Dinic

```
int n,m,F,D,S,T;
int e[M],ne[M],h[N],f[M],idx=0;
int cur[M],q[N],d[N];

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

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

bool bfs(){
    memset(d,-1,sizeof d);
    int hh = 0, tt = 0;
    q[0] = S, cur[S] = h[S], d[S] = 0;

    while(hh <= tt){
        int u = q[hh ++];
        for(int i=h[u];~i;i=ne[i]){
            int ver = e[i];
            if(d[ver] == -1 && f[i]){
                d[ver] = d[u] + 1;
                cur[ver] = h[ver];
                if(ver == T) return true;
                q[++ tt] = ver;
            }
        }
    }
    return false;
}

int dinic(){
    int ans = 0, flow = 0;
    while(bfs()) while(flow = find(S,INF)) ans += flow;
    return ans;
}
```

双精度边Dinic

```
#define long double double
const int INF = 1e8;
int n,m,S,T;
int e[M],ne[M],h[N],idx=0;
double f[M]
int q[M],d[N],cur[N];

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

bool bfs(){
    int hh = 0 , tt = 0;
    memset(d,-1,sizeof d);
    q[0] = S,d[S] = 0,cur[S] = h[S];

    while(hh <= tt){
        int u = q[ hh ++ ];
        for(int i=h[u];~i;i=ne[i]){
            int ver = e[i];
            if(d[ver] == -1 && f[i]){
                d[ver] = d[u] + 1;
                cur[ver] = h[ver];
                if(ver == T) return true;
                q[++ tt] = ver;
            }
        }
    }
    return false;
}

double find(int u,double limit){
    if(u == T) return limit;
    double flow = 0; //准备从u点向后找最大的增广路上的残留容量

    for(int i = cur[u]; ~i && flow < limit;i = ne[i]){  //flow < limit 必加优化
        cur[u] = i;
        int ver = e[i];
        if(d[ver] == d[u] + 1 && f[i]){
            double t = find(ver,min(f[i],limit-flow));
            if(t<=0) d[ver] = -1; //如果此边到达不了终点，就直接删除
            f[i] -= t, f[i ^ 1] += t , flow += t;
        }
    }
    return flow;
}

double dinic(){
    double ans = 0, flow = 0;
    while(bfs()) while(flow = find(S,INF)) ans += flow;
    return ans;
}
```

最小费用最大流

```
int n,m,S,T;
```

```cpp
int h[N],e[M],w[M],f[M],ne[M],idx; //f[]存储残留网络容量
int q[N],d[N],pre[N],incf[N]; //q[]为bfs用队列,d[]为增广路径的残存容量,pre[N]为记录的反向边
bool st[N];         //incf[u]表示到u点的最大流量

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

bool spfa(){
    int hh=0,tt = 1;
    memset(d, 0x3f, sizeof d);
    memset(incf, 0, sizeof incf);
    q[0] = S, d[S] = 0, incf[S] = INF;
    while(hh != tt ){  //由于spfa会使得每个点入队多次，因此需要使用循环队列
        int t = q[hh ++];
        if(hh == N) hh = 0;
        st[t] = false;

        for(int i=h[t];~i;i=ne[i]){
            int ver = e[i];
            if(f[i] && d[ver] > d[t] + w[i]){
                d[ver] = d[t] + w[i];
                pre[ver] = i;
                incf[ver] = min(f[i],incf[t]);
                if(!st[ver]){
                    q[tt ++ ] = ver;
                    if(tt == N) tt = 0;
                    st[ver] = true;
                }
            }
        }
    }
    return incf[T] > 0;
}

void EK(int &flow,int &cost){
    flow = cost  =0;
    while(spfa()){
        int t = incf[T];  //走到终点时的最大流量
        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;
        }
    }
}
```

最大流

1飞行员配对方案问题

> m个外籍和n-m个英国飞行员两两配对，问最大匹配数和匹配方案

- 建图方式：

根据流网络的定义，先建立源点S和汇点T，从源点S向所有外籍飞行员建立一条容量为1的边，从所有外籍飞行员向其可搭档的英国飞行员建立一条容量为1的边，从所有英国飞行员向汇点T建立一条容量为1的边。整个流网络的可行流中的最大流便是最大匹配方案

2.圆桌问题

> 对于m个单位,第i个单位r[i]个人，n张餐桌，第i个餐桌容纳c[i]个人，要求一张餐桌上不能出现二个人来自同一家公司，问是否有方案能满足所有人都能有座位

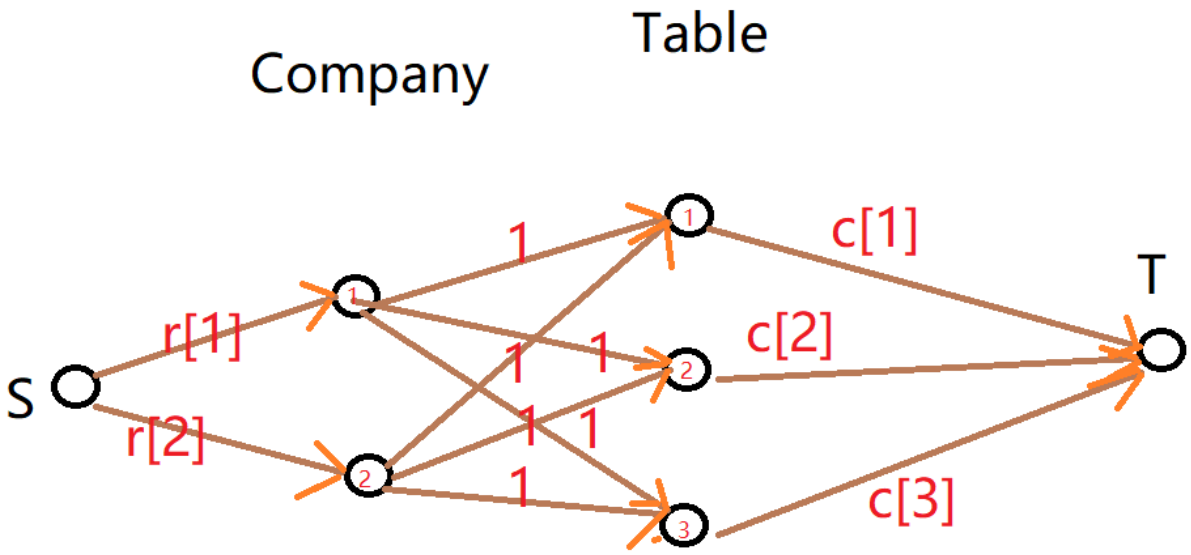
- 建图：

左集合S为每一家公司，右集合D为每一张桌子。建立源点S和汇点T，从源点向S集合的每一家公司连一条容量为r[i]的边，从S集合每一家公司向D集合每一张餐桌连一条容量为1的边，从D集合每一张餐桌向汇点T连一条容量为c[i]的边。

- 证明：

$$s \to f\text{可行解对应可行流}$$

左边：每个公司派到对应桌子的人；右边：每张桌子坐的人；



无源汇上下界可行流

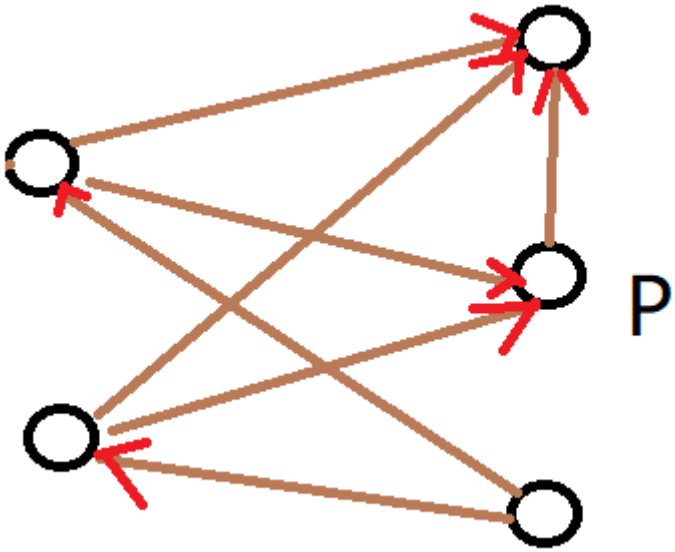
> n个点m条边的有向图，每条边有一个流量下界和流量上界规范，求是否存在一个可行流

设原网络为(G,F),变换后网络为(G',S'),每条边上界Cl(u,v)，下界Cu(u,v),流量f(u,v).对于上下界，我们很容易将等式变形：

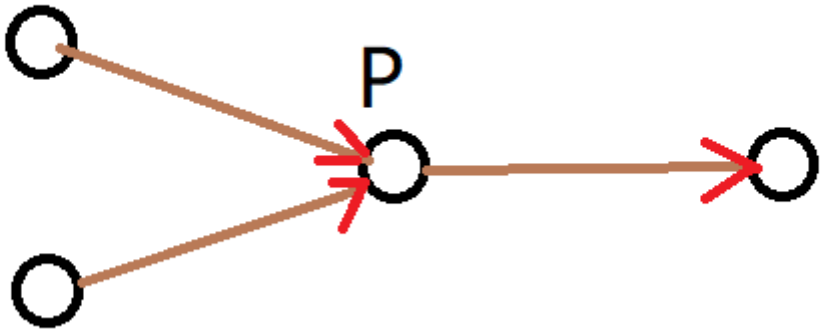
$$c_l(u,v) \le f(u,v) \le c_u(u,v) \to 0 \le f(u,v) - c_l(u,v) \le c_u(u,v) - c_l(u,v)$$

考察变形之后的网络，满足容量限制，但是并不满足流量守恒，原因：

对于一个有向图G



考察图上P点：



$$变换前：f(v1,p)+f(v2,p)=f(p,v3)$$

$$变换后：(f(v1,p)-c)+(f(v2,p)-c)\neq f(p,v3)-c$$

因此我们需要进行构造来使得变换后的仍为一个流网络

- 构造方法：（此处略与y总不同，但是更好理解）

> $C_{入}$:进入P点，因变换而减少的流量（相当于进来的补给变少了）；$C_{出}$:从P点出，因变换而增加的流量（相当于出去的消耗变少了）；C：$C = C_{出} - C_{入}$

对于每一个点，若其$C > 0$则说明补给减少的量小于消耗 减少的量（虽然给我少了，但是我用的更少），说明该点有余量，因此从此点向汇点T连接一条容量为c的边来释放多余的量

若$C < 0$则说明消耗减少的量小于补给减少的量（流出了很多但是流进来很少），说明该点缺少流量，因此需要从源点S流入c的流量才能满足流出的流量。因此从源点S向此点连接一条容量为c的边来补充缺少的量

> 注意：此时可以发现对于任意P点,若S与之相连，那条边是满流的，之若与T相连，那条边也是满流（因为构造就是这么构造的），因此**形成了一个从源点到汇点的最大流**

