 double ans = 1e9;  
 int x = n;  
 do  
 {  
 double res = dijkstra(pre[x],x);  
 ans = min(ans,res);  
 x = pre[x];  
 }while(x!=1);  
 if(ans==1e9) puts("-1");  
 else printf("%.2lf",ans);  
 return 0;  
}

### K短路

#include<bits/stdc++.h>  
using namespace std;  
#define x first  
#define y second  
typedef pair<int, int> PII;  
typedef pair<int,pair<int,int>> PIII;  
const int N=1010,M=2e5+5;  
int h[N],rh[N],e[M],ne[M],w[M],idx;  
int dist[N],st[N];  
int S,k,T;  
int n,m;  
void add(int h[],int a,int b,int c)  
{  
 e[idx]=b,w[idx]=c,ne[idx]=h[a],h[a]=idx++;  
}  
void dijstra()  
{  
 priority\_queue<PII,vector<PII>,greater<PII>> heap;  
 memset(dist,0x3f,sizeof dist);  
 heap.push({0,T});  
 dist[T]=0;  
 while(heap.size())  
 {  
 auto t= heap.top();  
 heap.pop();  
  
 int ver= t.y;  
 if(st[ver]) continue;  
 st[ver]=1;  
 for(int i=rh[ver];~i;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 a\_star()  
{  
 priority\_queue<PIII,vector<PIII>,greater<PIII>> heap;  
 memset(st, 0, sizeof st);  
 heap.push({dist[S],{0,S}});  
 while(heap.size())  
 {  
 auto t= heap.top();  
 heap.pop();  
 int ver=t.y.y,distance=t.y.x;  
 st[ver]++;  
 if(st[T]==k) return distance;  
 for(int i=h[ver];~i;i=ne[i])  
 {  
 int j=e[i];  
 if(st[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);  
 for(int i=0;i<m;i++)  
 {  
 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++;  
 dijstra();  
 printf("%d\n",a\_star());  
 return 0;  
}

### 最短路算法的一些应用

#### SPFA求负环

int h[N],ne[M],e[M],w[M],idx;  
int dist[N],cnt[N];  
bool st[N];  
void add(int a, int b, int c) // 添加一条边a->b，边权为c  
{  
 e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;  
}  
bool spfa()  
{  
 memset(st, 0, sizeof st);  
 memset(cnt,0,sizeof cnt);  
 queue<int> q;  
 for(int i = 1;i<=n;i++)  
 {  
 q.push(i);  
 st[i]=1;  
 }  
 while(q.size())  
 {  
 auto t = q.front();  
 q.pop();  
 st[t]=0;  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(dist[j]>dist[t]+w[i])  
 {  
 dist[j]=dist[t]+w[i];  
 cnt[j]=cnt[t]+1;  
 if(cnt[j]>=n) return 1;  
 if(!st[j])  
 {  
 q.push(j);  
 st[j]=1;  
 }  
 }  
 }  
 }  
 return 0;  
}

**注意**

这个模板解决的是**给的图中**是否有负环，若指定开头了，那就不要将所有点都加入队列

**在求负环的算法中，我们加入一些玄学优化吗**

* 将队列改为栈
* 加入Trick，当总点数入队100000次左右时，直接返回false

**但一定要说的是，两种方法都有缺点**

* 队列换栈，可能会增加时间复杂度
* Trick就更好说了，可能真的就是在很大的位置，就跑出来了，不是负环。

**求负环算法，本身不难，但依托于出现了以下两个应用:01分数规划，差分约束**

#### 01分数规划

这个算法，说起来比较抽象，所以我们用一个题目来说明。

给定一张 **L** 个点、**P** 条边的有向图，每个点都有一个权值 **f[i]**，每条边都有一个权值 **t[i]**。

求图中的一个环，使“环上各点的权值之和”除以“环上各边的权值之和”最大。

输出这个最大值。

**注意**：数据保证至少存在一个环。

**输入格式**

第一行包含两个整数 **L** 和 **P**。

接下来 **L** 行每行一个整数，表示 **f[i]**。

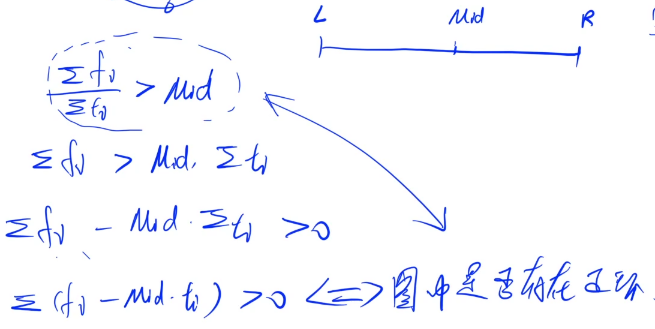
再接下来 **P** 行，每行三个整数 **a**，**b**，**t[i]**，表示点 **a** 和 **b** 之间存在一条边，边的权值为 **t[i]**。

**输出格式**

输出一个数表示结果，结果保留两位小数。

**数据范围**

**分析**：通过题目我们知道我们要求的是的最大值,则想到甩01分数规划解决问题。



以上分析过程即为01分数规划的大致解决思路。

#include<bits/stdc++.h>  
using namespace std;  
const int N = 1010,M = 5010;  
int h[N],e[M],ne[M],wt[M],idx;  
int wf[N],cnt[N];  
double dist[N];  
bool st[N];  
int n,m;  
  
void add(int a, int b, int c) // 添加一条边a->b，边权为c  
{  
 e[idx] = b, wt[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;  
}  
  
bool check(double mid)  
{  
 memset(cnt,0,sizeof cnt);  
 memset(st, 0, sizeof st);  
 queue<int> q;  
 for(int i=1;i<=n;i++)  
 {  
 q.push(i);  
 st[i]=1;  
 }  
 while(q.size())  
 {  
 auto t = q.front();  
 q.pop();  
 st[t]=0;  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(dist[j]<dist[t]+wf[t]-mid\*wt[i])//这里的wf应该是t，因为是更新所有连在t上的边的点。  
 {  
 dist[j]=dist[t]+wf[t]-mid\*wt[i];  
 cnt[j]=cnt[t]+1;  
 if(cnt[j]>=n) return 1;  
 if(!st[j])  
 {  
 q.push(j);  
 st[j]=1;  
 }  
 }  
 }  
 }  
 return 0;  
}  
int main()  
{  
 cin>>n>>m;  
 memset(h,-1,sizeof h);  
 for(int i=1;i<=n;i++) cin>>wf[i];  
 while (m -- )  
 {  
 int a,b,c;  
 cin>>a>>b>>c;  
 add(a,b,c);  
 }  
 double l=0,r=1000;  
 while(r-l>1e-4)  
 {  
 double mid = (l+r)/2;  
 if(check(mid)) l=mid;  
 else r=mid;  
 }  
  
 printf("%.2lf",r);  
 return 0;  
}

解决01分数规划问题，有一些经验。

如果我们求得是最大值，那跑的就是最长路。

若求得是最小值，那跑的就是最短路。

#### 差分约束

[差分约束图解](https://www.acwing.com/solution/content/20514/)   
**差分约束主要解决问题**  
1.求不等式的可行解

**原点需要满足的条件:从 源点 出发，一定可以走到所有的边。**

步骤:  
 1.现将每个不等式 ,转化为一条从走到，长度为的一条边  
 2.找一个 超级源点 ，使得该源点一定可以 遍历所有边 。  
 3.从源点求一边单源最短路  
 **结果1**:如果 存在负环，则原不等式组一定 无解  
 **结果2**:如果 没有负环，则**dist[i]**就是原不等式组的 可行解

2.如何求 最大值 或 最小值   
**结论**:如果求的是 最小值，则应该求 最长路 ，不等式组的符号列成   
 如果求的是 最大值，则应该求 最短路 ，不等式组的符号列成

问题:如何转化,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，然后建立0->i长度是c的边即可

**经验**  
写了几道题的感受

1.差分约束题目的一些特征:拥有大量的不等关系，且这些不等关系是有联系的。

2.一些小细节:

* 要确认时算的最大值还是最小值，确定不等号方向，这样能确定跑最长路还是最短路。
* 一定要记得超级源点，很关键，确定了超级源点，才能确定最开始放入队列的元素。
* 还有一定要找到边界，就是一个未知量大于一个准确值，这样我们就找到了，该未知量和边界的情况，从而递归得到，上下界。
* 等式条件一定也不要忘了。

### 最小生成树

#### Prim

时间复杂度：

const int N = 510,M=1e5+5,INF=0x3f3f3f3f;  
int n,m;  
int g[N][N];  
int dist[N];  
bool st[N];  
int Prim()  
{  
 memset(dist,0x3f,sizeof dist);  
 int res=0;  
 for(int i=0;i<n;i++)  
 {  
 int t=-1;  
 for(int j=1;j<=n;j++)  
 if(!st[j]&&(t==-1||dist[t]>dist[j]))  
 t=j;  
 if(i&&dist[t]==INF) return INF;  
 if(i) res+=dist[t];  
 st[t]=1;  
 for(int j=1;j<=n;j++) dist[j]=min(dist[j],g[t][j]);  
 }  
 return res;  
}

其实Prim也是分，朴素与堆优化版本的。

但后面我们会发现，Prim算法的时间复杂度和代码复杂度都不如Kruskal，所以这里就不放堆优化的了。

#### Kruskal

时间复杂度：

struct Edge  
{  
 int a,b,w;  
 bool operator<(Edge &e) const   
 {  
 return w<e.w;  
 }  
}edges[M];  
int n,m;  
int p[N];  
int find(int x)  
{  
 if(p[x]!=x) p[x]=find(p[x]);  
 return p[x];  
}  
int Kruskal()  
{  
 for(int i=1;i<=n;i++) p[i]=i;  
 int res=0,cnt=0;  
 for(int i=0;i<m;i++)  
 {  
 int a=edges[i].a,b=edges[i].b,w=edges[i].w;  
 a=find(a),b=find(b);  
 if(a!=b)  
 {  
 p[a]=b;  
 cnt++;  
 res+=w;  
 }  
 }  
 if(cnt<n-1) return INF;  
 return res;  
}

可以看出Kruskal 的时间复杂度极低，而且代码非常容易写。

同时，有一个很重要的结论。

**不论，什么时候开始(是已经连了一些边还是一点都没有连接)，我们的Kruskal算法都是生成的当前条件下的最小生成树**

同时，我们可以发现，**随着Kruskal的进行，连通块的数量在不断的减小**。

#### 一些小结论

* 对一个最小生成树，将其扩充为一个完全图时，算所需要加的边权最小值。

我们可以在每一次合并集合的时候，将两个集合连接成完全图。除了要合并的边之外，其他边贪心的想，应该为w+1，这样才能使得完全图的新加权值最小

* 当n不大时，我们可以用dfs+Kruskal算出次小生成树，这个，将会在下边给出详细展开。

### 次小生成树

#### 引入

计算次小生成树，我们有两种方法、

1. 先求最小生成树，在枚举删去最小生成树中的边求解。时间复杂度：
2. 先求最小生成树，然后一次枚举非树边，然后将该边加入树中，同时从树中去掉一条边，使得最终的图仍是一棵树，则一定可以求出次小生成树。

第一种方法，不论是从时间还是其本身的缺点来说，我们都不会选择。

接下来我们展开说说，第二种方法。

#### 解题步骤

1. 求最小生成树，统计标记每条边是树边，还是非树边;同时把最小生成树建立出来。
2. 预处理任意两点间的边权最大值与次小值dist1[a] [b], dist2[a] [b]
3. 以此枚举所有非树边，求min(sum+w-dist [a] [b]),满足w>dist [a] [b];

