

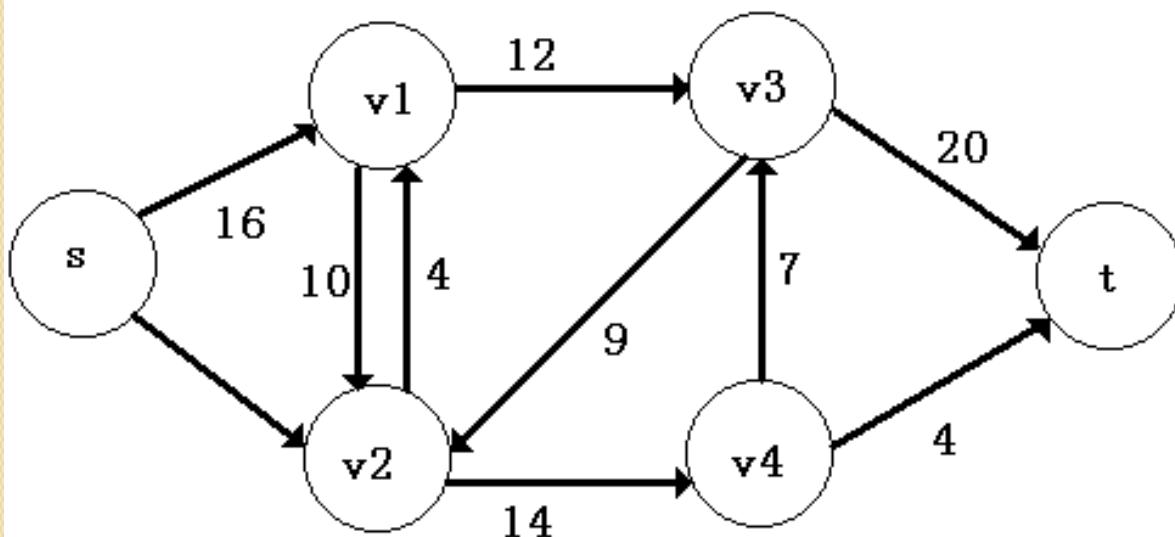


网络流算法

北京大学信息学院 郭炜

本讲义部分内容引自李晔晨叶天扬斯文俊等同学的讲义，以及网上讲义

- 给定一个有向图 $G = (V, E)$ ，把图中的边看作管道，每条边上有一个权值，表示该管道的流量上限。给定源点 s 和汇点 t ，现在假设在 s 处有一个水源， t 处有一个蓄水池，问从 s 到 t 的最大水流量是多少

