图论 2

网络流

最大流

EdmondsKarp 算法

每次通过 bfs 找到一条边数最少的增广路。

bfs 时间复杂度 $O(m)$，最多增广 $O(nm)$ 次，总时间复杂度 $O(nm^2)$。

```
const int MAXN=205,MAXM=5005,Inf=0x7fffffff;
struct Edge{
    int to,cap,next;
    Edge(int to=0,int cap=0,int next=0){
        this->to=to;
        this->cap=cap;
        this->next=next;
    }
}edge[MAXM<<1];
int head[MAXN],edge_cnt;
void Clear(){mem(head,-1);edge_cnt=0;}//边从0开始编号
void Insert(int u,int v,int c){
    edge[edge_cnt]=Edge(v,c,head[u]);
    head[u]=edge_cnt++;
    edge[edge_cnt]=Edge(u,0,head[v]);
    head[v]=edge_cnt++;
}
struct EdmondsKrap{
    int a[MAXN],p[MAXN];
    int Maxflow(int s,int t){
        int flow=0;
        while(true){
            mem(a,0);
            queue<int>q;
            a[s]=Inf;
            q.push(s);
            while(!q.empty()){
                int u=q.front();q.pop();
                for(int i=head[u];~i;i=edge[i].next){
                    int v=edge[i].to;
                    if(!a[v]&&edge[i].cap){
                        p[v]=i;
                        a[v]=min(a[u],edge[i].cap);
                        q.push(v);
                    }
```

```
                }
            }
            if(!a[t])
            return flow;
            for(int i=t;i!=s;i=edge[p[i]^1].to){
                edge[p[i]].cap-=a[t];
                edge[p[i]^1].cap+=a[t];
            }
            flow+=a[t];
        }
    }
};
```

Dinic 算法

每次通过 bfs 构造层次图，然后沿层次图进行 dfs

每次 bfs 使得源点到汇点的最小距离减 1，所以最多构图 $n-1$ 次。每次 dfs 时间复杂度 $O(nm)$总时间复杂度 $O(n^2m)$

如果所有边容量相等Dinic 算法时间复杂度变为 $O(\min(n^{\frac 23},m^{\frac 12})m)$

如果图为二分图Dinic 算法时间复杂度变为 $O(\sqrt nm)$

```
struct Dinic{
    int s,t;
    int pos[MAXN],vis[MAXN],dis[MAXN];
    bool bfs(int k){
        queue<int>q;
        q.push(s);
        vis[s]=k,dis[s]=0,pos[s]=head[s];
        while(!q.empty()){
            int u=q.front();q.pop();
            for(int i=head[u];~i;i=edge[i].next){
                int v=edge[i].to;
                if(vis[v]!=k&&edge[i].cap){
                    vis[v]=k,dis[v]=dis[u]+1,pos[v]=head[v];
                    q.push(v);
                    if(v==t)
                    return true;
                }
            }
        }
        return false;
    }
    int dfs(int u,int max_flow){
        if(u==t||!max_flow)
        return max_flow;
        int flow=0,temp_flow;
```

```
        for(int &i=pos[u];~i;i=edge[i].next){
            int v=edge[i].to;
if(dis[u]+1==dis[v]&&(temp_flow=dfs(v,min(max_flow,edge[i].cap)))){
                edge[i].cap-=temp_flow;
                edge[i^1].cap+=temp_flow;
                flow+=temp_flow;
                max_flow-=temp_flow;
                if(!max_flow)
                break;
            }
        }
        return flow;
    }
    int Maxflow(int s,int t){
        this->s=s;this->t=t;
        int ans=0,k=0;
        mem(vis,0);
        while(bfs(++k))
        ans+=dfs(s,Inf);
        return ans;
    }
};
```

ISAP 算法

总体思路类似 Dinic 算法，但没有多次 bfs 修改 dis而是动态修改。

时间复杂度同 Dinic 算法，常数更优。

```
struct ISAP{
    int s,t,n;
    int pos[MAXN],dis[MAXN],p[MAXN],num[MAXN];
    bool vis[MAXN];
    void bfs(){
        mem(vis,0);
        queue<int>q;
        q.push(t);
        vis[t]=true,dis[t]=0;
        while(!q.empty()){
            int u=q.front();q.pop();
            for(int i=head[u];~i;i=edge[i].next){
                int v=edge[i].to;
                if(!vis[v]&&edge[i^1].cap){
                    vis[v]=true,dis[v]=dis[u]+1;
                    q.push(v);
                }
            }
        }
        mem(num,0);
        _rep(i,1,n)
```

```
        pos[i]=head[i],num[dis[i]]++;
    }
    int augment(){
        int flow=Inf;
        for(int i=t;i!=s;i=edge[p[i]^1].to)
        flow=min(flow,edge[p[i]].cap);
        for(int i=t;i!=s;i=edge[p[i]^1].to){
            edge[p[i]].cap-=flow;
            edge[p[i]^1].cap+=flow;
        }
        return flow;
    }
    int Maxflow(int s,int t,int n){
        this->s=s;this->t=t;this->n=n;
        bfs();
        int ans=0,u=s;
        bool flag;
        while(dis[s]<n){
            if(u==t)
            ans+=augment(),u=s;
            flag=false;
            for(int &i=pos[u];~i;i=edge[i].next){
                int v=edge[i].to;
                if(dis[u]==dis[v]+1&&edge[i].cap){
                    flag=true,p[v]=i,u=v;
                    break;
                }
            }
            if(!flag){
                int d=n-1;
                for(int i=head[u];~i;i=edge[i].next){
                    if(edge[i].cap)
                    d=min(d,dis[edge[i].to]);
                }
                if(--num[dis[u]]==0)
                break;
                num[dis[u]=d+1]++;
                pos[u]=head[u];
                if(u!=s)
                u=edge[p[u]^1].to;
            }
        }
        return ans;
    }
};
```