#### 细节

这里需要注意的第一点是：

你求的是严格的还是非严格的次小生成树。

若是严格的，那进行替换时，我们删去的是a,b 两点之间的最长边，但是，若是删去的最长边和我们要加的边权相等时，我们就要删去次长边了。

若求的是非严格的，那就只需要wdist [a] [b]（对于枚举的每条非树边，其长都大于等于树边的长，因此如果要求非严格的，那直接比较是否大于等于最大值即可。只需要在严格条件下，把大于号改为大于等于号即可）。

#### DFS 时间复杂度

#include<bits/stdc++.h>  
using namespace std;  
typedef long long LL;  
const int N = 510,M = 1e4 + 10;  
struct Edge  
{  
 int a,b,w;  
 bool f;  
 bool operator<(const Edge& t)const  
 {  
 return w<t.w;  
 }  
}edge[M];  
int p[N];  
int dist1[N][N],dist2[N][N];  
int h[N],e[N\*2],ne[N\*2],w[N\*2],idx;  
int n,m;  
void add(int a,int b,int c)  
{  
 e[idx]=b,ne[idx]=h[a],w[idx]=c,h[a]=idx++;  
}  
int find(int x)  
{  
 if(p[x]!=x) p[x]=find(p[x]);  
 return p[x];  
}  
void dfs(int u, int fa, int maxd1, int maxd2, int d1[], int d2[])  
{  
 d1[u] = maxd1, d2[u] = maxd2;  
 for (int i = h[u]; ~i; i = ne[i])  
 {  
 int j = e[i];  
 if (j != fa)  
 {  
 int td1 = maxd1, td2 = maxd2;  
 if (w[i] > td1) td2 = td1, td1 = w[i];//当最大边可以被更新时，使次长边等于最大边，然后更新  
 else if (w[i] < td1 && w[i] > td2) td2 = w[i];//次长边更新  
 dfs(j, u, td1, td2, d1, d2);  
 }  
 }  
}  
//相对于下面的那个来说，只保存的一条路上的最大值，但是是有bug的  
//当最大值和我们要去掉的边长度相等时，我们就要去掉次短边了  
// int dfs(int u,int fa,int maxd,int d[])  
// {  
// d[u]=maxd;  
// for(int i = h[u];~i;i=ne[i])  
// {  
// int j = e[i];  
// if(j!=fa)  
// {  
// dfs(j,u,max(maxd,w[i]),d);  
// }  
// }  
// }  
int main()  
{  
 cin>>n>>m;  
 memset(h, -1, sizeof h);  
 for(int i=0;i<m;i++)  
 {  
 int a,b,w;  
 cin>>a>>b>>w;  
 edge[i]={a,b,w};  
 }  
 sort(edge,edge+m);  
 LL sum = 0;  
 for(int i=1;i<=n;i++) p[i]=i;  
 for(int i=0;i<m;i++)  
 {  
 int a=edge[i].a,b=edge[i].b,w = edge[i].w;  
 int pa=find(a),pb=find(b);  
 if(pa!=pb)  
 {  
 p[pa]=pb;  
 sum+=w;  
 add(a,b,w),add(b,a,w);  
 edge[i].f=1;  
 }  
 }  
 for (int i = 1; i <= n; i ++ ) dfs(i, -1, -1e9, -1e9, dist1[i], dist2[i]);  
  
 LL res = 1e18;  
 for (int i = 0; i < m; i ++ )  
 if (!edge[i].f)  
 {  
 int a = edge[i].a, b = edge[i].b, w = edge[i].w;  
 LL t;  
 if (w > dist1[a][b])  
 t = sum + w - dist1[a][b];  
 else if (w > dist2[a][b])  
 t = sum + w - dist2[a][b];  
 res = min(res, t);  
 }  
 cout<<res<<endl;  
 return 0;  
}

#### LCA 时间复杂度

#include<bits/stdc++.h>  
using namespace std;  
typedef long long LL;  
const int N = 1e5+10,M = 3e5 + 10,INF = 0x3f3f3f3f;  
struct Edge  
{  
 int a,b,w;  
 bool used;  
 bool operator<(const Edge& t) const  
 {  
 return w<t.w;  
 }  
}edge[M];  
int h[N],e[M],ne[M],w[M],idx;  
int p[N];  
int depth[N],fa[N][17],d1[N][17],d2[N][17];  
int n,m;  
void add(int a,int b,int c)  
{  
 e[idx]=b,ne[idx]=h[a],w[idx]=c,h[a]=idx++;  
}  
  
int find(int x)  
{  
 if(p[x]!=x) p[x]=find(p[x]);  
 return p[x];  
}  
  
LL Kruskal()  
{  
 LL sum = 0;  
 for(int i=1;i<=n;i++) p[i]=i;  
 sort(edge,edge+m);  
 for(int i=0;i<m;i++)  
 {  
 int a = find(edge[i].a),b = find(edge[i].b);  
 if(a!=b)  
 {  
 p[a]=b;  
 edge[i].used=1;  
 sum+=edge[i].w;  
 }  
 }  
 return sum;  
}  
  
void bfs()//递归得到fa,depth,d1,d2  
{  
 memset(depth,0x3f,sizeof depth);  
 depth[0]=0,depth[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(depth[j]>depth[t]+1)  
 {  
 q.push(j);  
 depth[j]=depth[t]+1;  
 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]};//最大值及次大值所有可能  
 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;  
 }  
 }  
 }  
 }  
 }  
}  
  
void build()//将所有树边建起来  
{  
 memset(h, -1, sizeof h);  
 for(int i=0;i<m;i++)  
 {  
 int a = edge[i].a,b = edge[i].b,w=edge[i].w;  
 if(edge[i].used) add(a,b,w),add(b,a,w);  
 }  
}  
  
int lca(int a,int b,int w)  
{  
 int cnt = 0;  
 static int distance[N\*2];//记录所有可能是最大值以及次大值的值  
 if(depth[a]<depth[b]) swap(a,b);//记得要让a是小的那一个  
 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 res1 = -INF,res2 = -INF;  
 for(int i=0;i<cnt;i++)  
 {  
 int d = distance[i];  
 if(d>res1) res2=res1,res1=d;  
 else if(d!=res1&&d>res2) res2=d;  
 }  
 if(w>res1) return w-res1;  
 if(w>res2) return w-res2;  
 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)  
 res=min(res,sum+lca(edge[i].a,edge[i].b,edge[i].w));  
  
 printf("%lld\n",res);  
 return 0;  
}

### LCA

#### 倍增法

时间复杂度：

**步骤**

1. 初始化:通过dfs初始化两个数组depth[],fa[i,j];  
    depth[i]:表示深度  
    fa[i,j]:表示从i开始，向上走步所能走到的节点编号()  
    哨兵:如果从i开始跳步会跳过根节点，那么fa[i,j]=0,depth[0]=0;
2. 查询  
   [1]现将两个点同时调到同一层  
   [2]让两个点同时往上跳，一直跳到它们的**最近公共祖先的下一层**

**注意**

哨兵一定一定记得要放，d[0]=0.

其次lca中，跳到同一层的判断是.

int n,m;  
int h[N],e[M],ne[M],idx;  
int fa[N][16];  
int d[N];  
void add(int a,int b)  
{  
 e[idx]=b,ne[idx]=h[a],h[a]=idx++;  
}  
void bfs(int root)  
{  
 memset(d,0x3f,sizeof d);  
 d[0]=0,d[root]=1;  
 queue<int> q;  
 q.push(root);  
 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;  
 q.push(j);  
 fa[j][0]=t;//递归找到，从j开始向上的所有倍增能走到的节点。  
 for(int k=1;k<=15;k++)  
 fa[j][k]=fa[fa[j][k-1]][k-1];  
 }  
 }  
 }  
}  
  
int lca(int a,int b)  
{  
 if(d[a]<d[b]) swap(a,b);  
 for(int k = 15;k>=0;k--)//将a于b放到同一层  
 if(d[fa[a][k]]>=d[b])//只要这一步让a的层数不超过b，那就走  
 a=fa[a][k];  
 if(a==b) return a;//如果已经重合为一个点，那就直接返回  
 for(int k=15;k>=0;k--)//接着让在同一层的点，同时向上走，走到最近公共父节点的前一层，停下。  
 if(fa[a][k]!=fa[b][k])  
 {  
 a=fa[a][k];  
 b=fa[b][k];  
 }  
  
 return fa[a][0];  
}

#### 离线Trajan

时间复杂度：

这里放的板子是结合了求两点之间距离的板子。

#include<bits/stdc++.h>  
using namespace std;  
typedef pair<int, int> PII;  
const int N = 1e4 + 10,M=N\*2;  
int h[N],e[M],ne[M],w[M],idx;  
int dist[N];  
int p[N];  
int res[M];//这里存的是询问个数，emm，存成N了，傻了  
int st[N];  
vector<PII> query[N];  
int n,m;  
  
void add(int a,int b,int c)  
{  
 e[idx]=b,ne[idx]=h[a],w[idx]=c,h[a]=idx++;  
}  
  
void dfs(int u,int fa)//dfs得到所有节点到根节点的距离  
{  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(j==fa) continue;  
 dist[j]=dist[u]+w[i];  
 dfs(j,u);  
 }  
}  
  
int find(int x)  
{  
 if(p[x]!=x) p[x]=find(p[x]);  
 return p[x];  
}  
  
void tarjan(int u)  
{  
 st[u]=1;//当开始遍历时，把改点标记成1  
  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(!st[j])  
 {  
 tarjan(j);//dfs下个节点，另外一定要先遍历再合并，这里比较好记。只要记住，我们只把已经回溯过的点进行合并就好  
 p[j]=u;  
 }  
 }  
  
 for(auto item:query[u])//遍历所有与u有关的查询  
 {  
 int y = item.first,id = item.second;  
 if(st[y]==2)//如果这个点已经回溯过了，那就可以直接找到最近公共祖先  
 {  
 int anc = find(y);  
 res[id] = dist[u] + dist[y] - dist[anc]\*2;  
 }  
 }  
  
 st[u]=2;//该点被利用过，回溯标记为2  
}  
  
int main()  
{  
 scanf("%d%d",&n,&m);  
 memset(h, -1, sizeof h);  
 for(int i=0;i<n-1;i++)  
 {  
 int a,b,c;  
 scanf("%d%d%d",&a,&b,&c);  
 add(a,b,c);  
 add(b,a,c);  
 }  
  
 for(int i=0;i<m;i++)  
 {  
 int a,b;  
 scanf("%d%d",&a,&b);  
 if(a!=b){  
 query[a].push\_back({b,i});  
 query[b].push\_back({a,i});  
 }  
 }  
  
 for(int i=1;i<=n;i++) p[i]=i;  
  
 dfs(1,-1);  
  
 tarjan(1);  
  
 for(int i=0;i<m;i++)  
 printf("%d\n",res[i]);  
  
 return 0;  
}

在优先遍历时，将所有点分为三大类:  
[0] 还未搜索过的点  
[1] 正在搜索的分支  
[2] 已经遍历过，且回溯过的点。

**步骤**:  
1.在进入递归层时，将点标记为1  
2.搜索所有没有遍历过的链接且与该点链接的点，搜索回溯后，完成集合合并。  
3.将所有与该层点有关系的询问，全部遍历，当另一个点已经被标记为2，则找到了最近公共祖先，就是另一个点的并查集标志节点。  
4.最后回溯时，将该节点标记为2.