代码

难在建图，建图建好了直接上板子就行

```
const int N = 510 , M = 2*(11000+N) , INF = 1e8;
int n,m,S,T;
int e[M],ne[M],h[N],f[M],l[M],idx=0;
int cur[M],q[M],d[M],A[M];

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

bool bfs(){
    memset(d,-1,sizeof d);
    int hh = 0 , tt = 0;
    q[0] = S , cur[S] = h[S] , d[S] = 0;

    while(hh <= tt){
        int u = q[hh ++];
        for(int i = h[u]; ~i ; i = ne[i]){
            int ver = e[i];
            if(d[ver] == -1 && f[i]){
                d[ver] = d[u] + 1;
                cur[ver] = h[ver];
                if(ver == T) return true;
                q[++ tt] = ver;
            }
        }
    }
    return false;
}

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

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

int dinic(){
    int ans = 0 , flow = 0;
    while(bfs()) while(flow = find(S,INF)) ans += flow;
    return ans;
}
//================================
int main(){
    memset(h,-1,sizeof h);
```

```
    n = read() , m = read();
    S = 0 , T = n + 1;

    int tot = 0;
    rep(i,1,m){
        int a = read() , b = read() , c = read() , d = read();
        add(a,b,c,d);
        // tot += d - c; 最大流流量应该与中间怎么流的没有关系
        A[a] += c , A[b] -= c;  //每有一条出边变换之后会少消耗c（相当于+c），每有一条入边会少补给c（相当于-c）
    }
    rep(i,1,n)
        if(A[i] > 0) add(i,T,0,A[i]) ;//还有余量，因此流向汇点
        else if(A[i] < 0 ) add(S,i,0,-A[i]),tot-=A[i]; //缺少补给，因此从源点流入
    /*y总写法，一样的道理，反过来考虑
    rep(i,1,m){
        int a = read() , b = read() , c = read() , d = read();
        add(a,b,c,d);
        A[a] -= c , A[b] += c;   //每有一条入边变换之后就能多消耗c,每有一条出边变换之后就能少消耗c
    }
    rep(i,1,n)
        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<2*m;i+=2)
            print(f[i^1] + l[i]);
    }
    return 0;
}
```

有源汇上下界最大流

> n个点m条边的有向图，每一条边都有一个流量下界和流量上界，求从源点S到汇点T的最大流

- 算法：

> 1.记原来的源点为s，汇点为t,同无源汇的方式，建立虚拟源点S和汇点T，通过式子变形将其转化为一个新图G'
>
> $$变换前：f(v1,p)+f(v2,p)=f(p,v3)$$
>
> $$变换后：(f(v1,p)-c)+(f(v2,p)-c)\neq f(p,v3)-c$$
>
> 由于建立新图之后，此时原来的源点s和汇点t会变成中间结点，因此为了使之流量守恒，我们需要从t向s连一条容量为正无穷的边。
>
> 2.从S向T跑一遍Dinic算法，如果从S出的流量是满流，说明新图存在可行流，即此上下行约束下存在可行流
>
> 3.若存在可行流，再原来的源点s到原来的汇点t的跑一遍dinic算法（在残留网络上不断的搞掉增广路径），此时得到的便是s到t的最大流

- 证明：
  - 证明思路：证明原图中的上下界可行流于新图中的可行流存在一一映射
- 令

$$(G原图, f_0可行上下界可行流) \to (G^{'}新图中的可行流, f_0^{'}新图中的可行流(从S出发的满流))$$

$$G^{'}_{f_0^{'}}为f_0^{'}的残留网络, f^{'}_{0s\to t}为f_0^{'}中的s \to t的可行流$$

证：

$$|f^{'}_{0s\to t} + f^{'}_0|仍然是新图上的可行流$$

流量守恒：由S出去的都是满流，进入T的也都是满流，因此对于s -> t的任意可行流都不经过S和T

容量限制：由于t向s连接了一条容量为无穷的边，因此每一条边都满足容量限制

$$f^{'}对应一条s \to t的可行流$$

`f'-f0'`可以构造出所有流量都在中间，因为S、T对于`f'`和`f0'`而言都是满流 。

综上,原图中的上下界可行流于新图中的可行流存在一一映射,因此算法X具有正确性

```
const int INF = 1e8,N=510,M=2*(N+10010);
int n,m,S,T;
int e[M],ne[M],h[N],f[M],idx=0;
int q[M],cur[M],d[M],A[M];

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

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

bool bfs(){
    memset(d,-1,sizeof d);

    int hh = 0 , tt = 0;
    q[0] = S , cur[S] = h[S] , d[S] = 0;

    while(hh <= tt){
        int u = q[hh ++];
```

```
        for(int i=h[u];~i;i=ne[i]){
            int ver = e[i];
            if(d[ver] == -1 && f[i]){
                d[ver] = d[u] + 1;
                cur[ver] = h[ver];
                if(ver == T) return true;
                q[++ tt] = ver;
            }
        }
    }
    return false;
}

int dinic(){
    int ans = 0 , flow = 0;
    while(bfs()) while(flow = find(S,INF)) ans += flow;
    return ans;
}
//================================
int main(){
    memset(h,-1,sizeof h);

    int s,t,ans=0;
    n = read() , m = read() , s = read() , t = read();

    S = 0 , T = n + 1;

    rep(i,1,m){
        int a = read(), b = read() , c = read() , d = read();
        add(a,b,d-c);
        A[a] += c ,A[b] -= c;
    }
    int tot = 0;
    rep(i,1,n){
        if(A[i] > 0) add(i,T,A[i]);
        else if(A[i] < 0 ) add(S,i,-A[i]),tot-=A[i];
    }
    add(t,s,INF);
    if(dinic() < tot) puts("No Solution");
    else{
        ans = f[idx - 1];  //f[idx-2]为容量，因此其反向边f[(idx-2)^1] = f[idx-1]为流量
        f[idx-1] = f[idx-2] = 0; //删除这两条边
        S = s , T = t;
        print(ans+dinic());
    }
    return 0;
}
```

有源汇上下界最小流

有源汇上下界最小流只需要对有源汇上下界最大流进行变形即可，本质相同。

- 思路：

> 由有源汇最大流可知，原网络G上任意一可行流f经过变换得到的新网络上可行流f'都可以通过新网络上一条可行流（S出发的满流）f0'与f s->t之和得到，即
>
> $$|f^{'}| = |f_0^{'}| + |f_{s \to t}|$$
>
> 又有
>
> $$|f_{s \to t}| = -|f_{t \to s}|$$
>
> 因此，我们可以通过按照无源汇的方式建立好虚拟源点S流入虚拟汇点T流出的流网络。通过dinic算法验证此流网络是否满足S所有出边都是满流来考察新网络是否存在可行流。若存在，从原来的汇点t向原来的源点s做一遍dinic算法，从而消除t -> s路径上的所有增广路径以获得t -> s的最大流，由上述公式，t -> s为最大流，则s -> t为最小流，因此最终的答案即为：
>
> $$ans = |f_0^{'}| - |f_{max(t \to s)}|$$

代码：

```
int n,m,S,T;
int e[M],ne[M],f[M],h[N],idx=0;
int q[M],d[N],cur[M],A[N];

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

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

bool bfs(){
    memset(d,-1,sizeof d);
    int hh = 0 , tt = 0;
    q[0] = S , cur[S] = h[S] , d[S] = 0;

    while(hh <= tt){
        int u = q[hh ++];
        for(int i=h[u];~i;i=ne[i]){
```

```
            int ver = e[i];
            if(d[ver] == -1 && f[i]){
                d[ver] = d[u] + 1;
                cur[ver] = h[ver];
                if(ver == T) return true;
                q[++ tt] = ver;
            }
        }
    }
    return false;
}

int dinic(){
    int ans = 0 , flow = 0;
    while(bfs()) while(flow = find(S,INF)) ans += flow;
    return ans;
}
//================================
int main(){
    memset(h,-1,sizeof h);

    int s , t , ans = 0;

    n = read() , m = read() , s = read() , t = read();
    S = 0 , T = n + 1;

    rep(i,1,m){
        int a = read() , b = read() , c = read() , d = read();
        add(a,b,d-c);
        A[a] += c , A[b] -= c;
    }
    int tot = 0;
    rep(i,1,n){
        if(A[i] > 0 ) add(i,T,A[i]);
        else if(A[i] < 0) add(S,i,-A[i]) , tot -= A[i];
    }
    add(t,s,INF);
    if(dinic() < tot) puts("No Solution");
    else{
        ans = f[idx - 1]; //s向t的流量
        f[idx - 1] = f[idx - 2] = 0;
        S = t, T = s;
        print(ans - dinic());
    }
    return 0;
}
```

最大流的判定

> 对于n个点m条边的无向图，求最小的最大长度L，使得图中可以找出k条从S到T的没有重复路径，其中L为所有路径中的最大值，请你最小化这个最大值

- 思路

本题考虑二分出最大边数限制，然后使用最大流。一般来说，最大流的题目都会有一定的限制条件，比如次数限制，数值限制等。还可能会出现改变边权，或者从起点到终点多次的要求。

- 做法

根据源点和汇点建立流网络，对无向图的每一条边建立正向和反向两种边，容量都为１，其中由于一开始每条边的流量不确定（待二分），因此只记录当前边的边长；

二分出最长边，根据将大于最长边的所有边删除后求S到T的最大流是否大于 k 来判断此时的二分答案是否合法。（S到T的最大流即是S到T无重边的路经数）

- 疑惑点

对于无向图在建边的时候，可以正向反向都建，而且分别对于其残留网络可以不用再多建了。就相当于，假如两条边都有流量的时候，即正向边与反向边都不经过即可。

```
const int N = 210 , M = (N + 40010) * 2;
const int INF = 1e8;
int n,m,K,S,T;
int e[M],ne[M],h[N],f[M],idx=0;
int cur[M],q[M],d[N],w[M];

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] = c , ne[idx] = h[b] , h[b] = idx ++;
}

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 ver = e[i];

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

bool bfs(){
    memset(d,-1,sizeof d);
    int hh = 0 , tt = 0;
    q[0] = S , cur[S] = h[S] , d[S] = 0;

    while(hh <= tt){
        int u = q[hh ++];
        for(int i=h[u];~i;i=ne[i]){
            int ver = e[i];
            if(d[ver] == -1 && f[i]){
                d[ver] = d[u] + 1;
                cur[ver] = h[ver];
                if(ver == T) return true;
                q[++ tt] = ver;
```

```
            }
        }
    }
    return false;
}

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

bool check(int mid){
    for(int i=0;i<idx;i++){
        if(w[i] > mid) f[i] = 0;
        else f[i] = 1;
    }
    return dinic() >= K;
}
//================================
int main(){
    memset(h,-1,sizeof h);

    n = read() , m = read() , K = read();
    S = 1 , T = n;
    rep(i,1,m){
        int a = read() , b = read() , c = read();
        add(a,b,c);
    }
    int l = 1 , r = 1e6;
    while(l < r){
        int mid = l + r >> 1;
        if(check(mid)) r = mid;
        else l = mid + 1;
    }
    print(l);
    return 0;
}
```