最小割

求最小割等价于求最大流。对求完最大流后的残量网络中的点，根据是否还可以从源点到达划分为两个集

合，两个集合间的连边即为最小割。

唯一性：如果残量网络中不存在既不能从源点到达也不能到达汇点的点，则最小割唯一。

最小割树

洛谷p4897

题意

给定 n 个点 m 条边的无向图，多次询问两点间的最小割。

题解

给出一种建树方法，首先任意选择两点求两点间的最小割，用一条权值为最小割的边连接两点。

根据最小割将点集分为两部分(但求最小割时仍然计算全图的最小割)，递归上述过程直到只剩下唯一一个点。

得到的树具有一个神奇的性质：任意两点间的最小割等于树上两点间简单路径中边权的最小值。~~证明略~~

建树过程求了 $n-1$ 次最小割，故时间复杂度为 $O(n^3m)$

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <algorithm>
#include <string>
#include <sstream>
#include <cstring>
#include <cctype>
#include <cmath>
#include <vector>
#include <set>
#include <map>
#include <stack>
#include <queue>
#include <ctime>
#include <cassert>
#define _for(i,a,b) for(int i=(a);i<(b);++i)
#define _rep(i,a,b) for(int i=(a);i<=(b);++i)
#define mem(a,b) memset(a,b,sizeof(a))
using namespace std;
typedef long long LL;
inline int read_int(){
    int t=0;bool sign=false;char c=getchar();
    while(!isdigit(c)){sign|=c=='-';c=getchar();}
    while(isdigit(c)){t=(t<<1)+(t<<3)+(c&15);c=getchar();}
```

```
    return sign?-t:t;
}
inline LL read_LL(){
    LL t=0;bool sign=false;char c=getchar();
    while(!isdigit(c)){sign|=c=='-';c=getchar();}
    while(isdigit(c)){t=(t<<1)+(t<<3)+(c&15);c=getchar();}
    return sign?-t:t;
}
inline char get_char(){
    char c=getchar();
    while(c==' '||c=='\n'||c=='\r')c=getchar();
    return c;
}
inline void write(LL x){
    register char c[21],len=0;
    if(!x)return putchar('0'),void();
    if(x<0)x=-x,putchar('-');
    while(x)c[++len]=x%10,x/=10;
    while(len)putchar(c[len--]+48);
}
inline void space(LL x){write(x),putchar(' ');}
inline void enter(LL x){write(x),putchar('\n');}
const int MAXN=505,MAXM=1505,Inf=0x7fffffff;
struct Edge{
    int to,cap,next;
    Edge(int to=0,int cap=0,int next=0){
        this->to=to;
        this->cap=cap;
        this->next=next;
    }
}edge[MAXM<<1];
int head[MAXN],edge_cnt;
void Clear(){mem(head,-1);edge_cnt=0;}
void Insert(int u,int v,int c){
    edge[edge_cnt]=Edge(v,c,head[u]);
    head[u]=edge_cnt++;
    edge[edge_cnt]=Edge(u,c,head[v]);
    head[v]=edge_cnt++;
}
struct ISAP{
    int s,t,n;
    int pos[MAXN],dis[MAXN],p[MAXN],num[MAXN];
    bool vis[MAXN];
    void bfs(){
        mem(vis,0);
        queue<int>q;
        q.push(t);
        vis[t]=true,dis[t]=0;
        while(!q.empty()){
            int u=q.front();q.pop();
```