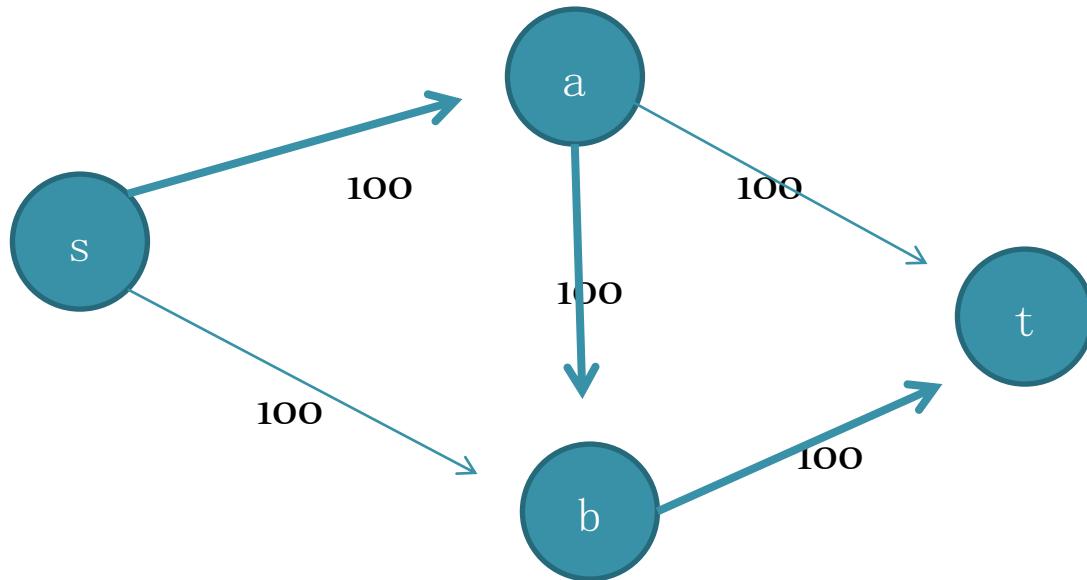


- 网络流图里，源点流出的量，等于汇点流入的量，除源汇外的任何点，其流入量之和等于流出两之和

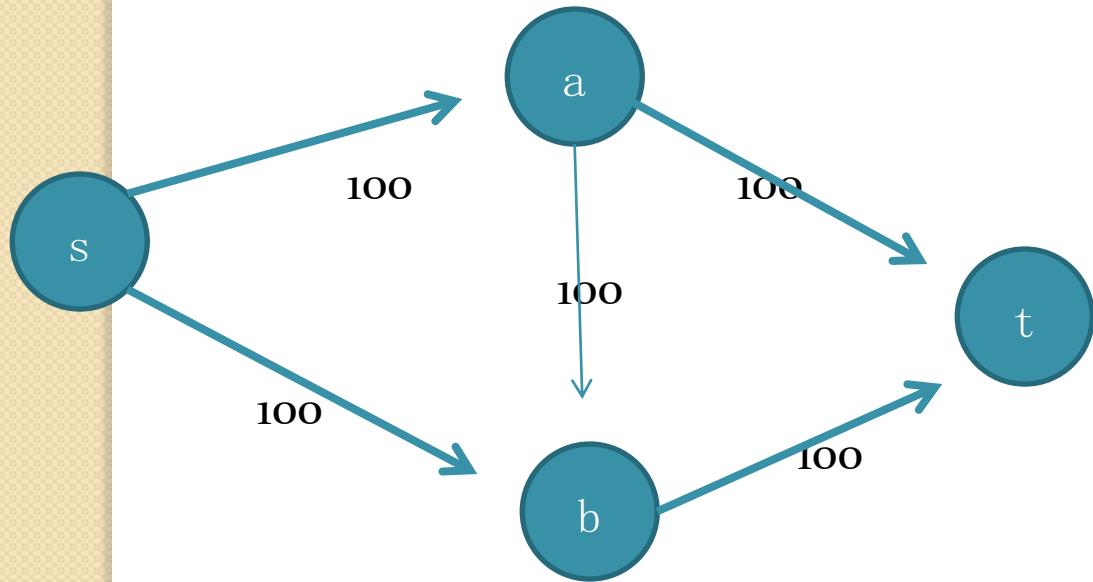
解决最大流的Ford-Fulkerson算法

- 基本思路很简单，每次用dfs从源到汇找一条可行路径，然后把这条路塞满。这条路径上容量最小的那条边的容量，就是这次dfs所找到的流量。然后对于路径上的每条边，其容量要减去刚才找到的流量。这样，每次dfs都可能增大流量，直到某次dfs找不到可行路径为止，最大流就求出来了