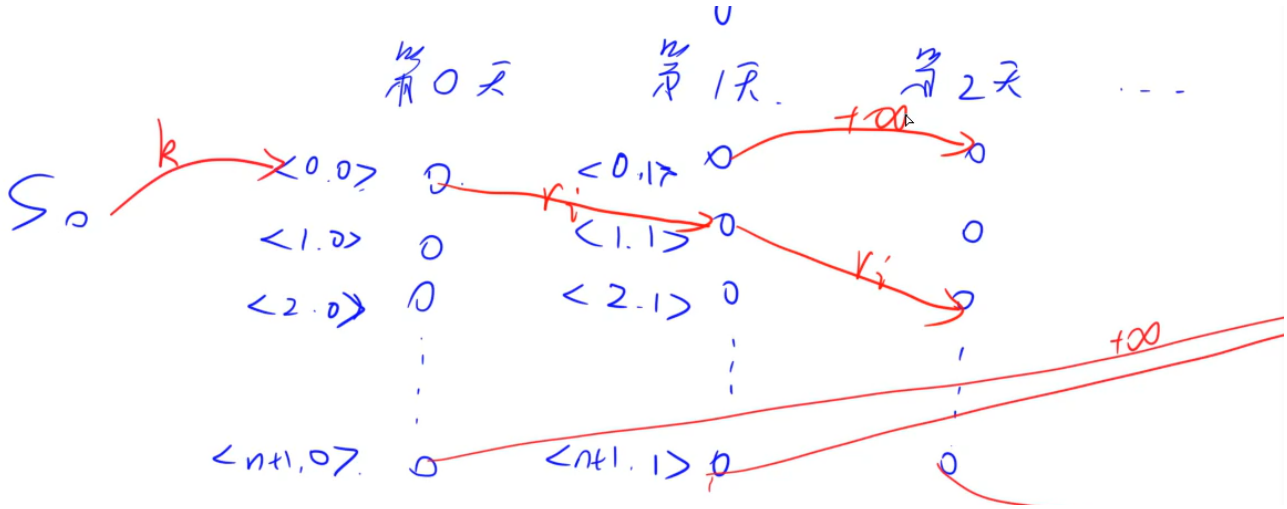
星际转移问题

> 从地球到月球由n个太空站，有m个飞船在做某些太空站中的循环飞行，每艘飞船都有其容量。问地球上有k个人时最短需要多长时间能把人全部从地球运往月球

- 思路

由于要求天数的最小值，此外本题的属性还包括了每一座空间站，因此我们借助分层图这一技巧来进行建图，分层图，即将流量与距离产生联系。建图如下：

流网络：<i，j>表示第i个工作站第j天所代表的点



图上有四种边：

[1]从源点S向<0,0>连一条容量为k的边，表示有k个人

[2]对于每一个点<i,j>向<i,j+1>连一条容量为正无穷的边表示每个人都可以在当前工作站停留一天

[3]对于第j列到第j+1列中，根据所有公交车对用站中的转换连接一条容量为公交车容量的边，表示可以通过公交车转换到下一站

[4]从所有天的第n+1个点向汇点T连一条容量为正无穷的边，表示已经到达了月球

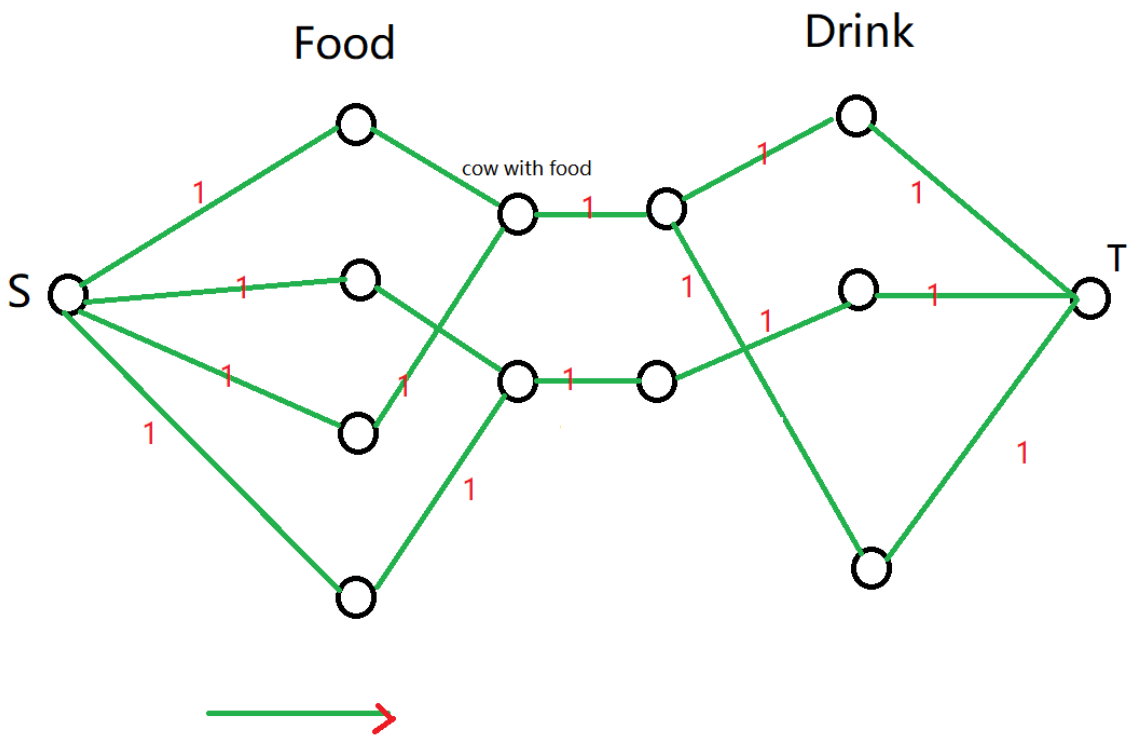
day可以用二分来枚举，但是由于day随着人数增多而增多而且删除比增加麻烦，因此直接枚举即可。

拆点

餐饮奶牛

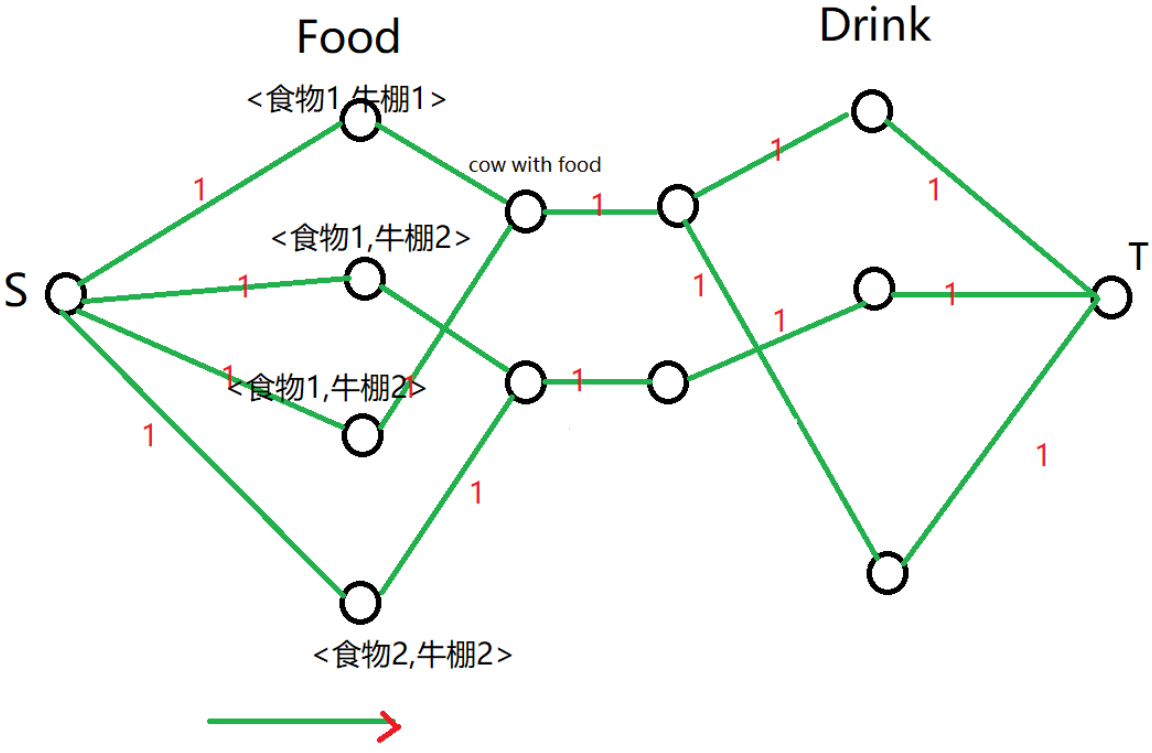
> n头奶牛，第i头有F[i]种喜欢的食物和D[i]种喜欢的饮料。现在有提供了F种食物和D种饮料，求能够分配的有吃有喝的最大奶牛数量

本题的拆点方式如下：



将cow拆成两个点，两点以一条容量为1的边相连，能够限制从而不出现上述情况，这样保证了一头牛只会匹配一个食物和饮料。

在y总下面的视频我曾经评论：假如这头牛有三个或者更多的需求的话该怎么做呢？显然，当时我的想法确实是一种方法，那就是将其中两个需求通过组合变成一个新点。例如本题中假如有两种食物，三种饮料，还有两种牛的饮食牛棚的话，我们可以将牛棚和食物通过组合的方式构造出新的四个点，然后再将对应的点进行连接，建图就完成了~



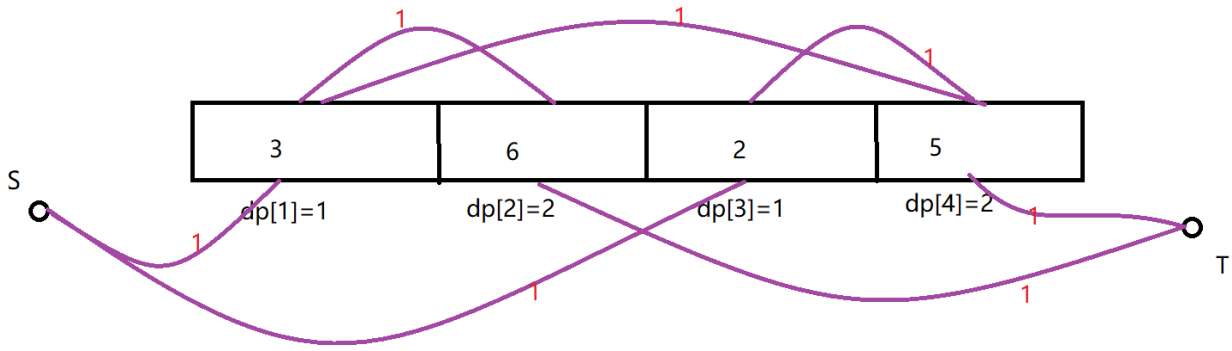
最长递增子序列问题

> 给定一个正整数序列x1,x2,x3...，求
>
> 1.其最长递增(非严格)子序列的长度s。
>
> 2.计算从给定的序列种可取出多少个长度为s的递增子序列。
>
> 3.如果允许多次使用x1和xn，求最多可取出多少个长度为s的递增子序列