```
            for(int i=head[u];~i;i=edge[i].next){
                int v=edge[i].to;
                if(!vis[v]&&edge[i^1].cap){
                    vis[v]=true,dis[v]=dis[u]+1;
                    q.push(v);
                }
            }
        }
        mem(num,0);
        _rep(i,1,n)
        pos[i]=head[i],num[dis[i]]++;
    }
    int augment(){
        int flow=Inf;
        for(int i=t;i!=s;i=edge[p[i]^1].to)
        flow=min(flow,edge[p[i]].cap);
        for(int i=t;i!=s;i=edge[p[i]^1].to){
            edge[p[i]].cap-=flow;
            edge[p[i]^1].cap+=flow;
        }
        return flow;
    }
    int Maxflow(int s,int t,int n){
        this->s=s;this->t=t;this->n=n;
        bfs();
        int ans=0,u=s;
        bool flag;
        while(dis[s]<n){
            if(u==t)
            ans+=augment(),u=s;
            flag=false;
            for(int &i=pos[u];~i;i=edge[i].next){
                int v=edge[i].to;
                if(dis[u]==dis[v]+1&&edge[i].cap){
                    flag=true,p[v]=i,u=v;
                    break;
                }
            }
            if(!flag){
                int d=n-1;
                for(int i=head[u];~i;i=edge[i].next){
                    if(edge[i].cap)
                    d=min(d,dis[edge[i].to]);
                }
                if(--num[dis[u]]==0)
                break;
                num[dis[u]=d+1]++;
                pos[u]=head[u];
                if(u!=s)
                u=edge[p[u]^1].to;
            }
```

```
        }
        return ans;
    }
};
struct Gomory_Hu_Tree{
    ISAP solver;
    int n,idx[MAXN],temp[MAXN],color[MAXN],k;
    int Head[MAXN<<1],To[MAXN<<1],W[MAXN<<1],Next[MAXN<<1],tree_edge;
    int Min_cut[MAXN][MAXN];
    void insert(int u,int v,int w){
        Next[++tree_edge]=Head[u];
        To[tree_edge]=v;W[tree_edge]=w;
        Head[u]=tree_edge;
    }
    void dfs(int u){
        color[u]=k;
        for(int i=head[u];~i;i=edge[i].next){
            if(edge[i].cap&&color[edge[i].to]!=k)
            dfs(edge[i].to);
        }
    }
    void build(int lef,int rig){
        if(lef>=rig)
        return;
        for(int i=0;i<edge_cnt;i+=2)
        edge[i].cap=edge[i|1].cap=edge[i].cap+edge[i|1].cap>>1;//边的还原
        int cut=solver.Maxflow(idx[lef],idx[rig],n);
        insert(idx[lef],idx[rig],cut);insert(idx[rig],idx[lef],cut);
        ++k;dfs(idx[lef]);
        int pos1=lef-1,pos2=rig+1;
        _rep(i,lef,rig){
            if(color[idx[i]]==k)
            temp[++pos1]=idx[i];
            else
            temp[--pos2]=idx[i];
        }
        _rep(i,lef,rig)
        idx[i]=temp[i];
        build(lef,pos1);
        build(pos2,rig);
    }
    void dfs2(int u,int fa,int p,int w){
        Min_cut[p][u]=w;
        for(int i=Head[u];i;i=Next[i]){
            int v=To[i];
            if(v==fa)
            continue;
            dfs2(v,u,p,min(w,W[i]));
        }
    }
```

```
    void build(int n){
        this->n=n;
        _rep(i,1,n)
        idx[i]=i;
        build(1,n);
        _rep(i,1,n)
        dfs2(i,i,i,Inf);
    }
}GHT;
int main()
{
    Clear();
    int n=read_int(),m=read_int();
    while(m--){
        int u=read_int(),v=read_int(),c=read_int();
        Insert(u,v,c);
    }
    GHT.build(n);
    int q=read_int();
    while(q--){
        int u=read_int(),v=read_int();
        enter(GHT.Min_cut[u][v]);
    }
    return 0;
}
```

费用流

EdmondsKarp 算法

每次通过最短路算法找到一条费用最小的增广路增广，不能处理负环。

时间复杂度上界为 $O(nmf)$

```
struct Edge{
    int to,cap,w,next;
    Edge(int to=0,int cap=0,int w=0,int next=0){
        this->to=to;
        this->cap=cap;
        this->w=w;
        this->next=next;
    }
}edge[MAXM<<1];
int head[MAXN],edge_cnt;
void Clear(){mem(head,-1);edge_cnt=0;}//边从0开始编号
void Insert(int u,int v,int w,int c){
    edge[edge_cnt]=Edge(v,c,w,head[u]);
    head[u]=edge_cnt++;
    edge[edge_cnt]=Edge(u,0,-w,head[v]);
```