**注意**:  
一定要先将要拓展的点进行拓展，回溯时再进行集合合并。记忆的话，就记住，我们进行集合合并的点一定是被回溯过的。

### LCA应用

#### 用于维护两点间的一些性质

例如我们可以维护最值。

#### 树上差分

这里直接放了个暗之连锁的板子了

树上差分进行的操作：

对两个节点x，y，以及他们的公共祖先p，对d[x]++，d[y]++，d[p]-=2。

从而对每个节点i而言，其与父节点之间连接的边权为i对应子树的节点权值和。

#include<bits/stdc++.h>  
using namespace std;  
const int N = 1e5 + 10,M = 2e5 + 10;  
int h[N],e[M],ne[M],w[M],idx;  
int d[N],depth[N];  
int fa[N][17];  
int n,m,ans;  
  
void add(int a,int b)  
{  
 e[idx]=b,ne[idx]=h[a],h[a]=idx++;  
}  
  
void bfs()  
{  
 memset(depth,0x3f,sizeof depth);  
 depth[0]=0,depth[1]=1;  
 queue<int> q;  
 q.push(1);  
 while(q.size())  
 {  
 auto t = q.front();  
 q.pop();  
 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.push(j);  
 fa[j][0]=t;  
 for(int k=1;k<=16;k++)  
 fa[j][k]=fa[fa[j][k-1]][k-1];  
 }  
 }  
 }  
}  
  
int lca(int a,int b)  
{  
 if(depth[a]<depth[b]) swap(a,b);  
 for(int i=16;i>=0;i--)  
 if(depth[fa[a][i]]>=depth[b])  
 a=fa[a][i];  
 if(a==b) return a;  
 for(int k=16;k>=0;k--)  
 if(fa[a][k]!=fa[b][k])  
 a=fa[a][k],b=fa[b][k];  
 return fa[a][0];  
}  
  
int dfs(int u,int fa)  
{  
 int t = d[u];  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(j==fa) continue;  
 int s = dfs(j,u);  
 if(s==0) ans+=m;  
 else if(s==1) ans++;  
 t+=s;  
 }  
 return t;  
}  
  
int main()  
{  
 cin>>n>>m;  
 memset(h, -1, sizeof h);  
 for(int i=0;i<n-1;i++)  
 {  
 int a,b;  
 cin>>a>>b;  
 add(a,b),add(b,a);  
 }  
   
 bfs();  
   
 for(int i=0;i<m;i++)  
 {  
 int a,b;  
 cin>>a>>b;  
 int p = lca(a,b);  
 d[a]++,d[b]++,d[p]-=2;  
 }  
   
 dfs(1,-1);  
   
 cout<<ans<<endl;  
   
 return 0;  
}

#### 算两点之间的距离。

emm，比较裸，说一下吧。

DFS抛出所有点距离根节点的距离。

接着对于每个节点x，y而言，其两点之间距离为，

d[x]+d[y]-2\*d[p]

#### 一些题目经验

1、两点之间距离，如果再添加一条边。求任意两点之间距离的时候，也可以o(logn)时间得到。

假设新添加的边为从u->v,则求从a-b之间的最短距离时，分类讨论

* 不过该条边
* a->u，u->v，v->b
* b->u，u->v，v->b

取三种情况最小值即可

### Tarjan求强连通分量

缩完点后的图为**拓扑图**

const int N = 1e4 + 10,M = 5e4 + 10;  
int h[N],e[M],ne[M],idx;  
int dfn[N];//到达该点的时间  
int low[N];//该点向上能到达的点的最小时间.  
int timestamp;//时间戳  
int id[N],cnt;//记录缩点后，每个点对应的缩点后的点  
bool st[N];//记录该点是否在栈中  
stack<int> //栈中记录的为，还有被回溯到的点.  
//trjan必备数组以上内容.  
int n,m;  
void add(int a,int b)  
{  
 e[idx]=b,ne[idx]=h[a],h[a]=idx++;  
}  
  
void tarjan(int u)  
{  
 s.push(u);//将该点放入栈中  
 dfs[u]=low[u]=++timestamp;  
 //将时间戳更新，并且记录一下该点的时间，并且初始化一下能到达的点的最小时间  
 st[u]=1;//记录点已经入栈.  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(!dfn[j])//若该点还没有被递归过  
 {  
 tarjan(j);//先进行递归  
 low[u]=min(low[u],low[j]);//接下来记录一下能到的点的最小时间.  
 }  
 else if(st[j]) low[u]=min(low[u],low[j]);  
 //若是该点j依旧在栈中，等于说与现在u在同一个SCC中，则更新一个low[u].  
 }  
 if(low[u]==dfn[u])//若是，能到的点的最小时间是自己，则说明是一个SCC的顶点  
 {  
 ++cnt;//先更新，SCC的计数  
 int y;//循环将其中的每一个元素取出  
 do  
 {  
 y = s.top();  
 s.pop();  
 st[y]=0;//标记一下已出站  
 id[y]=cnt;//将该点放入SCC  
 }while(y!=u)  
 }  
}

缩点操作后变成有向无环图

就能做拓扑排序了(此时连通分量编号id[]递减的顺序就是拓扑序了)

因为我们++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]

### Tarjan求双连通分量

#### 边的双连通分量(e-DCC)

桥：将桥断开后，连通图，会变成两个连通块。

e-DCC的概念：极大的不包括桥的连通块。

关于e-DCC的性质：其中任意两点之间，都有两条不相交的路径。

注意：其中的不相交，指的是，没有公共路径。

缩点后的图是一颗树。

因此引出的一个常见问题：

给一个无向图最少加几条边后，能够使其任意两点都有两条无公共路径的路径。

其代码，几乎与求SCC相同。

接下来，看代码，因为代码与SCC几乎相同，不做过多注释。

const int N = 5e3 + 10,M = 2e4 + 10;  
int n,m;  
int h[N],e[M],ne[M],idx;  
int dfn[N],low[N],timestamp;  
int id[N],dcc\_cnt;  
stack<int> s;  
bool is\_bridge[M];//标记边是否为桥  
int totedg;//统计一个图中的边个数，因为图可能不连通  
//双连通分量，还有一大特点就是，没有了标记数组  
  
void add(int a,int b)  
{  
 e[idx]=b,ne[idx]=h[a],h[a]=idx++;  
}  
  
void tarjan(int u,int fa)  
{  
 dfn[u]=low[u]=++timestamp;  
 s.push(u);  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(!dfn[j])  
 {  
 tarjan(j,u);  
 low[u]=min(low[u],low[j]);  
 if(dfn[u]<low[j]) is\_bridge[i]=is\_bridge[i^1]=1;//如果在j再也无法到u上方，则u-j是一个桥  
 }  
 else if(j!=fa) low[u]=min(low[u],dfn[j]);//同时，没有标记数组后，我们每次都需要对low[u]进行更新。  
 if(dfn[u]<dfn[j]) totedg++;  
 }  
 if(dfn[u]==low[u])  
 {  
 int y;  
 dcc\_cnt++;  
 do  
 {  
 y=s.top();  
 s.pop();  
 id[y]=dcc\_cnt;  
 }while(y!=u);  
 }  
}  
//缩点(可以这样缩，或者直接根据idx枚举边)  
for(int i=1;i<=n;i++)  
 for(int j=h[i];~j;j=ne[j])  
 {  
 int k = e[j];  
 if(id[i]!=id[k]) add(id[i],id[k]);  
 }

关于，边的双连通分量，我们可以引出简单的几个应用

* 每有一个桥，我们就可以断开它，获得多一个连通块。
* 桥内部，每两个点之间有两条不相交的路径。

#### 点的双连通分量(v-DCC)

割点：将割点去掉后，连通图将分为，不同的连通块。

**注意：每个割点至少属于两个连通分量，这就意味着，缩点后，同时会缩到两个点内**

v-DCC：极大的不包含割点的连通块

关于这个问题，就要麻烦的多了，我们拆成两个部分。

1.如何求割点  
 O x low(y)>=dfn(x)时分情况讨论  
 / 1.如果不是根节点，那就是  
 O y 2.如果是根节点，当连接在其上的V-DCC数量大于1时，则是一个割点。  
2.如何求双连通分量  
if(dfn[x]<=low[y])  
{  
 cnt++;//连接在该割点上的V-DCC数量  
 if(x非根节点||cnt>1) x是割点  
 将栈中元素弹出直至弹出y为止  
 且x也属于该"点双连通分量"  
}

而关于，点的双连通分量，我们先引入代码，在对其进行分析作用

const int N = 2e5 + 10,M = 4e5 + 10;  
//要缩点建图的点与边都多开一倍保险  
int h[N],e[M],ne[M],idx1;  
int hv[N],ev[M],nev[M],idx2;  
int dfn[N],low[N],ts;  
int id[N],d[N];  
int dcc\_cnt;  
bool cut[N];  
stack<int> s;  
vector<int> dcc[N];  
int root,n,m;  
  
void add(int a,int b)  
{  
 e[idx1] = b,ne[idx1] = h[a],h[a] = idx1++;  
}  
  
void addv(int a,int b)  
{  
 ev[idx2] = b,nev[idx2] = hv[a],hv[a] = idx2++;  
}  
  
void tarjan(int u)  
{  
 dfn[u]=low[u]=++ts;  
 s.push(u);  
 if(u==root&&h[u]==-1)//孤立点也是一个v-DCC  
 {  
 dcc\_cnt++;  
 dcc[dcc\_cnt].push\_back(u);  
 return ;  
 }  
 int cnt = 0;//用来统计每个割点断开后，会出现的v-DCC或者说连通块个数  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(!dfn[j])  
 {  
 tarjan(j);  
 low[u]=min(low[u],low[j]);  
 if(dfn[u]<=low[j])//判断u是否为割点  
 {  
 cnt++;//则出现一个连通块  
 if(u!=root||cnt>1) cut[u]=1;//如果u不是根节点，或者是根节点，但是连接在其上的v-DCC大于1，则u也为一个割点  
 ++dcc\_cnt;  
 int y;  
 do  
 {  
 y = s.top();  
 s.pop();  
 dcc[dcc\_cnt].push\_back(y);//将v-DCC中所有的点存起来  
 }while(y!=j);  
 dcc[dcc\_cnt].push\_back(u);//将割点放入，但是不将割点从栈中弹出。  
 }  
 }else low[u]=min(low[u],dfn[j]);  
 }  
 if(u!=root) cnt++;  
}  
  
//缩点略显复杂  
void get()  
{  
 memset(hv,-1,sizeof hv);  
 idx2 = 0;  
 int num = dcc\_cnt;  
 for(int i=1;i<=n;i++)  
 if(cut[i])  
 id[i] = ++num;//给割点一个编号  
 for(int i=1;i<=dcc\_cnt;i++)  
 for(auto x:dcc[i])  
 {  
 if(cut[x])  
 {  
 addv(i,id[x]);  
 addv(id[x],i);//利用割点建图  
 // d[i]++;d[id[x]]++;//如果用到度数，直接加就好。  
 }  
 else id[x] = i;  
 }  
 //注意缩完点后的图里边的点数为num  
}

##### 性质

1. **除了一种比较特殊的点双，其他的点双都满足：任意两点间都存在至少两条点不重复路径。**

* **比较特殊的点双**
* O - O

1. **图中任意一个割点都在至少两个点双中。**
2. **任意一个不是割点的点都只存在于一个点双中。**

**细节**  
1.点双连通分量的栈弹出位置与强连通分量和边的双连通分量不同。  
它是在当发现这是一个割点的时候，将栈内所有元素弹出，只剩割点。  
2.点双连通分量在进行缩点操作的时候，是将每一个割点所在的两个V-DCC向割点连一条边

O---O O <- V-DCC  
 \ / |  
 O <- 割点 缩点后 O <- 割点  
 / \ |  
 O---O O <- V-DCC

#### 小结：

对于双连通分量，我们可以做以下总结

1. 相对于强连通分量来说，两者都没有st数组(即判断是否在栈中的判断)。当其子节点已经在栈中了，那就直接更新low[u] = min(low[u],dfn[j])
2. 对点的双连通分量，不用判断是否会重复搜到父节点，因为判断割点的时候，是dfn[u]<=low[j]，此时在栈中的所有点都是一个dcc的
3. 而边的双连通分量，是根据定义用dfn[u]<low[j]来判断u - j是否是桥，因此一定要加入重复收到父节点的判断，不然当其子节点j能扫到其父节点u时，那其low[j]最小也是dfn[u]了，这样就无法判断出来桥了
4. 对点的双连通分量的判断是当dfn[u]<=low[j]出现时，等于出现了一个割点，则表示出现了一个包括u在内的dcc。记住对点的双连通分量，其出栈时，割点不出栈，但需要将割点放入dcc中。同时需要强调的是，虽然孤立点不是一个割点，但是其也是一个dcc，需要在最前面进行判断。
5. 对边的双连通分量的判断是当dfn[u]==low[j]时出现，此时说明u为dcc的顶点了。