第一问的话很好求解，直接DP即可，状态转移方程为：

$$dp[i] = max(dp[i], a[i] >= a[j]?dp[j] + 1 : 1)$$

巧妙的是建图：



[1]从源点 S 向所有dp[i]=1连一条容量为1的边

[2]从所有dp[i]=s 向汇点 T 连一条容量为1的边

[3]由于一个点只能被使用一次，因此顺理成章我们需要拆点，即对于每个点向拆出来的点连一条边

[4]对于j<i若有dp[j]==dp[i]-1则从j向i连一条边，表示可以组成一个单调不减子序列

这样，从S->T的最大流便是可拆个数

对于第三小问只需要：

- 将源点流出和流入汇点的边全改为正无穷即可
- 将所有dp[i]=1和dp[i]=s的点与拆出来的点之间的边改为正无穷即可

巧妙建图

> 猪：题目可抽象为m个点有对应的数值mi，有n个人取数，每次在能取的点中共取值最多ki的数，取完之后可以在能取到的点中重新分配每个点的权值。问最多能共取得多少值的数

由于存在时序性，因此需要考虑特殊的建图方式：

- 思路：建立源点S和汇点T，中间集合为来取数的人。对于每个点，从S向所有人中第一次来取这个点的人连一条容量为对应权值的边。如果当前人不是第一个来取这个点的值的人，为了体现时序性，记录每一个点上一次取数的人是谁，然后从上一次取此点的人向当前人连一条容量为正无穷的边。最后从所有人向汇点连一条容量为此人取数值的上限的边

最小割

网络战争

> 一个带权无向图G=(V,E),每条边e有一个权we求将点S和点T分开的一个边集C是的该割集的平均边权最小，即最小化$\frac{\sum_{e\in C} w_e}{\absC}$
>
> 注意，此处的边权指的是将某些边删去后S和T将不再连通

- 分析：

这是一道01分数规划的题目，老规矩设$\frac{\sum_{e\in C} w_e}{\absC} = \lambda$然后通过不断二分λ来求得最小值

既然删去某些边之后将不会再连通，因此对于两集合之间的割是必选的。其次选上这些割之后，我们仍然可以在两集合中的边进行选择。化简式子$\frac{\sum_{e\in C} w_e}{\absC} < \lambda$ then $\sum_{e\in C} w_e < \lambda \cdot \absC$ then $\sum_{e\in C} w_e - \lambda \cdot \absC < 0$ then |C|表示边数，将其带入得到最终式子：$\sum_{e\in C}(w_e - \lambda) < 0$

因此对于不是割边的边，如果小于λ，我们选上能够使得答案最小化。

- 有技巧

对于无向图，统一的建边方式：

```
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] = c, ne[idx] = h[b], h[b] = idx ++;
}
```

然后对于每次二分通过对w[i]进行转化即可

Code：

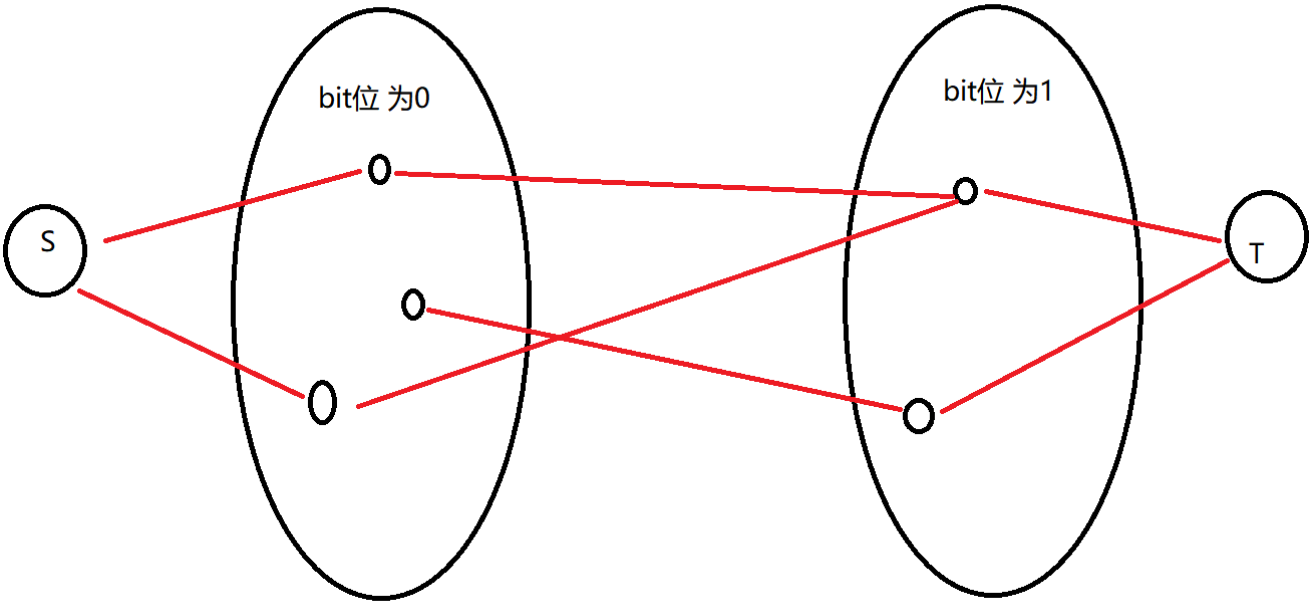
```
bool check(double x){
    double ans = 0;
    for(int i=0;i<idx;i+=2)
        if(w[i]<=x) {
            ans += w[i] - x;
            f[i]=f[i^1] = 0;
        }
        else f[i] = f[i^1] = w[i] -  x;
    return ans + dinic() < 0;
}
```

最优标号

> 给定一个无向图 G=(V,E)G=(V,E)，每个顶点都有一个标号，它是一个$[0, 2^{31} - 1]$内的整数。
>
> 不同的顶点可能会有相同的标号。对每条边 (u,v)，我们定义其费用 cost(u,v) 为 u 的标号与 v的标号的异或值。现在我们知道一些顶点的标号。你需要确定余下顶点的标号使得所有边的费用和尽可能小。

思路：位运算经典思考方式，按位考虑。下面对于每一位：

我们可以抽象图为：



对于有边的两点我们连双向边，对于已经确定的且第bit位的点如果是0则从源点连单向，如果为1则向汇点连单向即可。最终求最小割就是当前位对应的最小值。

```
void build(int bit){
    memset(h,-1,sizeof h); idx =0 ;
    rep(i,1,m) {
        int u=edge[i].x,v=edge[i].y;
        add(u,v,1,1);
    }
    rep(i,1,n)
        if(p[i]>=0){
            if(p[i]>>bit&1) add(i,T,INF,0);
            else add(S,i,INF,0);
        }
}

LL solve(int bit){
    build(bit);
    return dinic();
}
```

格取数问题

> 给定一些方格，请设计一个算法取数要求取出的所有数中没有公共边

- 思路

对于方格问题，首先因该想到的是进行二染色，然后将`i+j&1==1`分成左集合，另外为右集合。这样左右集合对应的数都不能同时取到。从源点向左集合连容量为取值的边，从右集合向汇点连容量为取值的边。用总点数减去最小割就能保证割去的边使得左右集合不连通的同时，割去的边权值最小。

技巧一：

如果有些边不想让其成为割边的话，则人为得将这些边设置为容量为正无穷的边。

网络流__11 最小割之最大权闭合图、最大密度子图

最小割之最大权闭合图

简单割定义：

所有割边都与源点或者汇点相连,由

$|f|_{最大流} = max\{\sum_{u \in V} f(s,u)\}$=最小化割 <= 所有割 ， 因此割边容量一定有限，因此此情况下最小割一定是简单割。

闭合图定义：

定义有向图$G=(V,E)$为一闭合子图当： $\forall v \in V$ 对应出边的端点v'都有$v' \in V$。简而言之，所有选的点集和边集中所有点集内点的出边都在边集中即可。

最大权闭合图：

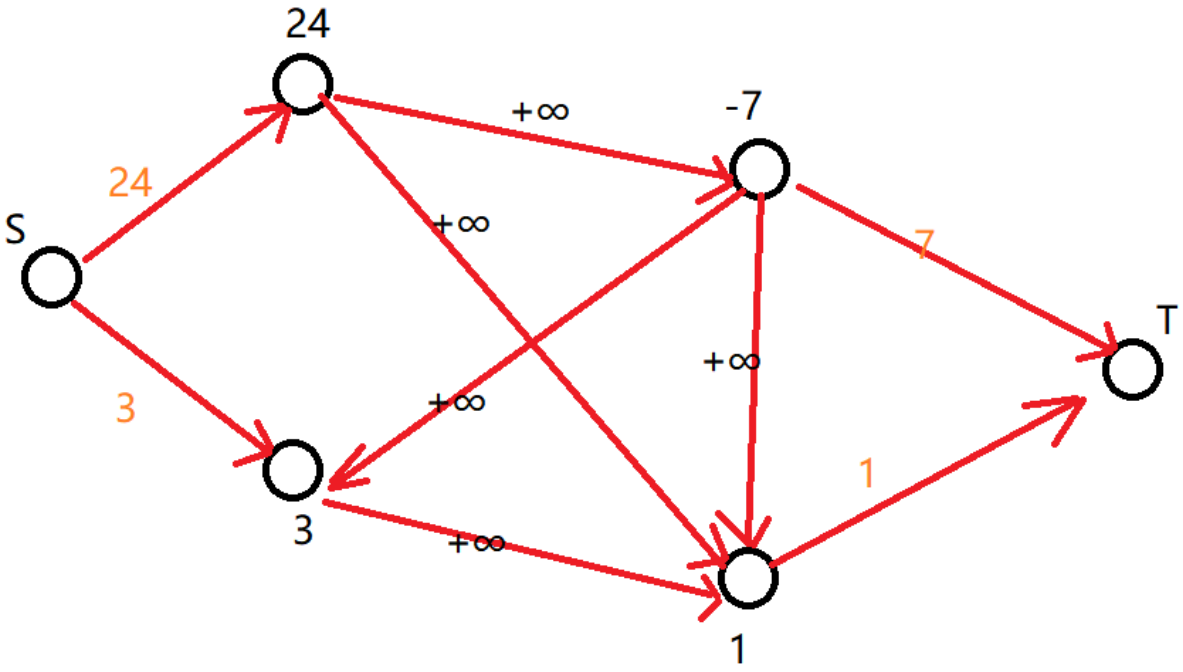
即再给出的边集以及边的关系中求得点权之和最大的闭合图集合

条件：

G为有向图。

一般性建图：

> 一堆点分别由正权点和负权点构成，每个点都至少有一种指向关系。建立源点和汇点，源点向所有正权点连对应权值的容量，所有负权点向汇点连对应绝对值权值的容量，每个点之间的转移容量为正无穷。此图中的最小割=最大流即为**最大收益**(描述的是一种正权减负权的关系)