```
    head[v]=edge_cnt++;
}
struct EdmondsKrap{
    int a[MAXN],dis[MAXN],p[MAXN];
    bool inque[MAXN];
    void MCMF(int s,int t,int &flow,LL &cost){
        flow=0,cost=0;
        while(true){
            mem(a,0);mem(dis,127/3);
            queue<int> q;
            a[s]=Inf,dis[s]=0,inque[s]=true;
            q.push(s);
            while(!q.empty()){
                int u=q.front();q.pop();
                inque[u]=false;
                for(int i=head[u];~i;i=edge[i].next){
                    int v=edge[i].to;
                    if(edge[i].cap&&dis[u]+edge[i].w<dis[v]){
                        p[v]=i;
                        dis[v]=dis[u]+edge[i].w;
                        a[v]=min(a[u],edge[i].cap);
                        if(!inque[v]){
                            inque[v]=true;
                            q.push(v);
                        }
                    }
                }
            }
            if(!a[t])
            return;
            for(int i=t;i!=s;i=edge[p[i]^1].to){
                edge[p[i]].cap-=a[t];
                edge[p[i]^1].cap+=a[t];
            }
            flow+=a[t];
            cost+=1LL*a[t]*dis[t];
        }
    }
};
```

Dinic 算法

思路类似 Dinic 最大流算法，不同点是由于图中存在很多零环，所以 dfs 过程注意标记在路径上的节点防止无限循环。

算法效率略优于 EdmondsKarp 算法。

```
struct Dinic{
    int s,t;
```

```
    int pos[MAXN],dis[MAXN];
    bool inque[MAXN];
    bool spfa(){
        mem(dis,127/3);
        queue<int>q;
        q.push(s);
        inque[s]=true,dis[s]=0,pos[s]=head[s];
        while(!q.empty()){
            int u=q.front();q.pop();
            inque[u]=false;
            for(int i=head[u];~i;i=edge[i].next){
                int v=edge[i].to;
                if(dis[u]+edge[i].w<dis[v]&&edge[i].cap){
                    dis[v]=dis[u]+edge[i].w,pos[v]=head[v];
                    if(!inque[v]){
                        q.push(v);
                        inque[v]=true;
                    }
                }
            }
        }
        return dis[t]!=dis[0];
    }
    int dfs(int u,int max_flow){
        if(u==t||!max_flow)
        return max_flow;
        int flow=0,temp_flow;
        inque[u]=true;
        for(int &i=pos[u];~i;i=edge[i].next){
            int v=edge[i].to;
if(!inque[v]&&dis[u]+edge[i].w==dis[v]&&(temp_flow=dfs(v,min(max_flow,edge[i
].cap)))){
                edge[i].cap-=temp_flow;
                edge[i^1].cap+=temp_flow;
                flow+=temp_flow;
                max_flow-=temp_flow;
                if(!max_flow)
                break;
            }
        }
        inque[u]=false;
        return flow;
    }
    void MCMF(int s,int t,int &flow,LL &cost){
        this->s=s;this->t=t;
        cost=flow=0;
        int temp_flow;
        mem(inque,0);
        while(spfa()){
            temp_flow=dfs(s,Inf);
            flow+=temp_flow;
```