这个想法是否正确？



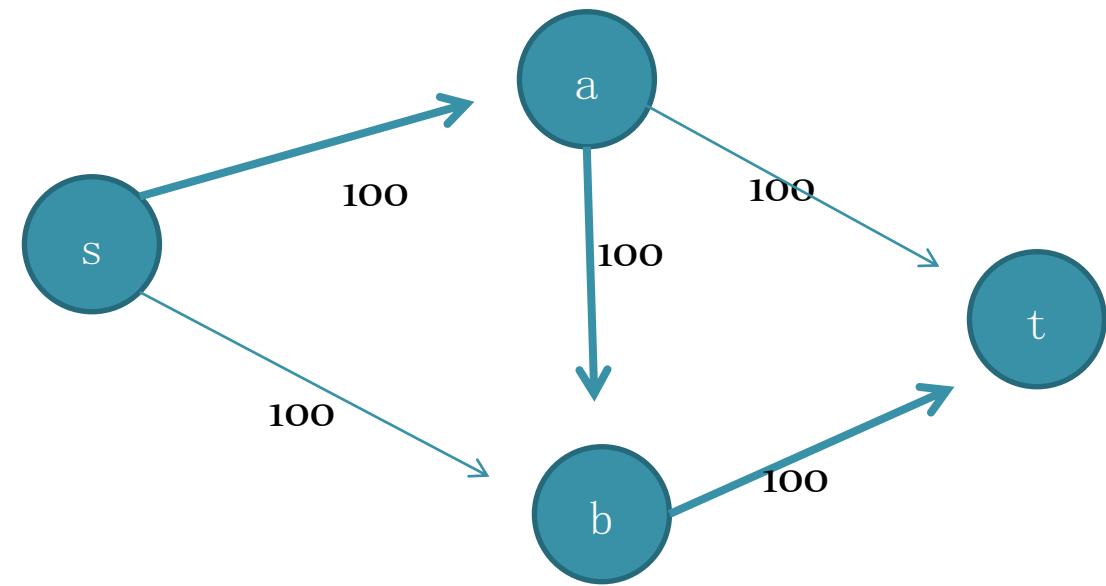
- 如果我们沿着s-a-b-t路线走 仅能得到一个100的流



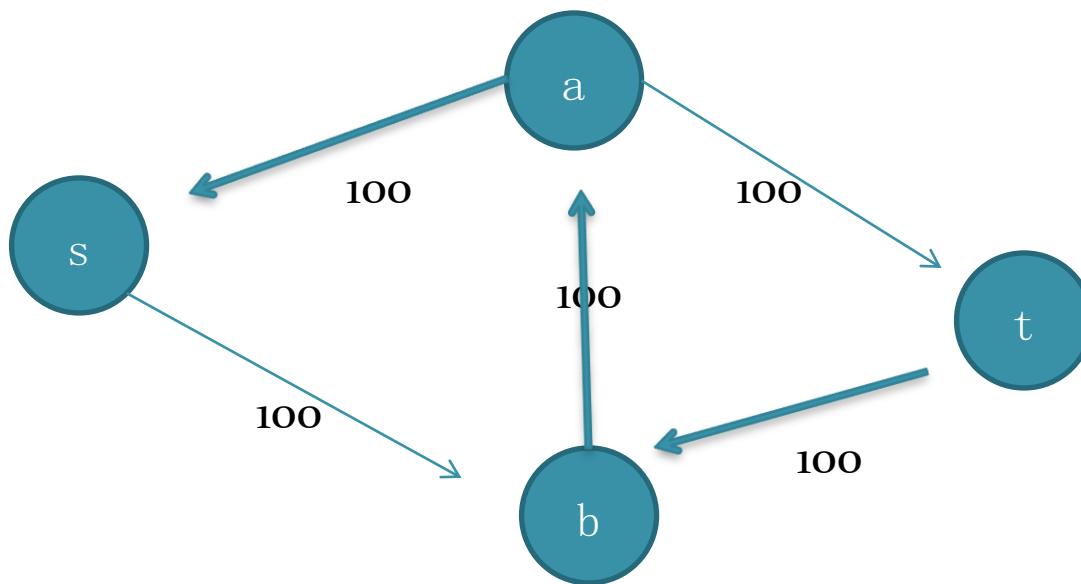
实际上此图存在流量
为200的流

- 问题出在过早地认为边 $a \rightarrow b$ 上流量不为0，因而“封锁”了流量继续增大的可能。
- 一个改进的思路：应能够修改已建立的流网络，使得“不合理”的流量被删掉。
- 一种实现：对上次dfs时找到的流量路径上的边，添加一条“反向”边，反向边上的容量等于上次dfs时找到的该边上的流量，然后再利用“反向”的容量和其他边上剩余的容量寻找路径。