证明 最小割情形下的闭合子图为最大权闭合子图：

由于不存在S->T的边，又因为是简单割，因此不存在内部点跨越两个割V1->v2,因此只存在S->v2和v1->T的两种边。其中V_1指的是选出的闭合图，V_2指的是选出的闭合图的补集

$C[S,T]=C[V_1,\{t\}]+C[\{t\},V_2]=\sum_{v\in V_2^+}W_v+\sum_{v\in V_1^-}-W_v$ (根据是简单边而得来)

（最优性）又有$W(V_1)=\sum_{v\in V_1^+}W_v-\sum_{v\in V_1^-}(-W_v)$

$C[S,T]+W(V_1)=\sum_{v\in v_2}W_v$,其中$\sum_{v\in v_2}W_v$为正权点集和，为定值，因此为了最大化$W(V_1)$,因此当为最小割的时候成立

例题：最大获利

> n个设施，建设耗费P_i花费，m个用户，第i个用户需要依赖一些设施才能使得你盈利C_i。问你如何建设才能获得最大盈利，求盈利

分析：因为每个点具有点权P_i或者C_i，且部分点有依赖，我们将其抽象为一个闭合子图，仿照闭合子图的建图方法，$S->^{C_i}$用户, 设施$->^{P_i}T$，由最大闭合子图的公式$Sum_+-C[S,T]=f_{\text{最大权闭合图}}$求解

```
void solve(){
    n=read(),m=read();
    S=0,T=n+m+1;
    rep(i,1,n){
        int x=read();
        add(i,T,x);
    }
    rep(i,1,m){
        int a=read(),b=read(),c=read();
        add(S,i+n,c);
        add(i+n,a,INF), add(i+n,b,INF);
        tot += c;
    }
    print(tot-dinic());
}
```

例题 Magic Slab

> 一个方格图，数组a[i],b[j]分别表示选择行i和列j的代价。每个格子都有个权值，当且仅当行和列的代价同时选上时会获得当前各自内权值的收益。此外有附加提交，即同时选上指定的两个格子时能获得一个额外的e[i]的收益；

建图方式与最大权闭合子图完全相同，额外收益的转化方式为新增加一个收益点连向四个代价点即可。

最大权闭合子图的方案

从源点开始沿着大于0的正向边搜索，能够搜到的点都是左集合，即选择的方案中的点。右边所有被割掉的点都是选择对应的必选项。

最大密度子图

定义：

定义一个无向图$G=(V,E)$,其中所有边的两个端点均被选择。即可以选择所有点但是不选边，但是选了边对应的点也要选

定义无向图$G=(V,E)$的密度D为该图的边数$|E|$与该图的点数$|V|$的比值$D=\frac{\absE}{\absV}$。给出一个无向图$G=(V,E)$,其密度最大的子图称为最大密度子图，及要求最大化D

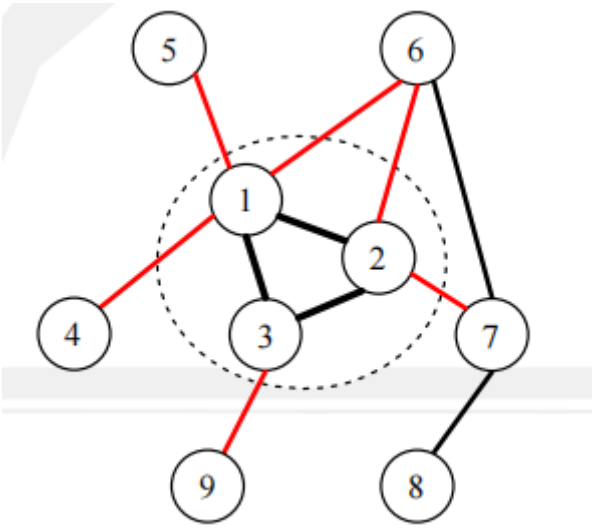
求解方法：01分数规划+最小割定理

$\frac{\absE}{\absV}=g$ 二分答案g，每次最大化 $\absE-g*\absV$，即最小化$g*\absV-\absE$(因为最小割定理解决的是最小化问题)，如果小于0，向左缩小区间，否则向右缩小区间

引理：

对于图G的子图$G'=(V',E')$在点集固定的情况下必然选的边越多越好，因此把选出点集所形成的所有边都选上就得到了一个G的导出子图，此方案下比其他方案更优

然后使用逆向思维：用所有边减去与V'关联的且不符合条件的边就可以得到E'，删去的边如红色所示：



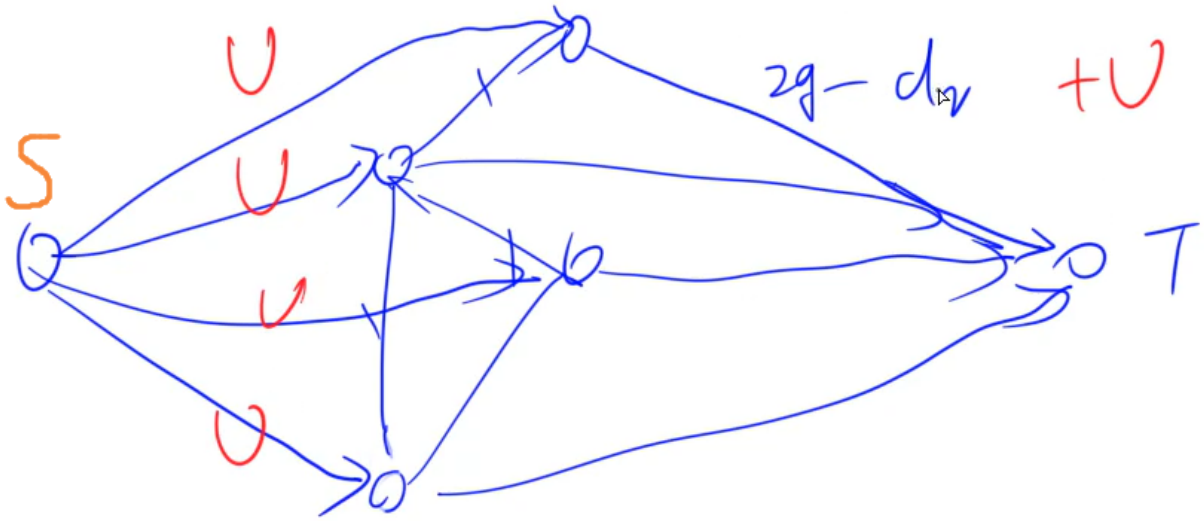
论文《最小割模型在信息学奥赛中的应用》提出红色的边实际上就是V'与其补集合$\overline{V'}$之间的边，因此尝试使用最小割定理。继续化简式子：

$g * \absV - \absE$

$= \sum_{v\in V'} g - (\frac{\sum_{v\in V'} deg_v}{2} - \frac{c[v',v'的补集]}{2}) = \sum_{v\in V'}(g - deg_v/2) + c[V', V'的补集]$

$= \frac{1}{2} \cdot (\sum_{v\in V'}(2g - d_v) + c[V', V'的补集])$

我们发现对于$\sum_{v\in V'}(2g - d_v)$处理方式是相当于选了这个点就要花费2*g-deg的代价，相当于负权边，因此建图方式为从此点向汇点连一条此绝对值容量的边，但是还有一个问题就是2*g-d有可能是是负数，因此处理方式就是加上一个偏移量U使得其为正数。因此最终的建图方式就是(中间的边权为1)：



此时$c[S,T] = \sum_{v\in V'} U + \sum_{v\in V'}(U + 2\cdot g - d) + \sum_{(u,v)\in E} 1 = U\cdot n - 2\cdot(\absE' - g\absV')$

生活的艰辛

> 在一个图G中找到一个最大密度子图

通过如上建图后跑最大流不断二分答案即可。

拓展1 有边权

如果有边权We的话，则对密度定义进行扩展为$D=\frac{\sum_{e\in E}W_e}{\absV}$

由于每条边赋上边权，则目标是最小化$g\cdot\absV'-\sum_{e\in E'}w_e$，定义$d_u=\sum$与$u$点相连的边

因此新的建图方式为：

$c(u,v)=c(v,u)=w_e$　　$c(S,v)=U$　　$c(v,T)=U+2g-d_v$

拓展2 同时有边权和点权

此时最大化的密度D定义为$D=\frac{\absW'_e+\absP'_v}{\absV'}$,根据二分答案，有：

最小化$\sum g-\sum\absP'_v-\sum\absW'_e=\sum(g-\absP'_v)-\sum\absW'_e$，同样定义$d_u=\sum$与$u$点相连的边

因此新的建图方式为：

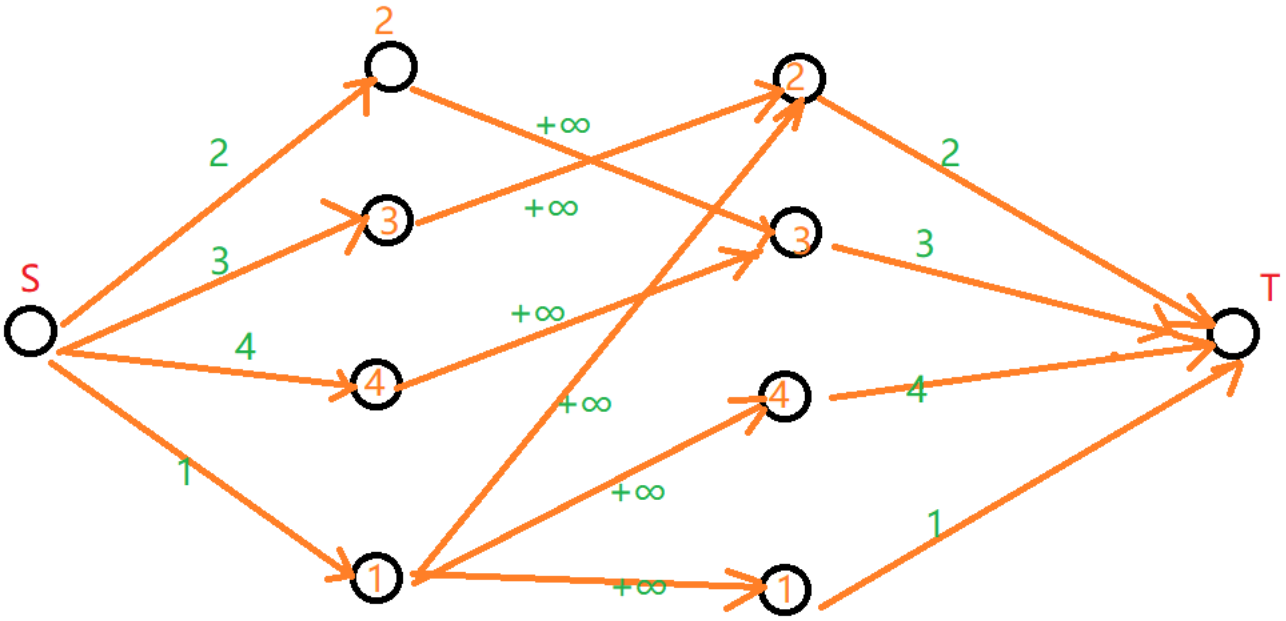
$c(u,v)=c(v,u)=w_e$　　$c(S,v)=U$　　$c(v,T)=U+2g-d_v-2p_v$