```
                cost+=1LL*temp_flow*dis[t];
            }
        }
};
```

上下界网络流

无源汇上下界可行流

问题描述

给定一张网络流图，每条边的流量必须在 $[L_i,R_i]$ 的范围内，且每个点满足流量平衡。

解决方案

现给每条边 L_i 的流量，则每条边变为容量为 R_i-L_i 的普通边。

考虑每个点，设每个点的流入量减流出量为 w

设置虚拟超级源和超级汇。

若 $w\gt0$则超级源向该点连一条容量为 w 的有向边，若 $w\lt0$则该点向超级汇连一条容量为 $-w$ 的有向边。

然后跑最大流，如果超级源的每条边均满载，则存在可行流，每条边(虚拟边除外)实际流量为当前流量 $+$ 流量下界，否则问题无解。

事实上强制给每条边下界等大的流量将导致部分点流量不平衡，设立虚拟超级源和超级汇的目的是调整流量。

由于每条虚拟边没有流量，所以某个节点接受来自虚拟超级源的流量实际上没有接受，等效于真实情况下流入量减少了等大的流量。

同理，某个节点向虚拟超级汇输送的流量实际上没有输送，等效于真实情况下流出量减少了等大的流量。

有源汇上下界可行流

问题描述

给定一张网络流图，每条边的流量必须在 $[L_i,R_i]$ 的范围内，且除源点汇点外每个点满足流量平衡。

解决方案

从汇点连一条容量无限大的边到源点(注意区分题目给定源/汇点和虚拟源/汇点)，这样使得汇点和源点在形式上满足流量平衡。

接下来的问题就转化为了无源汇上下界可行流问题，最后汇点到源点的边的流量即为源点到汇点的真实流量。

有源汇上下界最大流

问题描述

在有源汇上下界的基础上要求源点到汇点的可行流流量最大。

解决方案

先跑一遍有源汇上下界可行流确认问题是否有解，然后在残量网络中求源点到汇点的最大流即可。

注意之前从汇点到源点的边在计算最大流时必然可以回流，所以不需要额外计算贡献。

有源汇上下界最小流

问题描述

在有源汇上下界的基础上要求源点到汇点的可行流流量最小。

解决方案

先跑一遍有源汇上下界可行流确认问题是否有解，然后记录之前从汇点到源点的边的流量，并删除该边与其反向边。

然后在残量网络中求汇点到源点的最大流，答案为之前的可行流减去当前的最大流。

例题

洛谷p5192

题意

一个摄影师□n 天时间□m 个取材对象。

要求 n 天后第 x 个取材对象至少需要取材 G_x 张照片。

第 k 天可以取材 C_k 个对象，总共可以取材 D_k 张照片，当天每个可取材对象的取材照片数量必须在区间 $[L_{k_i},R_{k_i}]$ 内。

问是否存在满足条件的方案，如果存在输出总共可以取材的最大照片数，如果不存在输出 -1。

数据范围 $n\le 365,m\le 1000,C_k\le 300$□保证计算结果在 int 范围内。

题解

把天和取材对象都当成一个节点，建有源上下界网络流图。

n 天后第 x 个取材对象至少需要取材 G_x 张照片可以理解为第 x 个取材对象代表节点向汇点连一条上下界为 $[G_x,\inf]$ 的边。

第 k 天总共可以取材 D_k 张照片可以理解为源点向第 k 天代表节点连一条上下界为 $[0,D_k]$ 的边。

第 k 天可取材对象的取材照片数量必须在区间 $[L_{k_i},R_{k_i}]$ 内可以理解为第 k 天代表节点向第 k_i 个取材对象连一条上下界为 $[L_{k_i},R_{k_i}]$ 的边。

最后求一下有源汇上下界的最大流即可。

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <algorithm>
#include <string>
#include <sstream>
#include <cstring>
#include <cctype>
#include <cmath>
#include <vector>
#include <set>
#include <map>
#include <stack>
#include <queue>
#include <ctime>
#include <cassert>
#define _for(i,a,b) for(int i=(a);i<(b);++i)
#define _rep(i,a,b) for(int i=(a);i<=(b);++i)
#define mem(a,b) memset(a,b,sizeof(a))
using namespace std;
typedef long long LL;
inline int read_int(){
    int t=0;bool sign=false;char c=getchar();
    while(!isdigit(c)){sign|=c=='-';c=getchar();}
    while(isdigit(c)){t=(t<<1)+(t<<3)+c-48;c=getchar();}
    return sign?-t:t;
}
inline LL read_LL(){
    LL t=0;bool sign=false;char c=getchar();
    while(!isdigit(c)){sign|=c=='-';c=getchar();}
    while(isdigit(c)){t=(t<<1)+(t<<3)+c-48;c=getchar();}
    return sign?-t:t;
}
inline char get_char(){
```

```
    char c=getchar();
    while(c==' '||c=='\n'||c=='\r')c=getchar();
    return c;
}
inline void write(LL x){
    register char c[21],len=0;
    if(!x)return putchar('0'),void();
    if(x<0)x=-x,putchar('-');
    while(x)c[++len]=x%10,x/=10;
    while(len)putchar(c[len--]+48);
}
inline void space(LL x){write(x),putchar(' ');}
inline void enter(LL x){write(x),putchar('\n');}
const int MAXN=2005,MAXM=2e5+5,Inf=0x7fffffff;
struct Edge{
    int to,cap,next;
    Edge(int to=0,int cap=0,int next=0){
        this->to=to;
        this->cap=cap;
        this->next=next;
    }
}edge[MAXM<<1];
int head[MAXN],edge_cnt;
void Clear(){mem(head,-1);edge_cnt=0;}//边从0开始编号
void Insert(int u,int v,int c){
    edge[edge_cnt]=Edge(v,c,head[u]);
    head[u]=edge_cnt++;
    edge[edge_cnt]=Edge(u,0,head[v]);
    head[v]=edge_cnt++;
}
struct ISAP{
    int s,t,n;
    int pos[MAXN],dis[MAXN],p[MAXN],num[MAXN];
    bool vis[MAXN];
    void bfs(){
        mem(vis,0);
        queue<int>q;
        q.push(t);
        vis[t]=true,dis[t]=0;
        while(!q.empty()){
            int u=q.front();q.pop();
            for(int i=head[u];~i;i=edge[i].next){
                int v=edge[i].to;
                if(!vis[v]&&edge[i^1].cap){
                    vis[v]=true,dis[v]=dis[u]+1;
                    q.push(v);
                }
            }
        }
        mem(num,0);
        _rep(i,1,n)
```

```
        pos[i]=head[i],num[dis[i]]++;
    }
    int augment(){
        int flow=Inf;
        for(int i=t;i!=s;i=edge[p[i]^1].to)
        flow=min(flow,edge[p[i]].cap);
        for(int i=t;i!=s;i=edge[p[i]^1].to){
            edge[p[i]].cap-=flow;
            edge[p[i]^1].cap+=flow;
        }
        return flow;
    }
    int Maxflow(int s,int t,int n){
        this->s=s;this->t=t;this->n=n;
        bfs();
        int ans=0,u=s;
        bool flag;
        while(dis[s]<n){
            if(u==t)
            ans+=augment(),u=s;
            flag=false;
            for(int &i=pos[u];~i;i=edge[i].next){
                int v=edge[i].to;
                if(dis[u]==dis[v]+1&&edge[i].cap){
                    flag=true,p[v]=i,u=v;
                    break;
                }
            }
            if(!flag){
                int d=n-1;
                for(int i=head[u];~i;i=edge[i].next){
                    if(edge[i].cap)
                    d=min(d,dis[edge[i].to]);
                }
                if(--num[dis[u]]==0)
                break;
                num[dis[u]=d+1]++;
                pos[u]=head[u];
                if(u!=s)
                u=edge[p[u]^1].to;
            }
        }
        return ans;
    }
};
int id1[MAXN],id2[MAXN],w[MAXN];
void Insert2(int u,int v,int lef,int rig){
    edge[edge_cnt]=Edge(v,rig-lef,head[u]);
    head[u]=edge_cnt++;w[u]-=lef;
    edge[edge_cnt]=Edge(u,0,head[v]);
```

```
    head[v]=edge_cnt++;w[v]+=lef;
}
ISAP solver;
int main()
{
    int kase=0,n,m,s_0=1,s=2,t_0=3,t=4;
    while(~scanf("%d%d",&n,&m)){
        if(kase++)puts("");
        Clear();mem(w,0);
        int pos_id=4;
        _rep(i,1,n)
        id1[i]=++pos_id;
        _for(i,0,m){
            id2[i]=++pos_id;
            Insert2(id2[i],t,read_int(),Inf);
        }
        _rep(i,1,n){
            int c=read_int(),d=read_int(),t,l,r;
            Insert2(s,id1[i],0,d);
            _for(j,0,c){
                t=read_int(),l=read_int(),r=read_int();
                Insert2(id1[i],id2[t],l,r);
            }
        }
        Insert(t,s,Inf);
        int sum=0;
        _rep(i,1,pos_id){
            if(w[i]>0){
                Insert(s_0,i,w[i]);
                sum+=w[i];
            }
            else if(w[i]<0)
            Insert(i,t_0,-w[i]);
        }
        if(solver.Maxflow(s_0,t_0,pos_id)<sum)
        puts("-1");
        else
        enter(solver.Maxflow(s,t,pos_id));
    }
    return 0;
}
```