### 对强连通和双连通的总结

其实大体上的总结已经说完了，我就是想说一下，关于一个争议。**low[u] = mini(low[u],dfn[j])**，在强连通中将dfn[j]改为low[j]是没问题的，但是在双连通中就wa。

下面是看到了一个大佬的解释，可供参考。

根据许多大佬的观点，我想提出自己的一点看法，在求强连通分量时，如果v已经在栈中，那么说明u，v一定在同一个强连通分量中，所以到最后low[u]=low[v]是必然的，提前更新也不会有问题，但是在求割点时，low的定义有了小小的变化，不再是最早能追溯到的祖先，（因为是个无向图）没有意义，应该是最早能绕到的割点，为什么用绕到，是因为是无向边，所以有另一条路可以走，如果把dfn[v]改掉就会上翻过头，可能翻进另一个环中，所以wa掉，仅是本人的一些个人看法，不知道讲的对不对，请各位指教。

### 欧拉回路与欧拉路径

存在欧拉路径或欧拉回路的条件  
无向图  
1.边都要具有**连通性**，可以有孤立点  
2.欧拉回路：奇数点个数为0，欧拉路径：奇数点个数为2

有向图  
1.边都要具有**连通性**，可以有孤立点

2.欧拉回路:对于每个点来说，出度与入相等。欧拉路径:除了两个点外，其他的点出度与入度相等。另外的两个点，一个入度比出度多1，另一个出度比入度多1

#### 求欧拉回路(路径)

#include<bits/stdc++.h>  
using namespace std;  
const int N = 1e5 + 10,M = 4e5 + 10;  
int n,m,type;  
int h[N],e[M],ne[M],idx;  
bool used[M];//标记该边是否被用过  
int ans[M],cnt;  
int din[N],dout[N];//如果是无向边，直接将din,dout相加就是对应的度数  
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;i=h[u])  
 //for(int &i=h[u];~i;)//这里的&i，就相当于h[u]了，如果不想这么写，那么h[u]=ne[i],i=ne[i],也行  
 { //因为这一步的优化其实就是，用头结点顶替我们删去的这条边。  
 if(used[i])//如果这条边用过了，那就顶替掉它；  
 {  
 h[u] = ne[i];  
 continue;  
 }  
   
 used[i]=1;//标记一下被用过了  
 if(type==1) used[i^1] = 1;  
   
 int t;//记录一下走过的边的编号  
 if(type==1)  
 {  
 t = i/2+1;  
 if(i&1) t = -t;//如果是反向边，需要加负号  
 }  
 else t = i+1;  
   
 int j = e[i];//如果用的是&i的for循环，那这个就必须加了，如果是，最后使得i=h[u]那就无所谓。  
   
 h[u]=ne[i];//这里要先删，再进行搜索；  
 dfs(j);  
   
 ans[++cnt] = t;//等搜索结束了，我们再进行赋值  
 }  
}  
  
int main()  
{  
 scanf("%d",&type);  
 scanf("%d%d", &n, &m);  
 memset(h, -1, sizeof h);  
 for(int i=0;i<m;i++)  
 {  
 int a,b;  
 scanf("%d%d",&a,&b);  
 add(a,b);  
 if(type==1) add(b,a);  
 din[b]++,dout[a]++;  
 }  
   
 if(type==1)//矛盾情况，直接输出NO  
 {  
 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)//孤立点可以有，因为我们需要的是边的连通性  
 {  
 dfs(i);  
 break;  
 }  
   
 if(cnt<m)  
 {  
 puts("NO");  
 return 0;  
 }  
   
 puts("YES");  
 for(int i=cnt;i;i--) printf("%d ",ans[i]);  
 puts("");  
 return 0;  
}

#### 求欧拉路径的一些变形

##### 求欧拉路径，是的最后搜的，路径上的点的字典序最小，

做法很简单，就是对一个节点，搜边的时候，以终点的字典序从小到大搜即可

举个例子:

对节点u而言，它连接的边，终点分别为1,3,5

则搜索的时候，就是以终点分别为1,3,5的边点的顺序搜即可、

给一个例题，同时这个例题，也展示了一下，邻接矩阵的欧拉路径的推导

**题目大意**

给一个节点数为500的图，其中的边1024，双向边。

要求输出，一个欧拉路径，其中经过的点是字典序最小的

**输入格式**

第 **1** 行:一个整数 **F**，表示栅栏的数目;

第 **2** 到**F+1** 行:每行两个整数 **i,j** 表示这条栅栏连接 **i** 与 **j** 号顶点。

**输出格式**

输出应当有 **F+1** 行，每行一个整数，依次表示路径经过的顶点号。

注意数据可能有多组解，但是只有上面题目要求的那一组解是认为正确的。

#include<bits/stdc++.h>  
using namespace std;  
const int N = 510,M = 1100;  
int g[N][N];  
int d[N];  
int ans[M],cnt;  
int n=500,m;  
  
void dfs(int u)  
{  
 for(int i=1;i<=n;i++)  
 {  
 if(g[u][i])  
 {  
 g[u][i]--,g[i][u]--;  
 dfs(i);  
 }  
 }  
   
 ans[++cnt]=u;  
}  
  
int main()  
{  
 cin>>m;  
 for(int i=0;i<m;i++)  
 {  
 int a,b;  
 cin>>a>>b;  
 g[a][b]++,g[b][a]++;  
 d[a]++,d[b]++;  
 }  
   
 int start = 1;  
 while(!d[start]) start++;  
 for(int i=1;i<=n;i++)  
 if(d[i]&1)  
 {  
 start = i;  
 break;  
 }  
   
 dfs(start);  
   
 for(int i=cnt;i;i--) cout<<ans[i]<<endl;   
 return 0;  
}

### 拓扑排序

const int N = 1e5+5;  
int e[N],ne[N],h[N],idx;  
int q[N],hh,tt;  
int d[N];  
int n,m;  
void add(int a,int b)  
{  
 e[idx]=b,ne[idx]=h[a],h[a]=idx++;  
}  
bool bfs()  
{  
 hh=0,tt=-1;  
 for(int i=1;i<=n;i++)  
 if(!d[i])  
 q[++tt]=i;  
 while(hh<=tt)  
 {  
 int t=q[hh++];  
 for(int i=h[t];i!=-1;i=ne[i])  
 {  
 int j=e[i];  
 if(--d[j]==0)  
 q[++tt]=j;  
 }  
 }  
 return tt==n-1;  
}

### 关键路径

### 二分图

**二分图<=>不存在奇数环<=>染色法不存在矛盾**

#### 染色图判定二分图

时间复杂度

#include<bits/stdc++.h>  
using namespace std;  
const int N = 1e5+5,M=N\*2;  
int h[N],e[M],ne[M],idx;  
int n,m;  
int color[N];  
void add(int a,int b)  
{  
 e[idx]=b,ne[idx]=h[a],h[a]=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;  
 }else if(color[j]==c) return false;  
 }  
 return 1;  
}  
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);  
 }  
 bool flag=1;  
 for(int i=1;i<=n;i++)  
 {  
 if(!color[i])  
 {  
 if(!dfs(i,1))  
 {  
 flag=0;  
 break;  
 }  
 }  
 }  
 if(flag) puts("Yes");  
 else puts("No");  
 return 0;  
}

#### 匈牙利算法

二分图最大匹配

时间复杂度

但是实际不会走这么多次

#include<bits/stdc++.h>  
using namespace std;  
  
const int N = 510,M=1e5+5;  
  
int h[N],e[M],ne[M],idx;  
int match[N];//标志与女生匹配的男生编号  
bool st[N];//对每一次男生选择而言，女生是否被选择过。  
int n1,n2,m;  
void add(int a, int b)   
{  
 e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;  
}  
  
bool find(int x)  
{  
 for(int i=h[x];i!=-1;i=ne[i])  
 {  
 int j=e[i];  
 if(!st[j])//女生是否被选择过  
 {  
 st[j]=1;//没选择过则标记一下  
 if(!match[j]||find(match[j])) //如果没匹配男生，或匹配的男生可以有其他选择，则可以匹配成功。  
 {  
 match[j]=x;  
 return 1;  
 }  
 }  
 }  
 return 0;  
}  
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, 0, sizeof st);  
 if(find(i)) res++;  
 }  
 printf("%d\n",res);  
 return 0;  
}

#### 匈牙利算法的应用

**注意，其中提到的图大部分只针对二分图，若没有提，默认二分图，若提出了特殊的图，则以提出的为主**

##### 匹配，最大匹配，匹配点，增广路径

一些基础概念，对理解后面一些概念有用。

匹配：指一组边，其中任意两条边没有公共点。

最大匹配：指的是一个二分图中的最大匹配边的数量。

匹配点：在匹配中的点，叫匹配点。

非匹配点：不在匹配中的点，叫做非匹配点。

增广路径：从左边的非匹配点出发，交替走非匹配边，匹配边，最后一定能走到右边的一个非匹配点上

**结论：最大匹配<=>不存在增广路径**

##### 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)

最小点覆盖：选出最少的点，使得每一条边两个端点里面至少选出了一个点。

最大独立集：选出最多的点，使得选出的点之间没有边。

这个概念等价于去掉最少的点，将所有的边都破坏掉。

最小路径点覆盖：针对一个DAG，用最少的互不相交的路径，将所有点覆盖。

应用时：

对一个DAG，拆点  
 1 1‘  
 2 2’  
 3 3‘  
 . .  
 . .  
 . .  
 n n'  
 G G'  
转化后，变为了一个二分图。则最小路径点覆盖=n-最大匹配数.

最小路径重复点覆盖：先求原DAG的传递闭包，原图的最小重复点覆盖 = 新图的最小路径点覆盖。

最小路径重复点覆盖和最小路径点覆盖的唯一区别，其实就体现在，在选择点的时候，是否可以选择相同的点。

**结论：最大匹配数 = 最小点覆盖 = 总点数 - 最大独立集 = 总点数 - 最小路径点覆盖(最小路径重复点覆盖)**

#### KM算法(最大权值匹配)

//KM模板   
//本题hdu6346 用于求二分图最小权匹配   
//如果边权不取反 则用于求二分图最大权匹配  
//否则边权取反之后再对答案取反 用于求二分图最小权匹配   
#include <bits/stdc++.h>  
using namespace std;  
typedef long long ll;  
const int N=205;  
const ll INF=0x3f3f3f3f3f3f3f3fll;  
int t,n;  
struct KM{  
 int n,m[N],way[N];  
 ll w[N][N],lx[N],ly[N],sl[N];  
 bool u[N];  
 void init(){  
 scanf("%d",&n);  
 for(int i=1;i<=n;++i){  
 for(int j=1;j<=n;++j){  
 scanf("%lld",&w[i][j]);  
 w[i][j]=-w[i][j];  
 }  
 }  
 }  
 void hungary(int x){  
 m[0]=x;  
 int j0=0;  
 fill(sl,sl+n+1,INF);  
 fill(u,u+n+1,0);  
 do{  
 u[j0]=1;  
 int i0=m[j0],j1=0;  
 ll d=INF;  
 for(int j=1;j<=n;++j)  
 if(u[j]==0){  
 ll cur=-w[i0][j]-lx[i0]-ly[j];  
 if(cur<sl[j]){sl[j]=cur;way[j]=j0;}  
 if(sl[j]<d){d=sl[j];j1=j;}  
 }  
 for(int j=0;j<=n;++j){  
 if(u[j]){lx[m[j]]+=d;ly[j]-=d;}  
 else sl[j]-=d;  
 }  
 j0=j1;  
 }while(m[j0]!=0);  
 do{  
 int j1=way[j0];m[j0]=m[j1];j0=j1;  
 }while(j0);  
 }  
 ll solve(){  
 for(int i=1;i<=n;++i)m[i]=lx[i]=ly[i]=way[i]=0;  
 for(int i=1;i<=n;++i)hungary(i);  
 ll sum=0;  
 for(int i=1;i<=n;++i)sum+=w[m[i]][i];  
 return sum;  
 }  
}q;  
int main()  
{  
 scanf("%d",&t);  
 for(int c=1;c<=t;++c){  
 q.init();  
 printf("Case #%d: %lld\n",c,-q.solve());  
 }  
 return 0;  
}