最小割之最小点权覆盖算法与最大独立集算法

由二分图我们知道最小覆盖集=最大二分图匹配=总点数-最大独立集，如果每个点都有权值，则最小点权覆盖应该对应KM二分图最优匹配算法，那么如果用网络流来求解，我们则需要借助最小割定理来辅助解决。

思路与建图方式：

在一个无向图中



从源点向左集合的所有点连容量为点权的边，从右集向汇点所有点连容量为点权的边，中间相关的点之间连接容量为+∞的边，这样可以使得原图的任意一个割为简单割，且由此，原图的最小割便是原图G的一个最小点全覆盖

例题一道：有向图破坏

> 给定一个有向图，，每次选取一个有权值的点，要么移除这个点的所有入边，要么移除这个点的所有出边，问最小花费

将点拆成移除入边和移除出边之后边能套上最小点权覆盖的板子。然后建好图跑完最大流就是关键的从最小割到最大流的转化：

> 割边的性质：不存在从源S到汇点T的一条路径，因此对应割在网络流上的一条路径：<S,u> <u,v> <v,T>，由于人为规定了<u,v>=INF，因此要么<S,u>=0,要么<u,T>=0;
> 如果<S,u>=0，对应左集合的一条割边
> 如果么<u,T>=0，对应右集合的一条割边

从源点出发，在残留网络中沿着剩余容量大于0的边走，所有遍历到的点构成集合S{}，剩余点构成集合T{}.注意，最小割中的割边都是正向边，因此i+=2进行遍历。

最大权独立集

点独立集定义：

$\forall e(u,v) \in E$满足$u \in V$和$v \in V$不同时成立

解决思路：

类比于二分图所有点权覆盖问题中的结论最大权独立集=所有点的总权值-最小权点覆盖

最大独立集=所有点的总权值-最小点权覆盖集

证明：反证法，任给一个覆盖集，其补集为独立集，其中假设有一边两点都没选上，与原集合为覆盖集矛盾。给定一个独立集，若其补集不是覆盖集，说明有一条边的两点都未被选择，说明独立集中有一边存在两个端点，矛盾。

例题

王者之剑 *姚金宇*

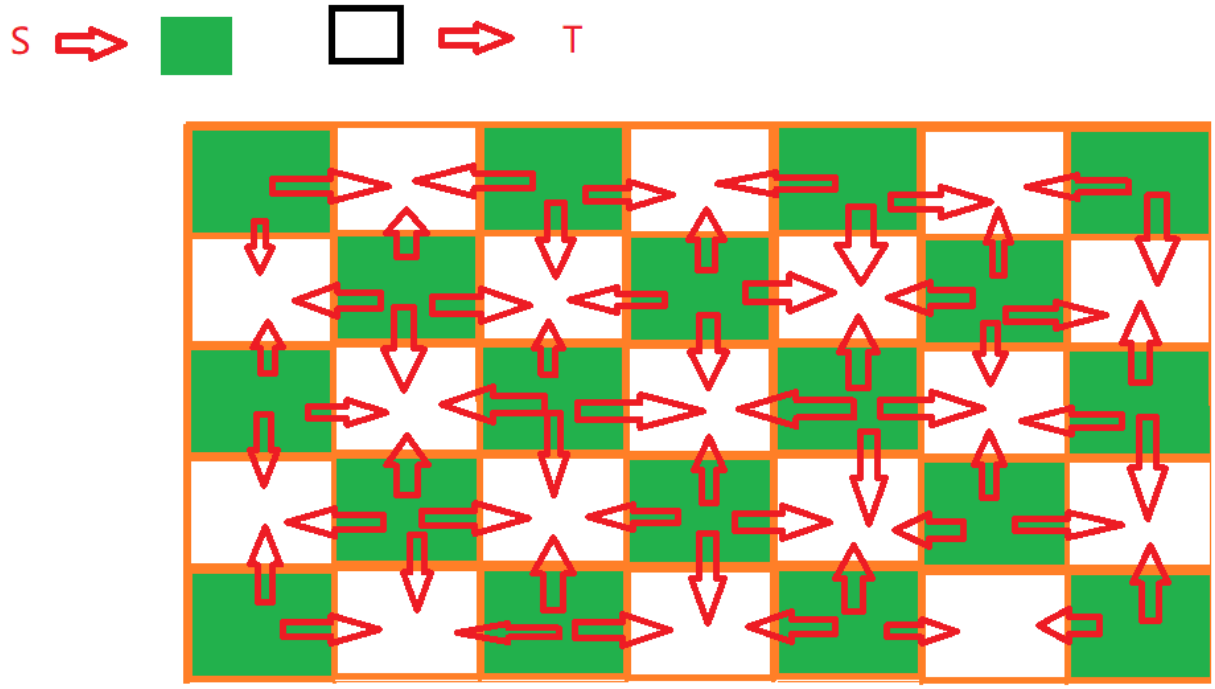
> wyh在一个n*m的网格，每个格子上有一个价值为$v_{i,j}$的宝石。wyh可以自己决定起点。开始时刻为第0秒，后面每秒按照顺序执行：
>
> 1、若第i秒开始时，wyh在(x,y)，则wyh可以拿走(x,y)上的宝石
>
> 2、在偶数秒时，wyh周围上下左右四格的宝石全部消失
>
> 3、若第i秒开始时，wyh在(x,y)，则在第(i+1)秒开始前，wyh可以移动到上下左右相邻的格子内（如果存在的话，移动速度为瞬间）

我们对奇偶秒不同这一条件进行挖掘可以得到1.取宝石必然在偶数秒（因为奇数秒进入的格子必为0），2.相邻两个格子的宝石必然不能同时拿

由2可以直接想到这和大多数二分图的问题的隐含条件类似，因此加入点权(宝石)这一条件后便是一个最大独立集问题。

建图思路：

每个格子进行黑白染色，相邻格子染上不同的颜色。不妨设左集合为坐标值和为偶数的点，右集合为坐标之和为奇数的点。因此从源点向左集合连对应点权的边，从左集合向右集合连容量为正无穷的边，从右集合向汇点连容量为点权的边。然后使用总点权-最小覆盖集=总点权-最小割=最大独立集进行求解



证明：

很容易证明每一个方案对应于二分图中的一种划分方式

下证明每一种划分方式对应一个可行方案：

> 每次考虑两行（第一行为主行，第二行只起辅助作用。因为第一行考虑结束后原来的第二行会变成考虑的主行）：
>
> 保证每次到达需要取得的格子时为偶数时间，如果是奇数时间，就在前两格的位置停顿一秒。

建图代码：

```
void solve(){
    LL ans = 0;
    n=read(),m=read(); S=0,T=n*m+1;
    rep(i,1,n) rep(j,1,m) {
        g[i][j] = read();
        ans+=g[i][j];
        if((i+j)%2==0) add(S,get(i,j),g[i][j]);
        else add(get(i,j),T,g[i][j]);
    }
    rep(i,1,n) rep(j,1,m) if((i+j)%2==0) rep(k,0,3) {
        int x = i + mov[k][0], y = j + mov[k][1];
        if(x<1||x>n||y<1||y>m) continue;
        add(get(i,j),get(x,y),INF);
    }
    print(ans-dinic());
}
```