图匹配

二分图最大匹配

匈牙利算法

从未盖点开始依次经过非匹配边，匹配边，非匹配边……所得的的路径称为交替路。

如果交替路的终点也为未盖点，则得到了增广路。易知增广路的非匹配边比匹配边多一条，于是将非匹配边与匹配边互换实现增广。

每个点增广时间复杂度为 $O(m)$，故总时间复杂 $O(nm)$。

```
struct KM{
    int match[MAXN],vis[MAXN];
    bool dfs(int u,int k){
        if(vis[u]==k)
        return false;
        vis[u]=k;
        for(int i=head[u];i;i=edge[i].next){
            int v=edge[i].to;
            if(!match[v]||dfs(match[v],k))
            return match[v]=u,true;
        }
        return false;
    }
    int get_pair(int n){
        mem(match,0);mem(vis,0);
        int ans=0;
        _rep(i,1,n)
        ans+=dfs(i,i);
        return ans;
    }
}
```

模型转换

设二分图两部分为 X，Y。

建立超级源与超级汇，超级源向 X 部连一条容量为 1 的单向边，Y 部向超级汇连一条容量为 1 的单向边。

原图中 X 部、Y 部之间的连边改为容量为 1 的单向边，最后跑 Dinic 算法取最大流即可。

时间复杂度 $O(\sqrt nm)$。

性质

1. 二分图的最小顶点覆盖数等于二分图的最大匹配数
2. 二分图的最大独立集数等于节点数减最大匹配数

二分图最大权完美匹配

KM 算法

dfs 版本

先给出一些概念。

- 可行顶标：给那个点一个顶标函数 $l(i)$，满足 $u\in \text{X},v\in \text{Y},l(u)+l(v)\ge w(u,v)$。
- 相等子图：只保留原图中 $u\in \text{X},v\in \text{Y},l(u)+l(v)=w(u,v)$ 的边构成的子图。

容易得到结论：相等子图的完美匹配是原图的最大权完美匹配。

于是算法的重点在于找到合适的顶标函数来构造满足可以完美匹配的相等子图。

不妨设 $\text{lx}(i)$ 表示左部顶标，$\text{ly}(i)$ 表示右部顶标。

首先初始化 $\text{lx}(i)$ 为节点 i 连出的最大边权，$\text{ly}(i)$ 为 0，该构造满足 $\text{lx}(u)+\text{ly}(v)\ge w(u,v)$。