### 网络流

#### 模板

**EK**

时间复杂度

#include<bits/stdc++.h>  
  
using namespace std;  
  
const int N = 1010,M = 20010,INF = 0x3f3f3f3f;  
  
int h[N],ne[M],e[M],w[M],idx;  
int pre[N],d[N],q[N];  
bool st[N];  
int n,m,S,T;  
  
void add(int a,int b,int c)  
{  
 e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx++;  
 e[idx] = a,ne[idx] = h[b],w[idx] = 0,h[b] = idx++;  
}  
  
bool bfs()  
{  
 int hh = 0,tt = -1;  
 memset(st,0,sizeof st);  
 memset(d,0,sizeof d);  
 q[++tt] = S;st[S] = 1;d[S] = INF;  
 while(hh<=tt)  
 {  
 auto t = q[hh++];  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(!st[j]&&w[i])  
 {  
 st[j] = 1;  
 d[j] = min(d[t],w[i]);  
 pre[j] = i;  
 if(j==T) return 1;  
 q[++tt] = j;  
 }  
 }  
 }  
 return 0;  
}  
  
int EK()  
{  
 int r = 0;  
 while(bfs())  
 {  
 r += d[T];  
 for(int i=T;i!=S;i=e[pre[i]^1])  
 w[pre[i]] -= d[T],w[pre[i]^1] += d[T];  
 }  
 return r;  
}  
  
int main()  
{  
 scanf("%d%d%d%d",&n,&m,&S,&T);  
 memset(h,-1,sizeof h);  
 for(int i=0;i<m;i++)  
 {  
 int u,v,c;scanf("%d%d%d",&u,&v,&c);  
 add(u,v,c);  
 }  
 printf("%d\n",EK());  
 return 0;  
}

**Dinic**

时间复杂度

#include<bits/stdc++.h>  
using namespace std;  
  
const int N = 10010,M = 200010,INF = 0x3f3f3f3f;  
  
int h[N],ne[M],e[M],w[M],idx;  
int cur[N],d[N],q[N];  
int n,m,S,T;  
  
void add(int a,int b,int c)  
{  
 e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx++;  
 e[idx] = a,ne[idx] = h[b],w[idx] = 0,h[b] = idx++;  
}  
  
bool bfs()  
{  
 int hh = 0,tt = -1;  
 memset(d,0x3f,sizeof d);  
 q[++tt] = S,d[S] = 1,cur[S] = h[S];  
 while(hh<=tt)  
 {  
 auto t = q[hh++];  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(d[j]==INF&&w[i])  
 {  
 d[j] = d[t] + 1;  
 cur[j] = h[j];  
 if(j==T) return 1;  
 q[++tt] = j;  
 }  
 }  
 }  
 return 0;  
}  
  
int find(int u,int limit)  
{  
 if(u==T) return limit;  
 int flow = 0;  
 for(int i=cur[u];~i&&flow<limit;i=ne[i])  
 {  
 int j = e[i];  
 cur[u] = i;  
 if(d[j]==d[u]+1&&w[i])  
 {  
 int t = find(j,min(limit-flow,w[i]));  
 if(!t) d[j] = -1;  
 w[i] -= t,w[i^1] += t,flow += t;  
 }  
 }  
 return flow;  
}  
  
int dinic()  
{  
 int r = 0,flow;  
 while(bfs()) if(flow = find(S,INF)) r += flow;  
 return r;  
}  
  
int main()  
{  
 scanf("%d%d%d%d",&n,&m,&S,&T);  
 memset(h, -1, sizeof h);  
 while(m--)  
 {  
 int u,v,c;scanf("%d%d%d",&u,&v,&c);  
 add(u,v,c);  
 }  
 printf("%d\n",dinic());  
 return 0;  
}

#### 最大流

##### 二分图最大匹配

**网络流建图**

* **从源点向左边点连一条容量为1的边**
* **从所有右边的点向汇点连一条容量为1的边**
* **将中间的所有连接，从左边点向右边点连接一条容量为1的边**

**确定方案**

**从左边的点开始扫描所有连向右边的边，若边的流量为0，则说明选择了该条边**

##### 二分图多重最大匹配

与二分图最大匹配的最大不同为：

**二分图最大匹配中，左右两个点都只能被用一次，而在多重匹配中，左右的点都可以多次被用**

**网络流建图**

* **从源点向左边点连一条容量为的边**
* **从所有右边的点向汇点连一条容量为的边**
* **将中间的所有连接，从左边点向右边点连接一条容量为1的边**

**确定方案**

**从左边的点开始扫描所有连向右边的边，若边的流量为0，则说明选择了该条边**

##### 无源汇上下界可行流

先放个板子

#include<bits/stdc++.h>  
using namespace std;  
const int N = 210,M = (10200+N)\*2,INF = INT\_MAX;  
int h[N],e[M],ne[M],w[M],l[M],idx;  
int d[N],cur[N],A[N];  
int n,m,S,T;  
  
void add(int a,int b,int c,int d)  
{  
 e[idx]=b,ne[idx]=h[a],l[idx]=c,w[idx]=d-c,h[a]=idx++;  
 e[idx]=a,ne[idx]=h[b],l[idx]=0,w[idx]=0,h[b]=idx++;  
}  
  
bool bfs()  
{  
 queue<int> q;  
 memset(d,-1,sizeof d);  
 d[S]=0,q.push(S),cur[S]=h[S];  
 while(q.size())  
 {  
 auto t = q.front();  
 q.pop();  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(d[j]==-1&&w[i])  
 {  
 d[j] = d[t] + 1;  
 cur[j] = h[j];  
 if(j==T) return 1;  
 q.push(j);  
 }  
 }  
 }  
 return 0;  
}  
  
int find(int u,int limit)  
{  
 if(u==T) return limit;  
 int flow = 0;  
 for(int i=cur[u];~i;i=ne[i])  
 {  
 cur[u]=i;  
 int j = e[i];  
 if(d[j]==d[u]+1&&w[i])  
 {  
 int t = find(j,min(w[i],limit-flow));  
 if(!t) d[j] = -1;  
 w[i] -= t,w[i^1] += t,flow+=t;  
 }  
 }  
 return flow;  
}  
  
int dinic()  
{  
 int r = 0,flow;  
 while(bfs()) while(flow=find(S,INF)) r+=flow;  
 return r;  
}  
  
int main()  
{  
 cin>>n>>m;  
 memset(h, -1, sizeof h);  
 S = 0,T = n+1;  
 for(int i=0;i<m;i++)  
 {  
 int a,b,c,d;  
 cin>>a>>b>>c>>d;  
 add(a, b, c, d);  
 A[a] -= c,A[b] += c;  
 }  
  
 int tot = 0;  
 for(int i=1;i<=n;i++)  
 if(A[i]>0) add(S,i,0,A[i]),tot+=A[i];  
 else if(A[i]<0) add(i,T,0,-A[i]);  
  
 if(dinic()!=tot) puts("NO");  
 else   
 {  
 puts("YES");  
 for(int i=0;i<m\*2;i+=2)  
 cout<<w[i^1]+l[i]<<endl;  
 }  
 return 0;   
}

##### 有源汇上下界最大流

#include<bits/stdc++.h>  
using namespace std;  
const int N = 210,M = (N + 10000)\*2,INF = INT\_MAX;  
int h[N],e[M],ne[M],w[M],idx;  
int d[N],cur[N],A[N];  
int n,m,S,T;  
  
void add(int a,int b,int c)  
{  
 e[idx]=b,w[idx]=c,ne[idx]=h[a],h[a]=idx++;  
 e[idx]=a,w[idx]=0,ne[idx]=h[b],h[b]=idx++;  
}  
  
bool bfs()  
{  
 queue<int> q;  
 memset(d,-1,sizeof d);  
 d[S] = 0,cur[S] = h[S],q.push(S);  
 while(q.size())  
 {  
 auto t = q.front();  
 q.pop();  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(d[j]==-1&&w[i])  
 {  
 d[j] = d[t] + 1;  
 cur[j] = h[j];  
 if(j==T) return 1;  
 q.push(j);  
 }  
 }  
 }  
 return 0;  
}  
  
int find(int u,int limit)  
{  
 if(u==T) return limit;  
 int flow = 0;  
 for(int i=cur[u];~i;i=ne[i])  
 {  
 cur[u] = i;  
 int j = e[i];  
 if(d[j] == d[u] + 1&&w[i])  
 {  
 int t = find(j,min(w[i],limit-flow));  
 if(!t) d[j] = -1;  
 w[i] -= t,w[i^1] += t,flow+=t;  
 }  
 }  
 return flow;  
}  
  
int dinic()  
{  
 int r = 0,flow;  
 while(bfs()) while(flow=find(S,INF)) r+=flow;  
 return r;  
}  
  
int main()  
{  
 int s,t;  
 cin>>n>>m>>s>>t;  
 S = 0,T = n + 1;  
 memset(h, -1, sizeof h);  
 while (m -- )  
 {  
 int a,b,c,d;  
 cin>>a>>b>>c>>d;  
 add(a,b,d-c);  
 A[a] -= c,A[b] +=c;  
 }  
 int tot = 0;  
 for(int i=1;i<=n;i++)  
 if(A[i]>0) add(S,i,A[i]),tot+=A[i];  
 else if(A[i]<0) add(i,T,-A[i]);  
 add(t,s,INF);  
 if(dinic()!=tot) puts("No Solution");  
 else  
 {  
 int res = w[idx-1];  
 S = s,T = t;  
 w[idx-1]=w[idx-2]=0;  
 cout<<res+dinic()<<endl;  
 }  
 return 0;  
}

##### 有源汇上下界最小流

#include<bits/stdc++.h>  
using namespace std;  
const int N = 50010,M = (N + 125010)\*2,INF = INT\_MAX;  
int h[N],e[M],ne[M],w[M],idx;  
int d[N],cur[N],A[N];  
int n,m,S,T;  
  
void add(int a,int b,int c)  
{  
 e[idx]=b,w[idx]=c,ne[idx]=h[a],h[a]=idx++;  
 e[idx]=a,w[idx]=0,ne[idx]=h[b],h[b]=idx++;  
}  
  
bool bfs()  
{  
 queue<int> q;  
 memset(d,-1,sizeof d);  
 d[S] = 0,cur[S] = h[S],q.push(S);  
 while(q.size())  
 {  
 auto t = q.front();  
 q.pop();  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(d[j]==-1&&w[i])  
 {  
 d[j] = d[t] + 1;  
 cur[j] = h[j];  
 if(j==T) return 1;  
 q.push(j);  
 }  
 }  
 }  
 return 0;  
}  
  
int find(int u,int limit)  
{  
 if(u==T) return limit;  
 int flow = 0;  
 for(int i=cur[u];~i&&flow<limit;i=ne[i])  
 {  
 cur[u] = i;  
 int j = e[i];  
 if(d[j] == d[u] + 1&&w[i])  
 {  
 int t = find(j,min(w[i],limit-flow));  
 if(!t) d[j] = -1;  
 w[i] -= t,w[i^1] += t,flow+=t;  
 }  
 }  
 return flow;  
}  
  
int dinic()  
{  
 int r = 0,flow;  
 while(bfs()) while(flow=find(S,INF)) r+=flow;  
 return r;  
}  
  
int main()  
{  
 int s,t;  
 cin>>n>>m>>s>>t;  
 S = 0,T = n + 1;  
 memset(h, -1, sizeof h);  
 while (m -- )  
 {  
 int a,b,c,d;  
 cin>>a>>b>>c>>d;  
 add(a,b,d-c);  
 A[a] -= c,A[b] +=c;  
 }  
 int tot = 0;  
 for(int i=1;i<=n;i++)  
 if(A[i]>0) add(S,i,A[i]),tot+=A[i];  
 else if(A[i]<0) add(i,T,-A[i]);  
 add(t,s,INF);  
 if(dinic()!=tot) puts("No Solution");  
 else  
 {  
 int res = w[idx-1];  
 S = t,T = s;  
 w[idx-1]=w[idx-2]=0;  
 cout<<res - dinic()<<endl;  
 }  
 return 0;  
}