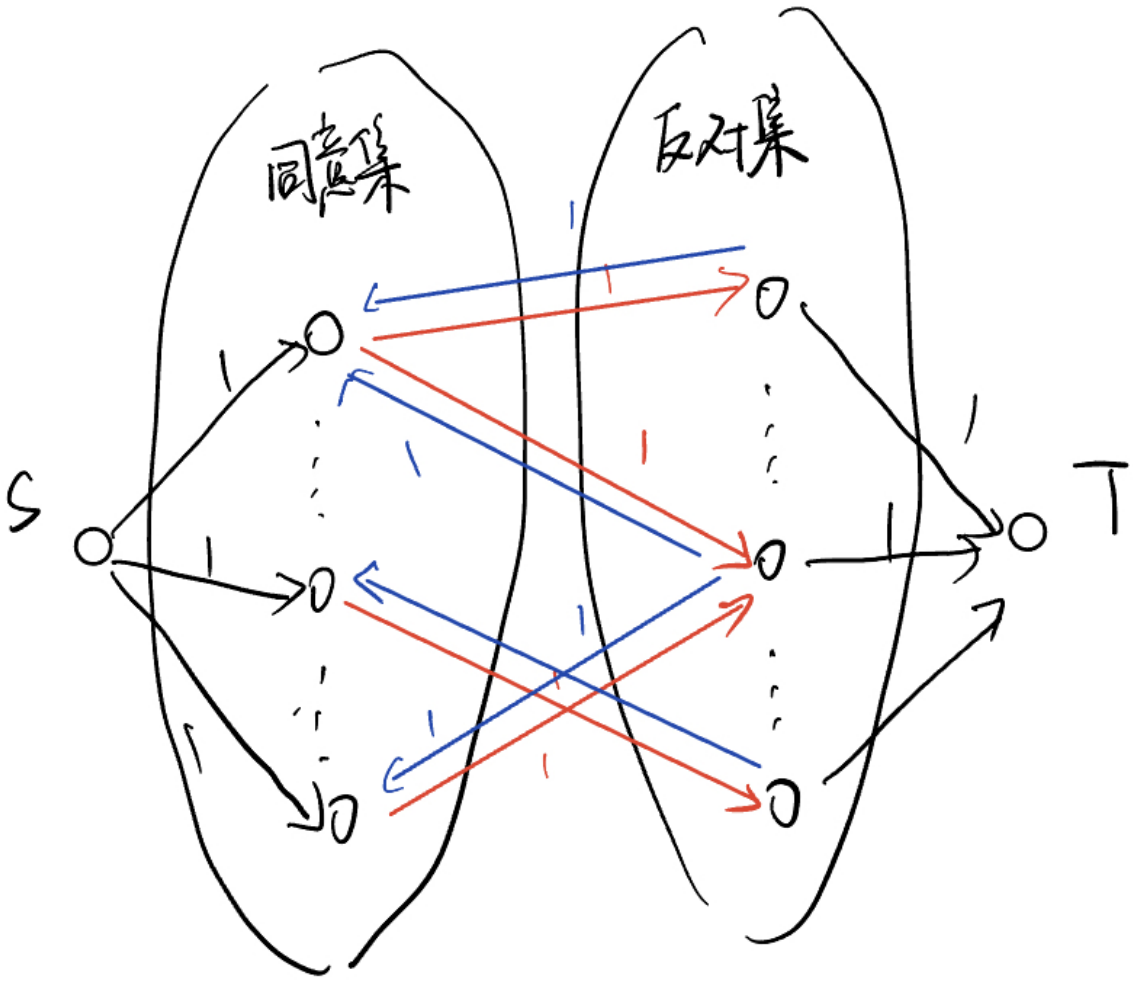
最小割中的二选一问题

对于处理网络流二选一问题，我们常常采用最小割算法进行求解。

P2057 [SHOI2007] 善意的投票 / [JLOI2010] 冠军调查

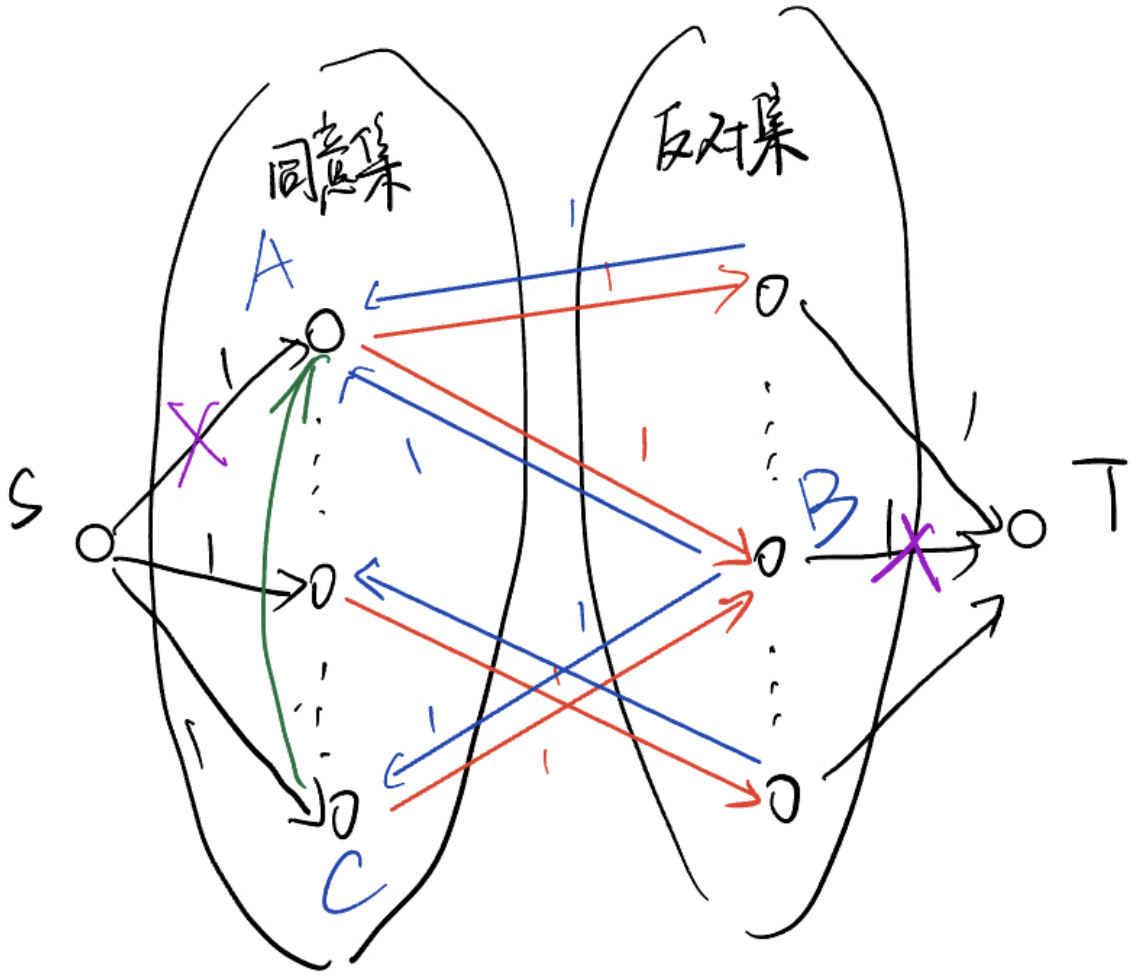
> 幼儿园里有 n*n* 个小朋友打算通过投票来 决定睡不睡午觉。
>
> 对他们来说，这个问题并不是很重要，于是他们决定发扬谦让精神。
>
> 虽然每个人都有自己的主见，但是为了照顾一下自己朋友的想法，他们也可以投和自己本来意愿相反的票。
>
> 我们定义一次投票的冲突数为好朋友之间发生冲突的总数加上和所有和自己本来意愿发生冲突的人数。
>
> 我们的问题就是，每位小朋友应该怎样投票，才能使冲突数最小?

对于这个问题，我们的建图方式是：建立源点S(表示同意睡午觉)和汇点T(表示不同意睡午觉)。然后对于每一对朋友i和j,i向j连一条容量为1的边，j向i连一条容量为1的边。



> 至于为什么朋友之间应该连双向边，我的解释：

如果只连接从左到右的单向边，会忽略如图的情况：尽管A的入边和A出边对应的出边都被割掉了，但是仍然有A->B->C->A的冲突没有解决，实际上说，这就是A选择了不同意，B选择了同意，C选择了同意，A和C的矛盾就靠双向边来计算。



P1361 小M的作物

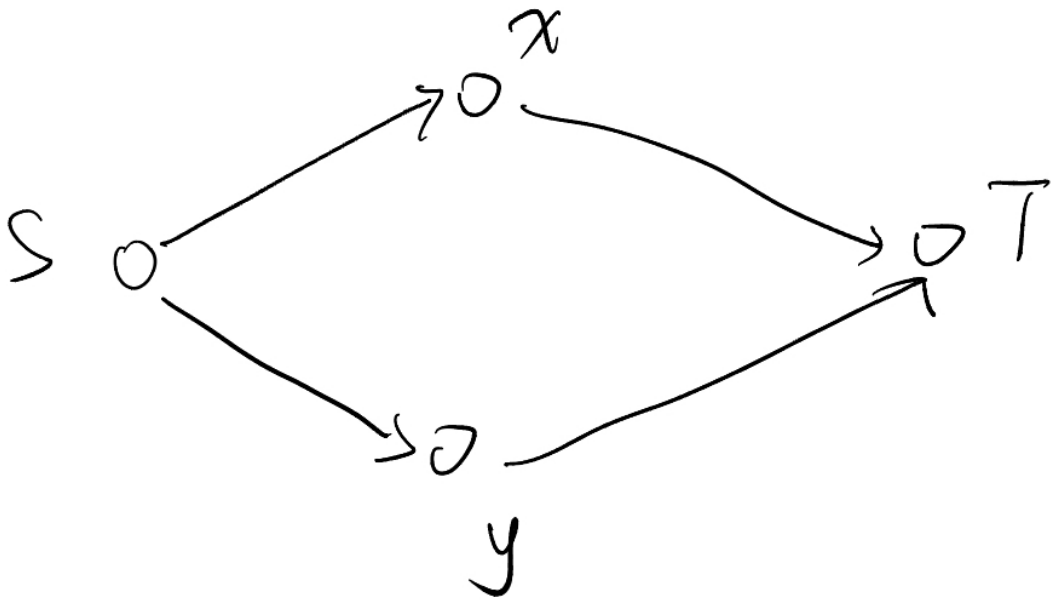
小 M 在 MC 里开辟了两块巨大的耕地 A*A* 和 B*B*（你可以认为容量是无穷），现在，小 P 有 n*n* 种作物的种子，每种作物的种子有 11 个（就是可以种一棵作物），编号为 11 到 n*n*。

现在，第 i*i* 种作物种植在A中种植可以获得 a_i*a***i*的收益，在 B*B* 中种植可以获得 b_i*b***i* 的收益，而且，现在还有这么一种神奇的现象，就是某些作物共同种在一块耕地中可以获得额外的收益，小 M 找到了规则中共有 m*m* 种作物组合，第 i*i* 个组合中的作物共同种在 A*A* 中可以获得 c*{1,i}c1,i 的额外收益，共同种在 BB 中可以获得 c*{2,i}*c*2,i*的额外收益。

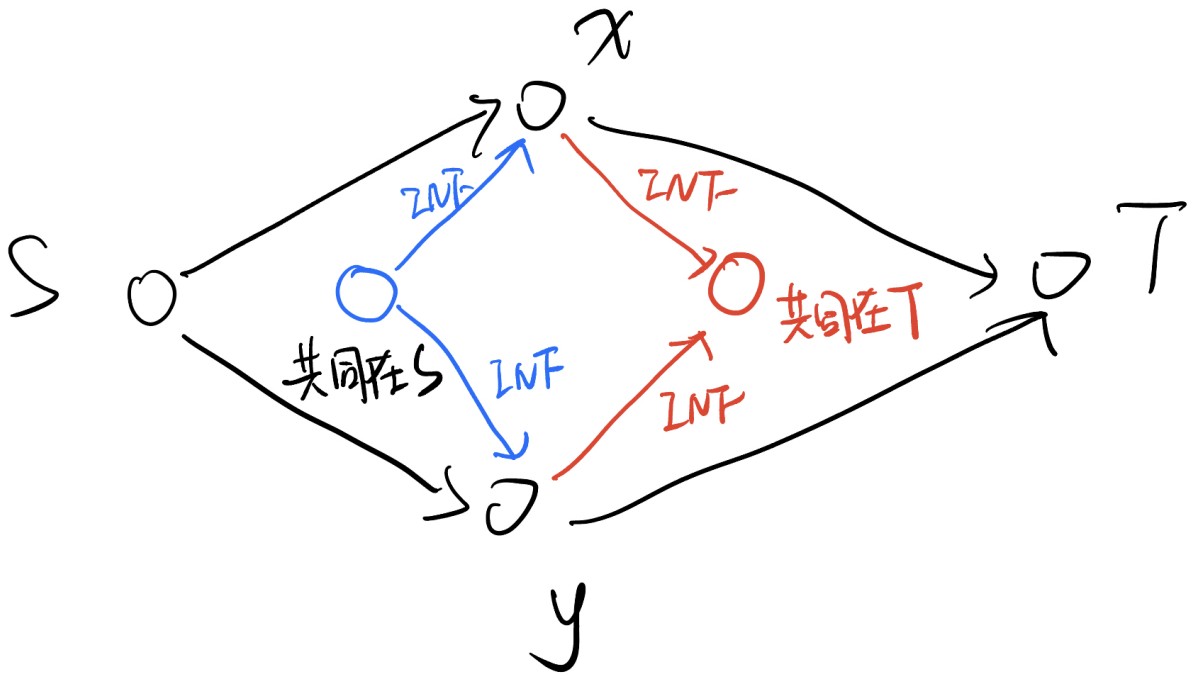
小 M 很快的算出了种植的最大收益，但是他想要考考你，你能回答他这个问题么?