考虑依次为左部每个点寻找匹配。

对左部的某点 u 考虑 dfs 沿着当前相等子图的边寻找增广路，如果可以找到增广路，则该次匹配结束。

否则将 dfs 过程中访问的左部点集合记为 S，右部点记为 T，令 $\overline S=X-S,\overline T=Y-T$。

考虑向相等子图中加入新边，同时不破坏顶标函数的性质。考虑寻找某个合适的 a，将 S 中每个点顶标减小 a，T 中每个点顶标增加 a。

本次修改对两端点属于 S 和 T 或 $\overline S$ 和 $\overline T$ 的边无影响。

对两端点属于 $\overline S$ 和 T 的边 $\text{lx}(u)+\text{ly}(v)$ 值增大，所以仍然满足 $\text{lx}(u)+\text{ly}(v)\ge w(u,v)$。

对两端点属于 S 和 $\overline T$ 的边，同时由于 S 的顶标减小，更容易加入相等子图中。

现在考虑 a 的取值，首先 a 必须不小于 $\{\min(\text{lx}(u)+\text{ly}(v)-w(u,v))|u\in S,v\in \overline T\}$。

否则不能满足 $u\in \text{X},v\in \text{Y},l(u)+l(v)\ge w(u,v)$ 破坏了顶标函数的性质。

同时如果 $a\lt \{\min(\text{lx}(u)+\text{ly}(v)-w(u,v))|u\in S,v\in \overline T\}$ 则不会有新边加入。

所以只能有 $a=\{\min(\text{lx}(u)+\text{ly}(v)-w(u,v))|u\in S,v\in \overline T\}$。

顶标修改后重复上述过程，直到找到增广路。由于每次修改顶标能保证 S 集合元素至少增加一个，所以最多修改 $O(n)$ 次顶标。

每次 dfs 寻找增广路算法时间复杂度 $O(n^2)$，暴力计算 a 时间复杂度 $O(n^2)$。

所以每个点的匹配复杂度为 $O(n^3)$，总时间复杂度 $O(n^4)$。

一个优化是动态维护 $lack(v)=\{\min(\text{lx}(u)+\text{ly}(v)-w(u,v))|u\in S\}$ 这样计算 a 的时间

复杂度优化为 $O(n)$

考虑 dfs 时间复杂度很难跑满 $O(n^2)$所以算法的平均时间复杂度为 $O(n^3)$

```
struct KM{
    int n,w[MAXN][MAXN],lx[MAXN],ly[MAXN],lack[MAXN];//注意提前把不存在的w设为-Inf
    int match[MAXN],visx[MAXN],visy[MAXN];
    bool dfs(int u,int k){
        visx[u]=k;
        _rep(v,1,n){
            if(visy[v]==k)
            continue;
            int t=lx[u]+ly[v]-w[u][v];
            if(!t){
                visy[v]=k;
                if(!match[v]||dfs(match[v],k))
                return    match[v]=u,true;
            }
            else
            lack[v]=min(lack[v],t);
        }
        return false;
    }
    int get_pair(int n){
        this->n=n;int k=0;
        mem(ly,0);mem(match,0);mem(visx,0);mem(visy,0);
        _rep(i,1,n){
            lx[i]=-Inf;
            _rep(j,1,n)
            lx[i]=max(lx[i],w[i][j]);
        }
        _rep(i,1,n){
            _rep(j,1,n)
            lack[j]=Inf;
            while(true){
                if(dfs(i,++k))
                break;
                int a=Inf;
                _rep(j,1,n){
                    if(visy[j]!=k)
                    a=min(a,lack[j]);
                }
                _rep(j,1,n){
                    if(visx[j]==k)
                    lx[j]-=a;
                    if(visy[j]==k)
                    ly[j]+=a;
                    else
                    lack[j]-=a;
```

```
                }
            }
        }
        int ans=0;
        _rep(i,1,n)
        ans+=w[match[i]][i];
        return ans;
    }
}
```