##### 多源汇最大流

**网络流建图**

* **从虚拟源点向所有源点连一条容量为的边**
* **从所有汇点向虚拟汇点连一条容量为的边**
* **将中间的所有连接保持**

#include<bits/stdc++.h>  
using namespace std;  
const int N = 1e4 + 10,M = (100010 + N)\*2,INF = 0x3f3f3f3f;  
  
int h[N],ne[M],e[M],w[M],idx;  
int cur[N],d[N],q[N];  
int n,m,S,T,Sc,Tc;  
  
void add(int a,int b,int c)  
{  
 e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx++;  
 e[idx] = a,ne[idx] = h[b],w[idx] = 0,h[b] = idx++;  
}  
  
bool bfs()  
{  
 int hh = 0,tt = -1;  
 memset(d,-1,sizeof d);  
 d[S] = 0,q[++tt] = S,cur[S] = h[S];  
 while(hh<=tt)  
 {  
 int t = q[hh++];  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(d[j]==-1&&w[i])  
 {  
 d[j] = d[t] + 1;  
 cur[j] = h[j];  
 if(j==T) return 1;  
 q[++tt] = j;  
 }  
 }  
 }  
 return 0;  
}  
  
int find(int u,int limit)  
{  
 if(u==T) return limit;  
 int flow = 0;  
 for(int i=cur[u];~i&&flow<limit;i=ne[i])  
 {  
 int j = e[i];  
 cur[u] = i;  
 if(d[j]==d[u]+1&&w[i])  
 {  
 int t = find(j,min(w[i],limit-flow));  
 if(!t) d[j] = -1;  
 w[i] -= t,w[i^1] += t,flow += t;  
 }  
 }  
 return flow;  
}  
  
int dinic()  
{  
 int r = 0,flow;  
 while(bfs()) if(flow = find(S,INF)) r += flow;  
 return r;  
}  
  
int main()  
{  
 scanf("%d%d%d%d",&n,&m,&Sc,&Tc);  
 memset(h,-1,sizeof h);  
 S = 0,T = n + 1;  
 while(Sc--)  
 {  
 int x;scanf("%d",&x);  
 add(S,x,INF);  
 }  
 while(Tc--)  
 {  
 int x;scanf("%d",&x);  
 add(x,T,INF);  
 }  
 while (m -- )  
 {  
 int u,v,c;scanf("%d%d%d",&u,&v,&c);  
 add(u,v,c);  
 }  
 printf("%d\n",dinic());  
 return 0;  
}

##### 关键边

将某一条边的容量增大，可以增加最大流。

**判断方法**

* **随便跑一个最大流**
* **接下来求一下在原图上看源点和汇点能到达的点**
* **最后，对每一条满流的边判断，S是否能到达其起点，其终点是否能到达T，可以的话则是一条关键边**

板子放的是个求关键边数量的。

#include<bits/stdc++.h>  
using namespace std;  
const int N = 510,M = 10100,INF = 1e8;  
int h[N],e[M],ne[M],w[M],idx;  
int d[N],cur[N];  
bool sst[N],tst[N];  
int n,m,S,T;  
  
void add(int a,int b,int c)  
{  
 e[idx]=b,ne[idx]=h[a],w[idx]=c,h[a]=idx++;  
 e[idx]=a,ne[idx]=h[b],w[idx]=0,h[b]=idx++;  
}  
  
bool bfs()  
{  
 queue<int> q;  
 memset(d,-1,sizeof d);  
 d[S] = 0,q.push(S),cur[S] = h[S];  
 while(q.size())  
 {  
 auto t = q.front();  
 q.pop();  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(d[j]==-1&&w[i])  
 {  
 d[j] = d[t] + 1;  
 cur[j] = h[j];  
 if(j==T) return 1;  
 q.push(j);  
 }  
 }  
 }  
 return 0;  
}  
  
int find(int u,int limit)  
{  
 if(u==T) return limit;  
 int flow = 0;  
 for(int i=cur[u];~i&&flow<limit;i=ne[i])  
 {  
 cur[u] = i;  
 int j = e[i];  
 if(d[j]==d[u]+1&&w[i])  
 {  
 int t = find(j,min(w[i],limit-flow));  
 if(!t) d[j] = -1;  
 w[i] -= t,w[i^1] += t,flow+=t;  
 }  
 }  
 return flow;  
}  
  
int dinic()  
{  
 int r = 0,flow;  
 while(bfs()) while(flow=find(S,INF)) r+=flow;  
 return r;  
}  
  
void dfs(int u,bool st[],int t)  
{  
 st[u]=1;  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = i^t,ver = e[i];  
 if(!st[ver]&&w[j])  
 dfs(ver,st,t);  
 }  
}  
  
int main()  
{  
 cin>>n>>m;  
 S = 0,T = n - 1;  
 memset(h, -1, sizeof h);  
 for(int i=0;i<m;i++)  
 {  
 int a,b,c;  
 cin>>a>>b>>c;  
 add(a,b,c);  
 }  
 dinic();  
 dfs(S,sst,0);  
 dfs(T,tst,1);  
 int res = 0;  
 for(int i=0;i<m\*2;i+=2)  
 if(!w[i]&&sst[e[i^1]]&&tst[e[i]])  
 res++;  
 cout<<res<<endl;  
 return 0;  
}

##### **最大流判断**

**判断从起点到终点是否有K条互相不相交的路径，每条路径只可以用一次**

**对于无向边和有向边都一样的**

**其中无向图建立残留网络时，可以将相同方向的路径容量合并，则就是建立两条正向容量为c，反向容量也为c的边**

**网络流建图**

* **将其中所有边的容量都设为1，然后开始建立残留网络**
* **接下来跑一个最大流，若最大流>=K则一定有解**

##### **拆点**

**网络流中拆点，特指将点拆开为入点与出点，以此来限制经过点的流量，从而达到限制点的使用次数**

**网络流建图**

* **将点拆为入点与出点**
* **从入点向出点连接一条容量为限制的边**

#### 最小割

性质：**最小割=最大流，同时两个割集之间的边，正向都是满流，反向都为0流。**

##### 直接应用

**求最小权值和边割集**

**若其中有负权边直接加上，然后去掉负权边，直接求最小割，两者相加即为答案**

##### 平面图转最短路

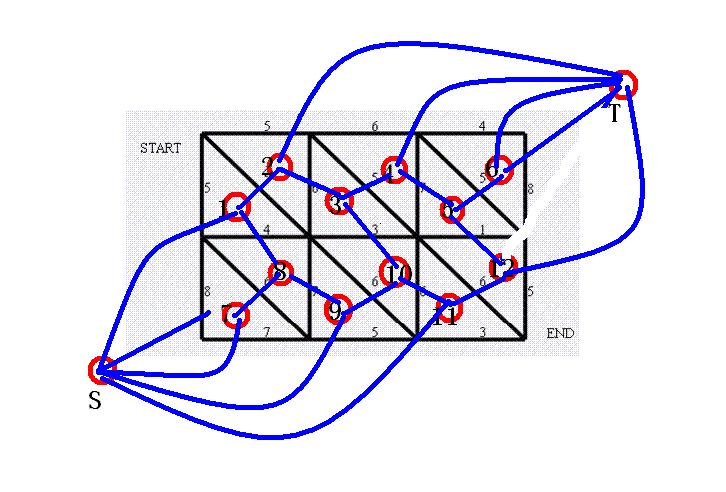
**平面图最小割=平面图最大流=对偶图的最短路**

BZOJ狼抓兔子

**我们一看就是最小割**

**但是如果直接建图跑Dinic会炸掉，所以我们建立对偶图。**

对偶图的建立，直接看这张图。



##### 最大权闭合图

**点有点权，求满足闭合图的最大权**

**闭合图，一些点构成的点集，其中没有点的边是指向集合外的点**

**实际应用中，即为，想取某些正权点的时候，必须取一些负权点**

**建图方式**

* **从源点向所有的正权点，连接一条容量为其正权权值的边**
* **从所有负权点向汇点，连接一条容量为其负权绝对值的边**
* **其中所有其他连接关系，连接容量为无穷的边**

**跑出来的答案即为，所有正权点点权和 - 最小割**

**求方案**

* **在残留网络中，从源点沿着f[i]>0的边搜索，能搜到的点都属于S，其余的属于T，横跨S->T的边是割边**
* **能搜到的属于S的点都是我们要选的点**

##### 最大密度子图

**其概念针对无向图而言**

**其外部是套了一层01分数规划**

**点和边的权值都为1**

我们要求的东西是，。则由01分数规划，推导后，我们想要求得东西就是最小化的

定义为点的度数，其中m为大于的值。

我们先说建图，再说最后答案，怎么求

* **从源点向所有点连一条容量为m的边**
* **从所有点向汇点连一条容量为的边**
* **原有图中所有边的容量连接为1**

**记住这式子**

**则，我们就是二分g，求最小割**

**点权值都为1，边的权值0**

相对于上一种情况，我们只需要将改变为点上的边权之和，然后把图中所有边的容量变为对应边权即可。

**点也有权值，边的权值0**

**设点权为**

这种就相对麻烦，我们需要改变的较多，但对比理解就好。

此时我们求的式子就变为了，则我们最小化的式子也变化了为

其中V1代表的是，**点权之和**。V2代表的是，**点数**。

此时，我们还是说相对于第一种情况，我们变化的。

1. **将改变为点上的边权之和，然后把图中所有边的容量变为对应边权。**
2. **从所有点向汇点连接的容量变为**

此时式子就变化了。

##### 最小权点覆盖

**其中点权为非负的**

**如果是负的直接选上就好**

**建图**

* **从源点向左边的点连接容量为点权的边**
* **从右边的点向汇点连接容量为点权的边**
* **中间的边容量为无穷**

**确定方案**

* **扫描残留网络的正向边，若边的容量为0，其不是S和T的端点，就为我们选择的点(错误)**
* **正确的应该是**
* **在残留网络中，从源点沿着f[i]>0的边搜索，能搜到的点都属于S，其余的属于T，横跨S->T的边是割边**

##### 最大权独立集

**最大权独立集 = 点权总权值 - 最小权点覆盖集**

##### 综合应用

**求删除最少几个点，可以将图变为不连通**

* **将点拆开，从入点向出点连一条容量为1的边**
* **其余边正常保留，容量为无穷**
* **枚举源点与汇点，取其中最小割即为答案**

memset(h,-1,sizeof h);idx = 0;  
for(int i=0;i<n;i++) add(i,i+n,1);  
for(int i=0;i<m;i++)  
{  
 int a,b;scanf(" (%d,%d)",&a,&b);  
 add(a+n,b,INF),add(b+n,a,INF);  
}  
int ans = n;  
for(int i=0;i<n;i++)  
 for(int j=i+1;j<n;j++)  
 {  
 S = i + n,T = j;  
 for(int k=0;k<idx;k+=2)  
 f[k] += f[k^1],f[k^1] = 0;  
 ans = min(ans,dinic());  
 }  
printf("%d\n",ans);