第一次dfs后：

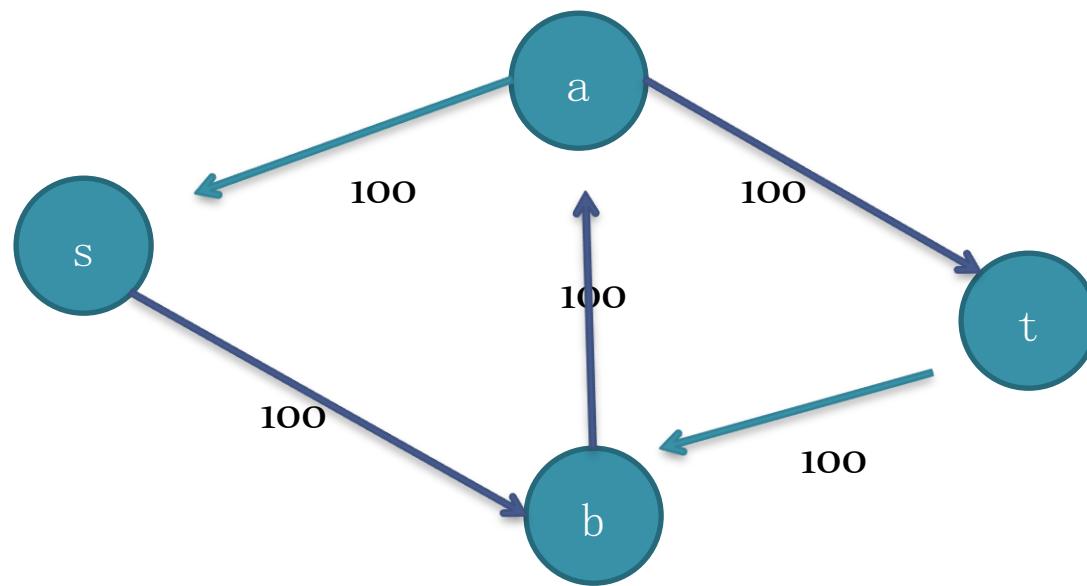


第一次dfs后，添加反向边得到的新图：

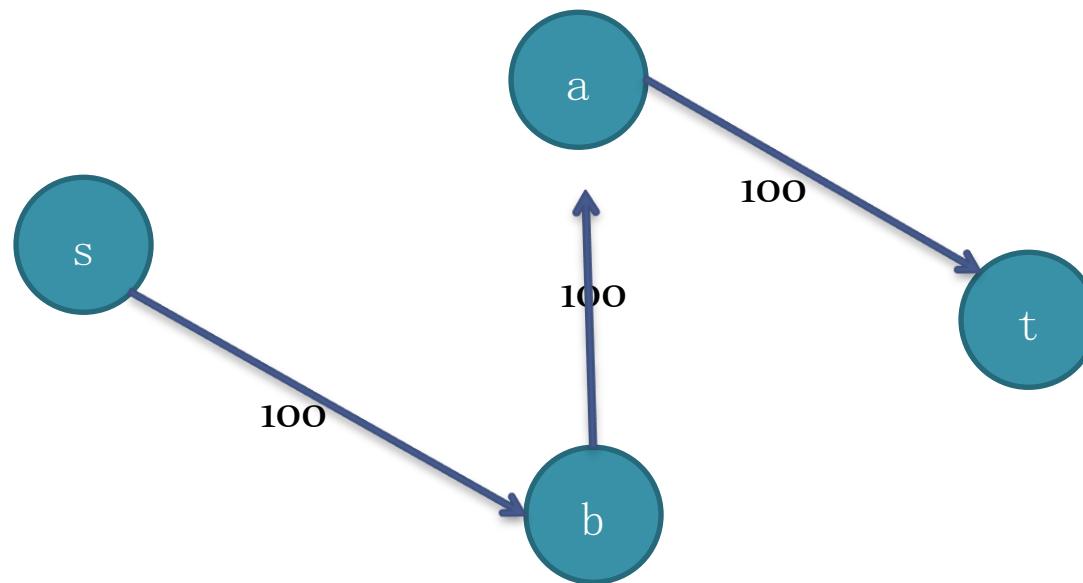


这样我们第二次dfs搜索的时候就可以在新的网络里找到新的路径

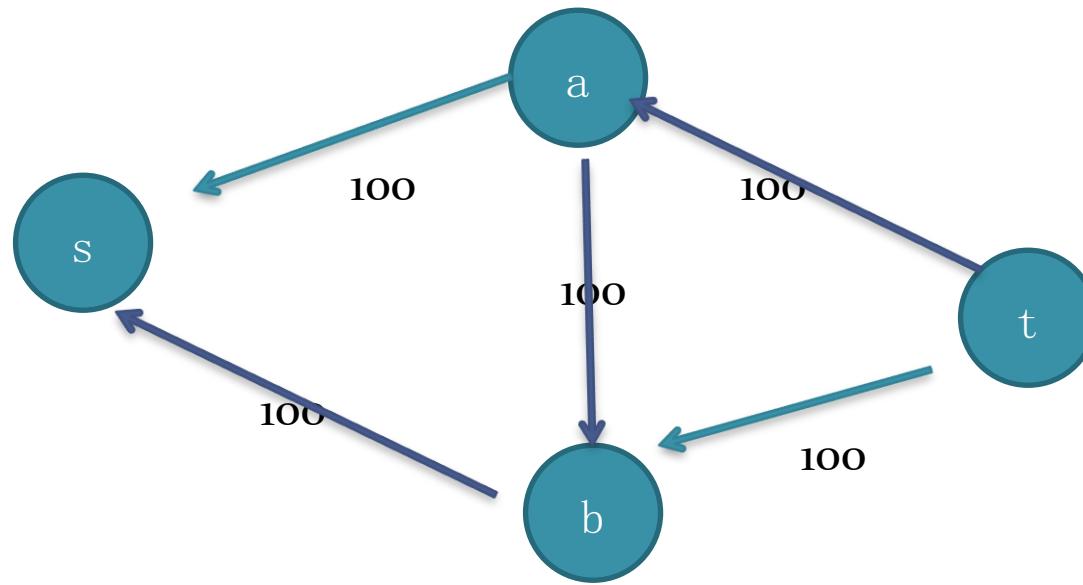