建图方式：

对于二选一问题，我们常用的建图思路:



对于x，y之间若有依赖关系的话，我们可以建立新的虚电来描述额外的需要割掉的边：



然后我们求出的最小割对应的是非最优收益，因此我们用总收益减去非最优收益便可以得到最优收益了。

费用流

费用流

定义：

给定一个流网络，给每条边赋予一个花费，为w(u,v)费用/流.定义一个网络的可行流的费用为：$\sum f(u,v)\cdot w(u,v)$

> 注意，在增广路径中费用应该取反，这样方便退流

算法1 EK算法求最最小费用最大流

板子

> 此算法无法处理具有负权回路的图，否则需要使用消圈法

思路

将BFS找增广路改为SPFA找一条最短的增广路即可

Code

```
/*链式前向星存图的好处是可以快速找到反向边 edge^1*/
int n,m,S,T;
int h[N],e[M],w[M],f[M],ne[M],idx; //f[]存储残留网络容量
int q[N],d[N],pre[N],incf[N]; //q[]为bfs用队列,d[]为增广路径的残存容量,pre[N]为记录的反向边
bool st[N];         //incf[u]表示到u点的最大流量

void add(int a,int b,int c,int d){
    e[idx] = b, f[idx] = c, w[idx] = d, ne[idx] = h[a] , h[a] = idx++;
    e[idx] = a, f[idx] = 0, w[idx] = -d, ne[idx] = h[b] , h[b] = idx++;
}
/*把bfs换成spfa就是求最小费用最大流的方法*/
bool spfa(){
    int hh=0,tt = 1;
    memset(d, 0x3f, sizeof d);
    memset(incf, 0, sizeof incf);
    q[0] = S, d[S] = 0, incf[S] = INF;
    while(hh != tt ){  //由于spfa会使得每个点入队多次，因此需要使用循环队列
        int t = q[hh ++];
        if(hh == N) hh = 0;
        st[t] = false;

        for(int i=h[t];~i;i=ne[i]){
            int ver = e[i];
            if(f[i] && d[ver] > d[t] + w[i]){
                d[ver] = d[t] + w[i];
                pre[ver] = i;
                incf[ver] = min(f[i],incf[t]);
                if(!st[ver]){
                    q[tt ++ ] = ver;
                    if(tt == N) tt = 0;
                    st[ver] = true;
                }
            }
        }
    }
    return incf[T] > 0;
```

```
}

void EK(int &flow,int &cost){
    flow = cost  =0;
    while(spfa()){
        int t = incf[T];  //走到终点时的最大流量
        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;
        }
    }
}
```

运输问题

> W公司有m 个仓库和 n 个零售商店。
>
> 第 i 个仓库有 a_i 个单位的货物；第 j 个零售商店需要 b_j 个单位的货物。
>
> 货物供需平衡，即$\sum_{i=1}^{m} a_i = \sum_{j=1}^{n}(b_j)$。
>
> 从第 i 个仓库运送每单位货物到第 j 个零售商店的费用为 $c_{i,j}$。
>
> 对于给定的 m 个仓库和 n 个零售商店间运送货物的费用，计算最优运输方案和最差运输方案。

做法：建立超级源汇点然后（1）从源点向仓库连仓库容量花费0的边（2）从仓库向商店连容量INF花费$c[i][j]$的边（3）从商店向汇点连容量为商店需求，价格为0的边，跑最小费用最大流即可

求最大值：先将所有边还原`f[i] += f[i^1], f[i^1] = 0`,然后将所有边的花费取反。

注意事项，在双向边中不能进行合并，因为反向边为了退流导致费用为负

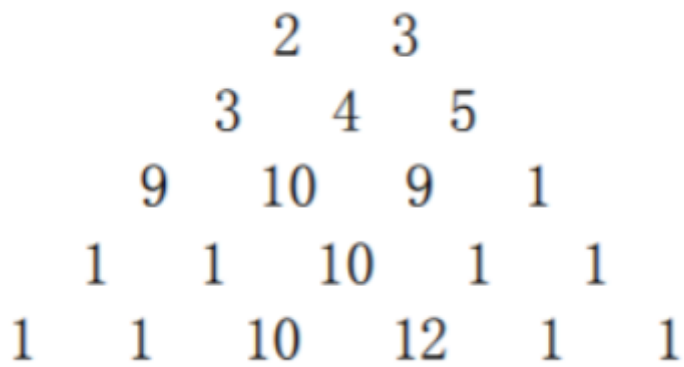
二分图最优匹配

> n个人n件物品，每个人对应很多件物品，对应每一件物品都有一个满意度。求最大满意度（保证是一个完美匹配）

- 思路

将所有花费去负后用二分图的建图方式跑最小费用最大流即可

数字梯形问题



> 给定一个由 n 行数字组成的数字梯形如下图所示。
>
> 梯形的第一行有 m 个数字。

> 从梯形的顶部的 m 个数字开始，在每个数字处可以沿左下或右下方向移动，形成一条从梯形的顶至底的路径。
>
> 规则 1：从梯形的顶至底的 m 条路径互不相交。
>
> 规则 2：从梯形的顶至底的 m 条路径仅在数字结点处相交。
>
> 规则 3：从梯形的顶至底的 m 条路径允许在数字结点相交或边相交。

建图方式：

`rule 1`

对每个点的限制 $\rightarrow$ 拆点，点之间容量限制为1

对每条边的限制 $\rightarrow$ 每条点与点之间的容量限制为1

`rule 2`

对每个点的限制 $\rightarrow$ 拆点，点之间容量限制为INF

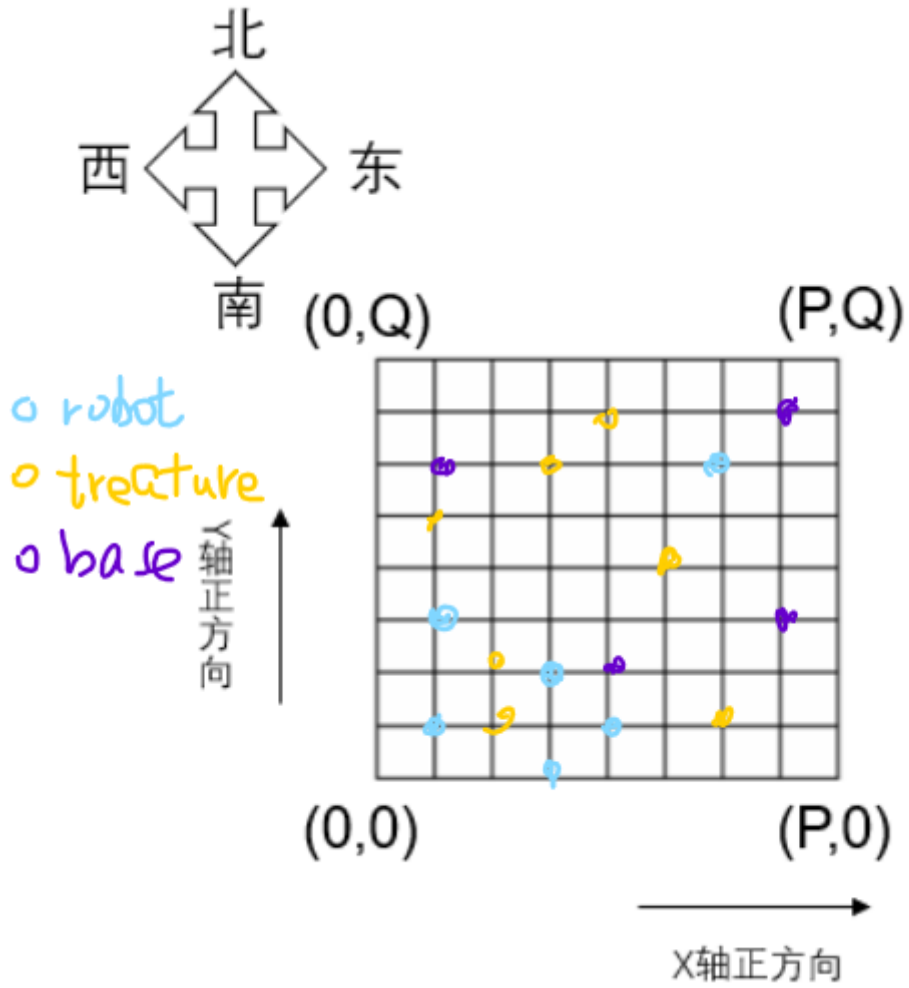
对每条边的限制 $\rightarrow$ 每条点与点之间的容量限制为1

`rule 3`

对每个点的限制 $\rightarrow$ 拆点，点之间容量限制为INF

对每条边的限制 $\rightarrow$ 每条点与点之间的容量限制为INF

深海机器人问题



> 地图上给出所有机器人的坐标和宝藏的坐标并给出机器人的目的地，限制机器人不能往回走(向下或者向左)，机器人必须到达基地，否则携带的的宝藏不能进行回收，没有价值。每个基地有一定的容量，即限制机器人到达此基地的个数。问机器人的最优移动方案是的尽可能多的机器人到达目的地。

- 建图思路

建立虚拟源点S和汇点T，从S向所有机器人连容量为机器人个数的边，从所有基地向汇点连容量为基地容量的边。然后根据网格图的建图方式，如果某个地方有宝藏，就建两种边`add(i,j,1,-val)`和`add(i,j,INF,0)`然后最大费用最大流即为所求。

餐巾计划问题

> 一个餐厅计划未来n天的餐巾安排。第i天需要r_i的餐巾。购买一块餐巾需要p元，送到快洗店需要fc元，需要等待fd天才能洗好。送到慢洗店需要sc元，需要等待sd天才能洗好。问合理安排n天的计划，最少花费是多少

- 建图思路

理一下关系：

第一类点：每一天需要使用r_i条毛巾，因此从第i天向汇点T连容量为r_i的边。

第二类点：每一天会产出r_i条毛巾，可以分别通过快洗店和慢洗店通过一定的花费转移到第i+sd和i+fd天，或者直接购买花费p元

因此我们可以将每一天拆出第一类点和第二类点，第一类点向汇点连边，源点向第一类点连边

NOI2008志愿者招募

> 经过估算，这个项目需要 NN天才能完成，其中第 i 天至少需要 Ai 个人。布布通过了解得知，一共有 M 类志愿者可以招募。其中第 ii 类可以从第 Si 天工作到第 Ti 天，招募费用是每人 Ci 元。
>
> 问最小花费

- 错误建图:

源点向每个人连边，每类人向能够服务的天连边，每天向汇点连边。

错误原因：每个人工作不连续而且不好分配费用。

- 正确建图

实际上是一个有下届的最小费用最大流。每一天都有一个下届，因此按照无源汇有下届的建图方式进行建图即可。