#### 费用流

**其中包含，最少费用最大流，最大费用最大流**

**模板**

**把EK里的bfs换成spfa**

**因为用了spfa，因此图中要无负权回路**

#include<bits/stdc++.h>  
  
using namespace std;  
  
const int N = 5010,M = 100010,INF = 0x3f3f3f3f;  
int h[N],ne[M],e[M],f[M],w[M],idx;  
int q[N],d[N],minT[N],pre[N];  
bool st[N];  
int n,m,S,T;  
  
void add(int a,int b,int c,int d)  
{  
 e[idx] = b,ne[idx] = h[a],w[idx] = d,f[idx] = c,h[a] = idx++;  
 e[idx] = a,ne[idx] = h[b],w[idx] = -d,f[idx] = 0,h[b] = idx++;  
}  
  
bool spfa()  
{  
 int hh = 0,tt = 1;  
 memset(d,0x3f,sizeof d);  
 memset(minT,0,sizeof minT);  
 q[tt++] = S,d[S] = 0,minT[S] = INF;  
 while(hh!=tt)  
 {  
 int t = q[hh++];  
 if(hh==N) hh = 0;  
 st[t] = 0;  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(f[i]&&d[j]>d[t]+w[i])  
 {  
 d[j] = d[t] + w[i];  
 pre[j] = i;  
 minT[j] = min(minT[t],f[i]);  
 if(!st[j]){  
 q[tt++] = j;  
 if(tt==N) tt = 0;  
 st[j] = 1;  
 }  
 }  
 }  
 }  
 return minT[T]>0;  
}  
  
void ek(int &flow,int &cost)  
{  
 flow = cost = 0;  
 while(spfa())  
 {  
 int t = minT[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()  
{  
 scanf("%d%d%d%d",&n,&m,&S,&T);  
 memset(h,-1,sizeof h);  
 while(m--)  
 {  
 int a,b,c,d;scanf("%d%d%d%d",&a,&b,&c,&d);  
 add(a,b,c,d);  
 }  
 int flow = 0,cost = 0;  
 ek(flow,cost);  
 printf("%d %d\n",flow,cost);  
 return 0;  
}

##### 最大（小）权匹配

**建图方式**

* **从源点向所有左边的点连接一条，费用为0，容量为1的边**
* **从所有右边的点向汇点连接一条费用为0，容量为1的边**
* **中间所有的边，保留，其容量为1，费用为题目给的匹配值**

#include<bits/stdc++.h>  
using namespace std;  
const int N = 110,M = 5210,INF = 1e8;  
int h[N],e[M],ne[M],w[M],f[M],idx;  
int d[N],pre[N],minf[N];  
bool st[N];  
int n,S,T;  
  
void add(int a,int b,int c,int d)  
{  
 e[idx] = b,ne[idx] = h[a],w[idx] = d,f[idx] = c,h[a] = idx++;  
 e[idx] = a,ne[idx] = h[b],w[idx] = -d,f[idx] = 0,h[b] = idx++;  
}  
  
bool spfa()  
{  
 queue<int> q;  
 memset(d,0x3f,sizeof d);  
 minf[T] = 0;  
 d[S] = 0,q.push(S),minf[S] = INF,st[S] = 1;  
 while(q.size())  
 {  
 auto t = q.front();  
 q.pop();  
 st[t] = 0;  
 for(int i=h[t];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(f[i]&&d[j] > d[t] + w[i])  
 {  
 d[j] = d[t] + w[i];  
 pre[j] = i;  
 minf[j] = min(f[i],minf[t]);  
 if(!st[j])  
 {  
 st[j] = 1;  
 q.push(j);  
 }  
 }  
 }  
 }  
 return minf[T] > 0;  
}  
  
int EK()  
{  
 int cost = 0;  
 while(spfa())  
 {  
 int t = minf[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;  
 }  
 return cost;  
}  
  
int main()  
{  
 cin>>n;  
 memset(h, -1, sizeof h);  
 S = 0,T = 2\*n + 1;  
 for(int i=1;i<=n;i++)  
 {  
 add(S,i,1,0);  
 add(i+n,T,1,0);  
 }  
 for(int i=1;i<=n;i++)  
 for(int j=1;j<=n;j++)  
 {  
 int c;cin>>c;  
 add(i,j+n,1,c);  
 }  
   
 cout<<EK()<<endl;  
   
 for(int i=0;i<idx;i+=2)  
 {  
 f[i] += f[i^1];  
 f[i^1] = 0;  
 w[i] = -w[i],w[i^1] = -w[i^1];  
 }  
   
 cout<<-EK()<<endl;  
 return 0;  
}

##### 最大权不重合路径

**从m个起点到许多的终点是否存在m条路径满足**

**首先，建立虚拟源点向所有的起点连一条容量为1，费用为0的边，再从所有的终点向虚拟汇点也连一条容量为1，费用为0的边**

**点和边都不可重复走**

**建图方式**

* **保留其中所有边，容量为1，费用为0**
* **拆点，其中入点与出点之间的边，容量为1，费用为点权**

**点可以走多次**

**建图方式**

* **保留其中所有边，容量为无穷，费用为0**
* **拆点，其中入点与出点之间的边，容量为1，费用为点权**

**点与边都可以走多次**

**建图方式**

* **保留其中所有边，容量为无穷，费用为0**
* **拆点，其中入点与出点之间的边，容量为无穷，费用为点权**

##### 网格图模型

**K取方格数**

**网格图，从(1,1)走到(n,m)总共k条路径中，所有路径权值的最大是多少，多次经过一个点，只能获得一次权值**

**其跟上个模型最后一个情况最大区别在于，上个的点权我们可以一直取，而本题的点权我们只能取一次**

**建图方式**

* **从虚拟源点向(1,1)连接一条容量为k，费用为0的边，并从(n,m)向虚拟汇点连接一条容量是k，费用是0的边**
* **按照其方格间能走的边，建立容量为无穷，费用为0的边**
* **拆点，建立一条入点与出点之间的边，容量为无穷，费用为0，再建立一条容量为1，费用为点权的边**

**深海机器人模型**

**有多个起点，多个终点，没有点权，边权只能取一次，问从所有起点到所有终点的最大权值和**

**跟上一题差不多，不过这次权值只能取一次放到了边上**

* **从虚拟源点从所有起点连一条容量为点上初始有的机器人数量，费用为0，从所有终点想虚拟汇点连接一条容量点上对应容积，费用为0的边**
* **保留其中可以走的边，建立一条容量为无穷，费用为0的边，再建立一条容量为1，费用为对应价值的边**

### 树

#### 求树的直径

#include<bits/stdc++.h>  
using namespace std;  
const int N = 1e4 + 10,M = N\*2;  
int h[N],e[M],ne[M],idx;  
int dist[N];  
int n;  
  
void add(int a,int b)  
{  
 e[idx] = b,ne[idx] = h[a],h[a] = idx++;  
}  
  
void dfs(int u,int fa)  
{  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(j==fa) continue;  
 dist[j] = dist[u] + 1;  
 dfs(j,u);  
 }  
}  
  
int main()  
{  
 cin>>n;  
 memset(h,-1,sizeof h);  
 for(int i=0;i<n-1;i++)  
 {  
 int u,v;cin>>u>>v;  
 add(u,v),add(v,u);  
 }  
 dfs(1,-1);  
 int mx = 0,t;  
 for(int i=1;i<=n;i++)  
 if(dist[i]>mx)   
 {  
 mx = dist[i];  
 t = i;  
 }  
 memset(dist,0,sizeof dist);  
 dfs(t,-1);  
 mx = 0;  
 for(int i=1;i<=n;i++)  
 if(dist[i]>mx)   
 mx = dist[i];  
 cout<<mx<<endl;  
 return 0;  
}

#### 求无权树中与每个点距离和最小的点

模板题[会议](https://www.luogu.com.cn/problem/P1395)

这个思路很常用。先得到与根的距离关系等，再进行递推

此题也是，先得到距离1的距离和。再通过公式

因为，从u->j中，有sz[j]个点距离-1，而有n-sz[j]个点距离+1

#include<bits/stdc++.h>  
using namespace std;  
const int N = 5e4 + 10,M = N\*2,INF = 0x3f3f3f3f;  
int h[N],e[M],ne[M],idx;  
int f[N],sz[N];  
int n;  
  
void add(int a,int b)  
{  
 e[idx] = b,ne[idx] = h[a],h[a] = idx++;  
}  
  
void dfs(int u,int fa,int sum)  
{  
 sz[u] = 1;f[1] += sum;  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(j==fa) continue;  
 dfs(j,u,sum+1);  
 sz[u] += sz[j];  
 }  
}  
  
void dfs1(int u,int fa)  
{  
 for(int i=h[u];~i;i=ne[i])  
 {  
 int j = e[i];  
 if(j==fa) continue;  
 f[j] = f[u] - sz[j] + n - sz[j];  
 dfs1(j,u);  
 }  
}  
  
int main()  
{  
 cin>>n;  
 memset(h,-1,sizeof h);  
 for(int i=0;i<n-1;i++)  
 {  
 int a,b;cin>>a>>b;  
 add(a,b),add(b,a);  
 }  
 dfs(1,-1,0);  
 dfs1(1,-1);  
 int ans = INF,t;  
 for(int i=1;i<=n;i++)  
 if(ans>f[i])  
 {  
 ans = f[i];  
 t = i;  
 }  
 cout<<t<<" "<<ans<<endl;  
 return 0;  
}

#### **求树中x个点的生成树的权值和**

静态计算是动态计算的其中一种情况，我们直接说动态计算。

首先，我们将树中所有点跑出dfs序。

我们计算贡献时用的是同一个式子。

假设，我们求得是z的贡献，我们取的点时x，y。其满足dfn[z]<dfn[z]<dfn[y]

**我们算出来的是z的贡献的二倍**

插入一个新点时，我们分两种情况。

* 第一种情况，插入的点在当前已经插入的点的边界处，那我们直接取已经插入的任意两点（可以取同一个）。为了方便，我们直接去当前的两个边界。
* 第二种情况，插入的点的两个边界可以找到，则直接代入式子。

如果是删除一个点，则反向减去贡献。

我们来看看代码。

我们以树中有多种颜色，即，同时维护多个求生成树权值和为例。

struct Scmp//按dfs序从小到大排序。  
{  
 bool operator()(int x,int y) const  
 {  
 return dfn[x]<dfn[y];  
 }  
};  
set<int,Scmp> s[N];  
int dis(int u,int v)  
{  
 return dep[u] + dep[v] - 2\*dep[lca(u,v)];  
}  
void update(int x,int f)//x代表的是树中的节点，f代表的操作类型  
{  
 if(f==-1) s[c[x]].erase(x);//-1则将x从原本所待的集合中删除  
 auto it = s[c[x]].lower\_bound(x);//找到x的在集合中边界  
 if(s[c[x]].empty()) ans[c[x]] = 0;//若此时为空，则对应集合的值为0  
 else if(it==s[c[x]].begin()||it==s[c[x]].end())//若x此时位于边界位置，则我们取的为集合中边界的两点（最大最小）  
 {  
 ans[c[x]] += f\*dis(\*s[c[x]].begin(),x);  
 ans[c[x]] += f\*dis(\*s[c[x]].rbegin(),x);  
 ans[c[x]] -= f\*dis(\*s[c[x]].begin(),\*s[c[x]].rbegin());  
 }  
 else //否则，我们就找x附近的两点  
 {  
 auto nt = it,pre = --it;  
 ans[c[x]] += f\*dis(\*nt,x);  
 ans[c[x]] += f\*dis(\*pre,x);  
 ans[c[x]] -= f\*dis(\*pre,\*nt);  
 }  
 if(f==1) s[c[x]].insert(x);//1，则将x计入新的集合  
}