这是一个**取消流**的操作 也可以看作是两条路径的合并



第二次dfs搜索又找到了一个流量为100的流，加上第一次搜索得到的流量为100的流，总流量上升到200



再对第二次dfs后的图添加反向边， 变成：



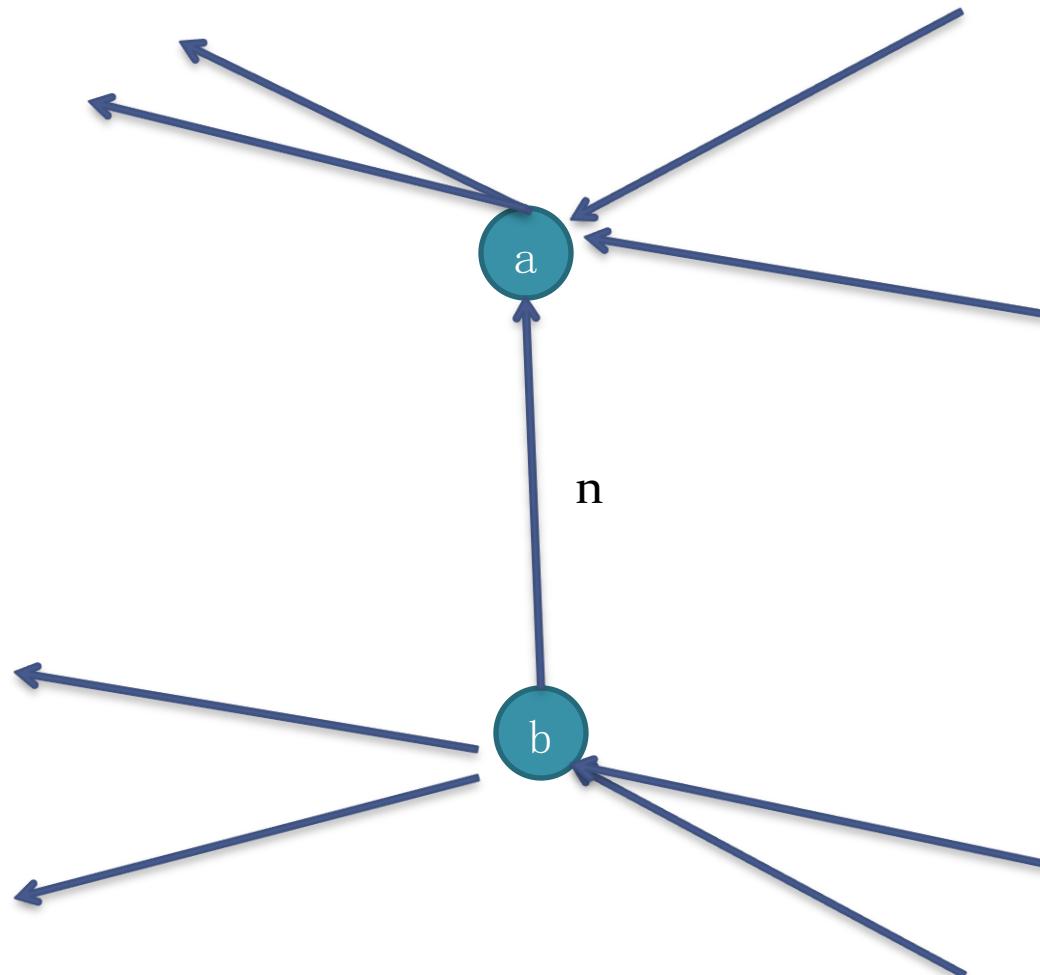
在此图上再次进行dfs,已经找不到路径了， 所以流量无法再增加， 最大流就是200

残余网络 (Residual Network)

- 在一个网络流图上，找到一条源到汇的路径（即找到了一个流量）后，对路径上所有的边，其容量都减去此次找到的流量，对路径上所有的边，都添加一条反向边，其容量也等于此次找到的流量，这样得到的新图，就称为原图的“残余网络”