bfs 版本

每次修改顶标新增边后重新 dfs 将导致代码效率低下。

考虑将算法过程中的 dfs 改为 bfs时间复杂度可优化为 $O(n^3)$

```
struct KM{
    int n,w[MAXN][MAXN],lx[MAXN],ly[MAXN],lack[MAXN],p[MAXN];//注意提前把不存在的w设为-Inf
    int linkx[MAXN],linky[MAXN],visx[MAXN],visy[MAXN];
    void augment(int pos){
        int temp;
        while(pos){
            temp=linkx[p[pos]];
            linkx[p[pos]]=pos;
            linky[pos]=p[pos];
            pos=temp;
        }
    }
    void bfs(int s,int k){
        queue<int>q;
        _rep(i,1,n)
        lack[i]=Inf;
        q.push(s);
        while(true){
            while(!q.empty()){
                int u=q.front();q.pop();
                visx[u]=k;
                _rep(v,1,n){
                    if(visy[v]==k)
                    continue;
                    if(lx[u]+ly[v]-w[u][v]<lack[v]){
                        lack[v]=lx[u]+ly[v]-w[u][v];p[v]=u;
                        if(!lack[v]){
                            visy[v]=k;
                            if(!linky[v])
                            return augment(v);
                            else
                            q.push(linky[v]);
                        }
```

```
                    }
                }
            }
            int a=Inf;
            _rep(i,1,n) if(visy[i]!=k)
            a=min(a,lack[i]);
            _rep(i,1,n){
                if(visx[i]==k)lx[i]-=a;
                if(visy[i]==k)ly[i]+=a;
                else lack[i]-=a;
            }
            _rep(i,1,n){
                if(visy[i]!=k&&!lack[i]){
                    visy[i]=k;
                    if(!linky[i])
                    return augment(i);
                    else
                    q.push(linky[i]);
                }
            }
        }
    }
    int get_pair(int n){
        this->n=n;int k=0;
        mem(ly,0);mem(linkx,0);mem(linky,0);mem(visx,0);mem(visy,0);
        _rep(i,1,n){
            lx[i]=-Inf;
            _rep(j,1,n)
            lx[i]=max(lx[i],w[i][j]);
        }
        _rep(i,1,n)
        bfs(i,i);
        int ans=0;
        _rep(i,1,n)
        ans+=w[linky[i]][i];
        return ans;
    }
}
```

二分图最大权匹配

考虑在右部图添加虚点保证右部图点数不小于左部图，然后在每对没有连边的左右部图点间添加权值为 0 的虚边，最后跑 KM 算法即可。

二分图最大权最大匹配

同样考虑在右部图添加虚点保证右部图点数不小于左部图，然后在每对没有连边的左右部图点间添加权值为 $-\inf$ 的虚边，最后跑 KM 算法即可。

需要保证答案绝对值一定小于 $\inf$同时 $n\times \inf$ 不溢出。

一般图最大匹配

带花树算法

把一般图转化为二分图，再套用 KM 算法解决。而只要消去一般图的所有奇环，就可以将一般图转化为二分图。

考虑奇环中有 $2k+1$ 个点，最多有 k 条匹配边，所以必然有一个点与其他点间不存在匹配边，而该点可以与奇环外的其他点匹配。

所以可以考虑先将奇环缩为一个点，并使用并查集维护。

具体算法仅改出代码和部分注释供参考。

```
struct KM{
    int n,link[MAXN],pre[MAXN],color[MAXN];
    int fa[MAXN],cyc_id[MAXN],cyc_cnt;
    queue<int> q;
    int Find(int x){return x==fa[x]?x:fa[x]=Find(fa[x]);}//寻找花根
    int LCA(int x,int y){//寻找x,y的总花根
        ++cyc_cnt;
        x=Find(x),y=Find(y);
        while(cyc_id[x]!=cyc_cnt){
            cyc_id[x]=cyc_cnt;//实质上是染色
            x=Find(pre[link[x]]);//跳到x的上一个花根
            if(y)swap(x,y);//如果y没有跳到根顶,则切换到y
        }
        return x;
    }
    void shrink(int x,int y,int p){//将环缩为以p为根的花
        while(Find(x)!=p){
            pre[x]=y,y=link[x];//奇环内部没有方向，所以pre需要补充成双向的
            if(color[y]==2){
                q.push(y);//现在白点也可以作为出点向外寻找增广路
                color[y]=1;
            }
            fa[x]=fa[y]=p;//修改花根
            x=pre[y];
        }
    }
    void update(int x){//从未配白点开始增广
        if(!x)
        return;
        update(link[pre[x]]);
        link[x]=pre[x],link[pre[x]]=x;//将未配边改为一配边
    }
    bool augment(int s){//尝试增广
        _rep(i,1,n)color[i]=pre[i]=0,fa[i]=i;
```

```
        color[s]=1;
        while(!q.empty())q.pop();
        q.push(s);
        while(!q.empty()){
            int u=q.front();q.pop();
            for(int i=head[u];i;i=edge[i].next){
                int v=edge[i].to;
                if(color[v]==1){//发现奇环
                    if(Find(u)==Find(v))//已经缩成环了就不必处理
                    continue;
                    int p=LCA(u,v);
                    shrink(u,v,p);shrink(v,u,p);//将通往花根的左右两条路径都修改
                }
                else if(!color[v]){
                    pre[v]=u,color[v]=2;
                    if(!link[v]){//找到未配白点
                        update(v);
                        return true;
                    }
                    else{
                        color[link[v]]=1;
                        q.push(link[v]);
                    }
                }
            }
        }
        return false;
    }
    int get_pair(int n){
        this->n=n;
        int ans=0;
        for(int i=1;i<edge_cnt;i+=2){//预处理加速
            int u=edge[i].to,v=edge[i+1].to;
            if(!link[u]&&!link[v])
            link[u]=v,link[v]=u,ans++;
        }
        _rep(i,1,n){
            if(!link[i]&&augment(i))
            ans++;
        }
        return ans;
    }
}
```