#### 用线段树维护树上多条路径之间的路径交，以及距离两个端点最远的点两个点

具体地，线段树每个节点维护u,v,du,dv，u,v是路径交，du,dv是分别是距离u,v最远的点。合并两个区间的过程是先对两个区间的路径求路径交，然后新的du,dv一定是两个区间中四个du,dv中的两个。

这样我们可以通过**倍增**找到距离多条路径的端点都最远的点。

对一段区间询问，先求出区间的u,v,du,dv，于是可以选择路径du−>dv上中间的位置作为车站，如果中间点不在路径交集上，那么让u,v中的一个作为车站。

#include<cstdio>  
#include<algorithm>  
#define pa pair<int,int>  
#define mp make\_pair  
using namespace std;  
const int N = 200005;  
struct Tree{ int u, v, du, dv; } T[N << 2];  
struct Edge{ int to, nxt; } e[N << 1];  
int head[N], dis[N], pos[N], Log[N], f[N][21], g[N << 1][21], tot, num, n, m;  
  
bool cmp(int x,int y) {  
 return dis[x] < dis[y];  
}  
int Min(int x,int y) {   
 return dis[x] < dis[y] ? x : y;  
}  
void add\_edge(int u,int v) {  
 ++tot; e[tot].to = v; e[tot].nxt = head[u]; head[u] = tot;  
}  
void dfs(int u) {  
 g[++num][0] = u; pos[u] = num;  
 for (int i = head[u]; i; i = e[i].nxt) {  
 int v = e[i].to;  
 if (v == f[u][0]) continue;  
 f[v][0] = u, dis[v] = dis[u] + 1, g[++num][0] = u;  
 dfs(v);  
 }  
}  
int LCA(int u,int v) {  
 if (u == v) return u;  
 u = pos[u], v = pos[v];  
 if (u > v) swap(u, v); u ++;  
 int k = Log[v - u + 1];  
 return Min(g[u][k], g[v - (1 << k) + 1][k]);  
}  
int getdis(int u,int v) {  
 return dis[u] + dis[v] - dis[LCA(u, v)] \* 2;  
}  
bool inchain(int x,int y,int z) {  
 int t = LCA(x, y), l1 = LCA(z, x), l2 = LCA(z, y);  
 if (l1 == t && l2 == z) return 1;  
 if (l2 == t && l1 == z) return 1;  
 return 0;  
}  
Tree operator + (const Tree &A,const Tree &B) {  
 Tree res;  
 int flag = true;  
 if (A.u == -1 || B.u == -1) flag = false;  
 if (flag && !inchain(A.u, A.v, LCA(B.u, B.v)) && !inchain(B.u, B.v, LCA(A.u, A.v))) flag = false;  
 if (!flag) {  
 res.u = res.v = res.du = res.dv = -1; return res;  
 }  
 int z[4];  
 z[0] = LCA(A.u, B.u), z[1] = LCA(A.u, B.v), z[2] = LCA(A.v, B.u), z[3] = LCA(A.v, B.v);  
 sort(z, z + 4, cmp);  
 res.u = z[3], res.v = z[2];  
 if (res.u == res.v) {  
 pa w[4];  
 w[0] = mp(getdis(A.u, res.u), A.u);  
 w[1] = mp(getdis(A.v, res.u), A.v);  
 w[2] = mp(getdis(B.u, res.u), B.u);  
 w[3] = mp(getdis(B.v, res.u), B.v);  
 sort(w, w + 4);  
 res.du = w[3].second, res.dv = w[2].second;  
 return res;  
 }  
 z[0] = A.du, z[1] = A.dv, z[2] = B.du, z[3] = B.dv;  
 int mx1 = -1, mx2 = -1;  
 for (int i = 0; i < 4; ++i) {  
 int d1 = getdis(z[i], res.u), d2 = getdis(z[i], res.v);  
 if (d1 < d2 && d1 > mx1) mx1 = d1, res.du = z[i];  
 else if (d2 < d1 && d2 > mx2) mx2 = d2, res.dv = z[i];  
 }  
 return res;  
}  
void build(int l,int r,int rt) {  
 if (l == r) {  
 scanf("%d%d", &T[rt].u, &T[rt].v); T[rt].du = T[rt].u, T[rt].dv = T[rt].v;  
 return ;  
 }  
 int mid = (l + r) >> 1;  
 build(l, mid, rt << 1);  
 build(mid + 1, r, rt << 1 | 1);  
 T[rt] = T[rt << 1] + T[rt << 1 | 1];  
}  
void update(int l,int r,int rt,int p,int u,int v) {  
 if (l == r) {  
 T[rt].u = T[rt].du = u, T[rt].v = T[rt].dv = v;  
 return ;  
 }  
 int mid = (l + r) >> 1;  
 if (p <= mid) update(l, mid, rt << 1, p, u, v);  
 else update(mid + 1, r, rt << 1 | 1, p, u, v);  
 T[rt] = T[rt << 1] + T[rt << 1 | 1];  
}  
Tree query(int l,int r,int rt,int L,int R) {  
 if (L <= l && r <= R) {  
 return T[rt];  
 }  
 int mid = (l + r) >> 1;  
 if (R <= mid) return query(l, mid, rt << 1, L, R);  
 else if (L > mid) return query(mid + 1, r, rt << 1 | 1, L, R);  
 else return query(l, mid, rt << 1, L, R) + query(mid + 1, r, rt << 1 | 1, L, R);  
}  
int find(int x,int y,int z) {  
 int lca = LCA(x, y), len = getdis(x, lca);  
 if (len < z) z = getdis(x, y) - z, x = y;  
 for (int i = Log[n]; ~i; --i)   
 if ((z >> i) & 1) x = f[x][i];  
 return x;  
}  
void Ask() {  
 int l, r;  
 scanf("%d%d", &l, &r);  
 Tree now = query(1, m, 1, l, r);  
 if (now.u == -1 && now.v == -1) { puts("-1"); return ; }  
 int d1 = getdis(now.du, now.u), d2 = getdis(now.dv, now.v), d = getdis(now.u, now.v);  
 if (d1 >= d2 + d) {  
 printf("%d\n", now.u); return ;  
 }  
 if (d2 >= d1 + d) {  
 printf("%d\n", now.v); return ;  
 }  
 int mid = (d1 + d2 + d) / 2;  
 if ((d1 + d2 + d) & 1)   
 printf("%d\n", min(find(now.du, now.dv, mid), find(now.du, now.dv, mid + 1)));  
 else   
 printf("%d\n", find(now.du, now.dv, mid));  
}  
void Change() {  
 int p, u, v;  
 scanf("%d%d%d", &p, &u, &v);  
 update(1, m, 1, p, u, v);  
}  
int main() {  
 scanf("%d%d", &n, &m);  
 for (int i = 1; i < n; ++i) {  
 int u, v;  
 scanf("%d%d", &u, &v);  
 add\_edge(u, v); add\_edge(v, u);  
 }  
 dfs(1);  
 Log[0] = -1;  
 for (int i = 1; i <= num; ++i) Log[i] = Log[i >> 1] + 1;  
 for (int j = 1; j <= Log[n]; ++j)   
 for (int i = 1; i <= n; ++i) f[i][j] = f[f[i][j - 1]][j - 1];  
 for (int j = 1; j <= Log[num]; ++j)   
 for (int i = 1; i <= num; ++i) g[i][j] = Min(g[i][j - 1], g[i + (1 << (j - 1))][j - 1]);  
 build(1, m, 1);  
 int Q, opt;  
 scanf("%d", &Q);  
 while (Q--) {  
 scanf("%d", &opt);  
 if (opt == 1) Ask();  
 else Change();  
 }  
 return 0;  
}

#### 树的重心的性质

1. 某个点是树的**重心**等价于它最大子树大小**不大于**整棵树大小的**一半**。
2. 树**至多有两个**重心。如果树有两个重心，那么它们**相邻**。此时树一定有**偶数**个节点，且可以被划分为两个大小相等的分支，每个分支各自包含一个重心。
3. 树中所有点到某个点的**距离和**中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。反过来，距离和最小的点一定是重心。

### 图

#### 无向图求三元环个数

时间复杂度，w是二进制位数

#include<bits/stdc++.h>  
  
using namespace std;  
typedef long long LL;  
const int N = 3010;  
  
char s[N][N];  
bitset<N> a[N];  
int n;  
  
int main()  
{  
 scanf("%d",&n);  
 for(int i=1;i<=n;i++)  
 {  
 scanf("%s",s[i]+1);  
 for(int j=i+1;j<=n;j++)  
 if(s[i][j]=='1')   
 a[i][j] = 1;  
 }  
 LL ans = 0;  
 for(int i=1;i<n;i++)  
 for(int j=i+1;j<=n;j++)  
 if(a[i][j])  
 ans += (a[i]&a[j]).count();  
 printf("%lld\n",ans);  
 return 0;  
}

#### Floyd的一些拓展用法

##### 无向图的最小环问题

#include<bits/stdc++.h>  
using namespace std;  
typedef long long LL;  
const LL inf = 1e18;  
LL n,m,u,v,w,ans = inf;  
LL dis[128][128];  
LL g[128][128];  
int main(){  
 cin>>n>>m;  
 for(int i=1;i<=n;i++)  
 for(int j=1;j<=n;j++)  
 if(i!=j) dis[i][j]=g[i][j]=inf;  
 for(int i=1;i<=m;i++){  
 cin>>u>>v>>w;  
 g[v][u]=g[u][v]=min(g[u][v],w);  
 dis[v][u]=dis[u][v]=min(dis[u][v],w);  
 }  
 for(int k=1;k<=n;k++){  
 for(int i=1;i<k;i++)  
 for(int j=i+1;j<k;j++)  
 ans = min(ans,dis[i][j]+g[i][k]+g[k][j]);  
 for(int i=1;i<=n;i++)  
 for(int j=1;j<=n;j++){  
 dis[i][j] = min(dis[i][j],dis[i][k]+dis[k][j]);  
 dis[j][i] = dis[i][j];  
 }  
   
 }  
 if(ans==inf)cout<<"No solution.";  
 else cout<<ans;  
 return 0;  
}

##### 有向图最小环问题

#include<bits/stdc++.h>  
using namespace std;  
const int N = 2e5 + 10;  
int n;  
int p[N],a[N],d[N];  
  
int find(int x)  
{  
 if(p[x]!=x)   
 {  
 int pre = p[x];  
 p[x]=find(p[x]);  
 d[x]+=d[pre];  
 }  
 return p[x];  
}  
  
int main()  
{  
  
 cin>>n;  
 for(int i=1;i<=n;i++)  
 {  
 cin>>a[i];  
 p[i]=i;  
 }  
  
 int ans = 0x3f3f3f3f;  
  
 for(int i=1;i<=n;i++)  
 {  
 int pa = find(i),pb = find(a[i]);  
 if(pa!=pb)   
 {  
 p[pa]=pb;  
 d[i]=d[a[i]]+1;  
 }  
 else ans=min(ans,d[i]+d[a[i]]+1);  
 }  
 cout<<ans<<endl;  
 return 0;  
}

#### 分层图与拆点

##### 区别与联系

**联系**

我们可以发现，拆点就是分层图的实现逻辑，其实两者是一个思想。都是一个点的状态表示不够了，我们对点进行的拆分。我们可以说分层图是拆点的一种实现。

**区别**

我们说拆点与分层图的最大不同，可能就是添加的状态的形式。

对于拆点而言，添加的状态一般都是具体的，比如方向，数量等。其转移时，并不是说一定会向下一层转移。比如，进来的方向是左，出必须向右或向下，其并没有具体的下一层。或者是，当前拥有的数量为c，而进入当前格子后会+d个数量，则就不是转移到+1的下一层。**因此其状态要不没有具体的下一层，要不就其转移就不是随着层数递增而进行的。**

而对于分层图而言，其新添加的状态就是，例如某些具体的对边的操作数，最短路上的相邻点与点之间的分层，随着状态的增加，其所有的转移都是在相邻的层与层之间进行的。**其转移就是随着状态层数递增而进行的。**