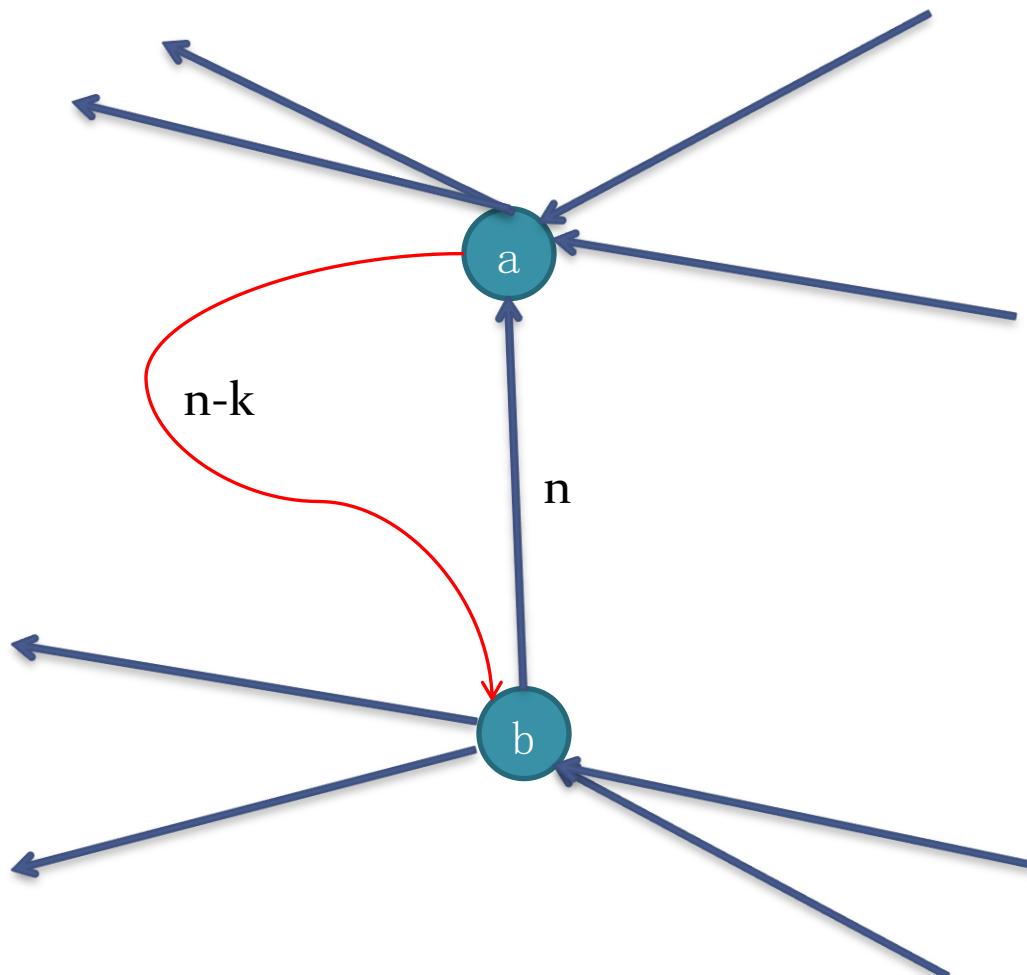
为什么添加反向边（取消流）是有效的？

假设在第一次寻找流的时候，发现在 $b \rightarrow a$ 上可以有流量n来自源，到达b，再流出a后抵达汇点。

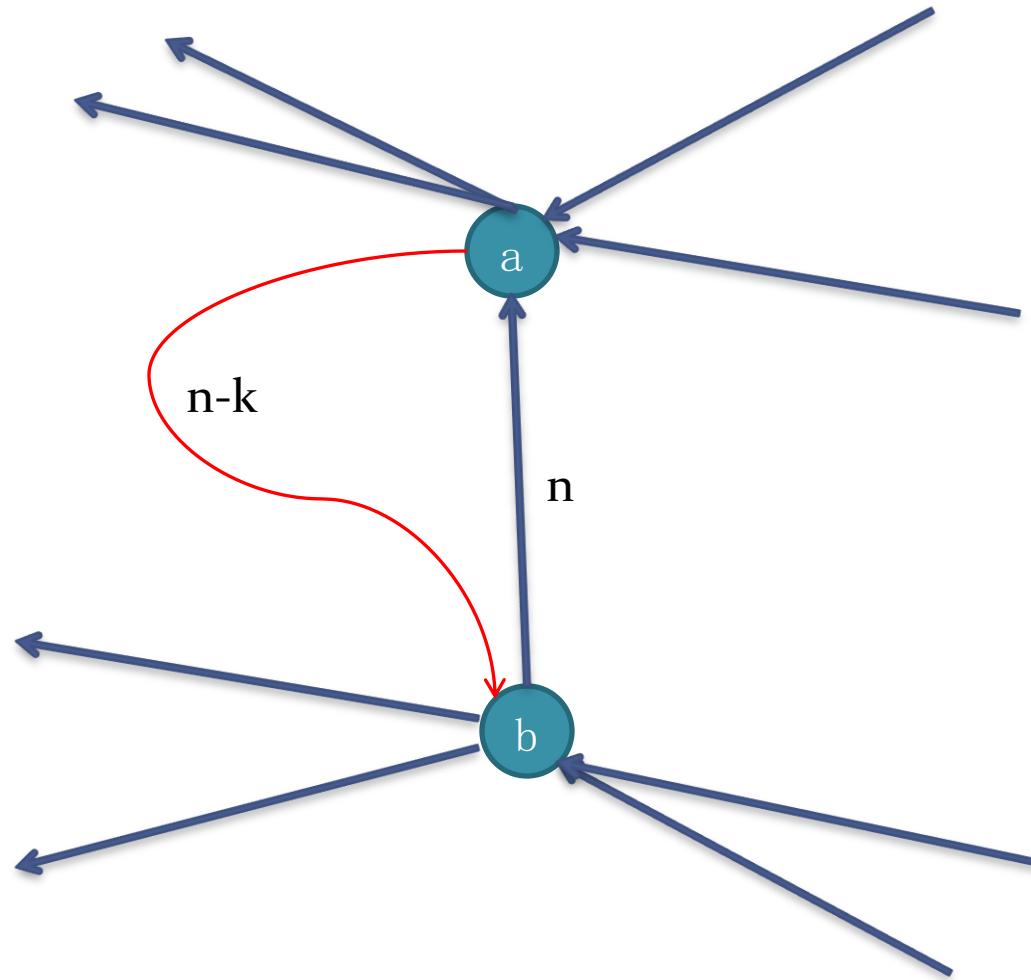


为什么添加反向边（取消流）是有效的？

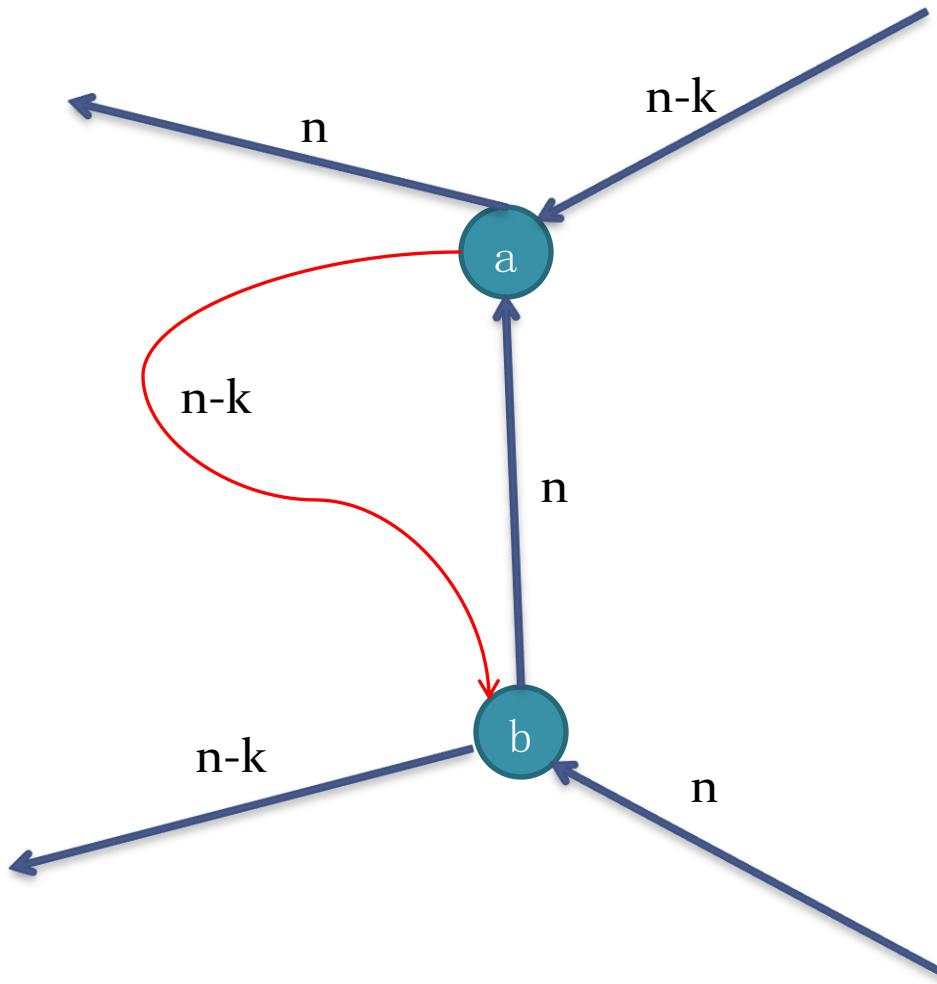
构建残余网络时添加反向边 $a \rightarrow b$, 容量是 n , 增广的时候发现了流量 $n-k$, 即新增了 $n-k$ 的流量。这 $n-k$ 的流量, 从 a 进, b 出, 最终流到汇



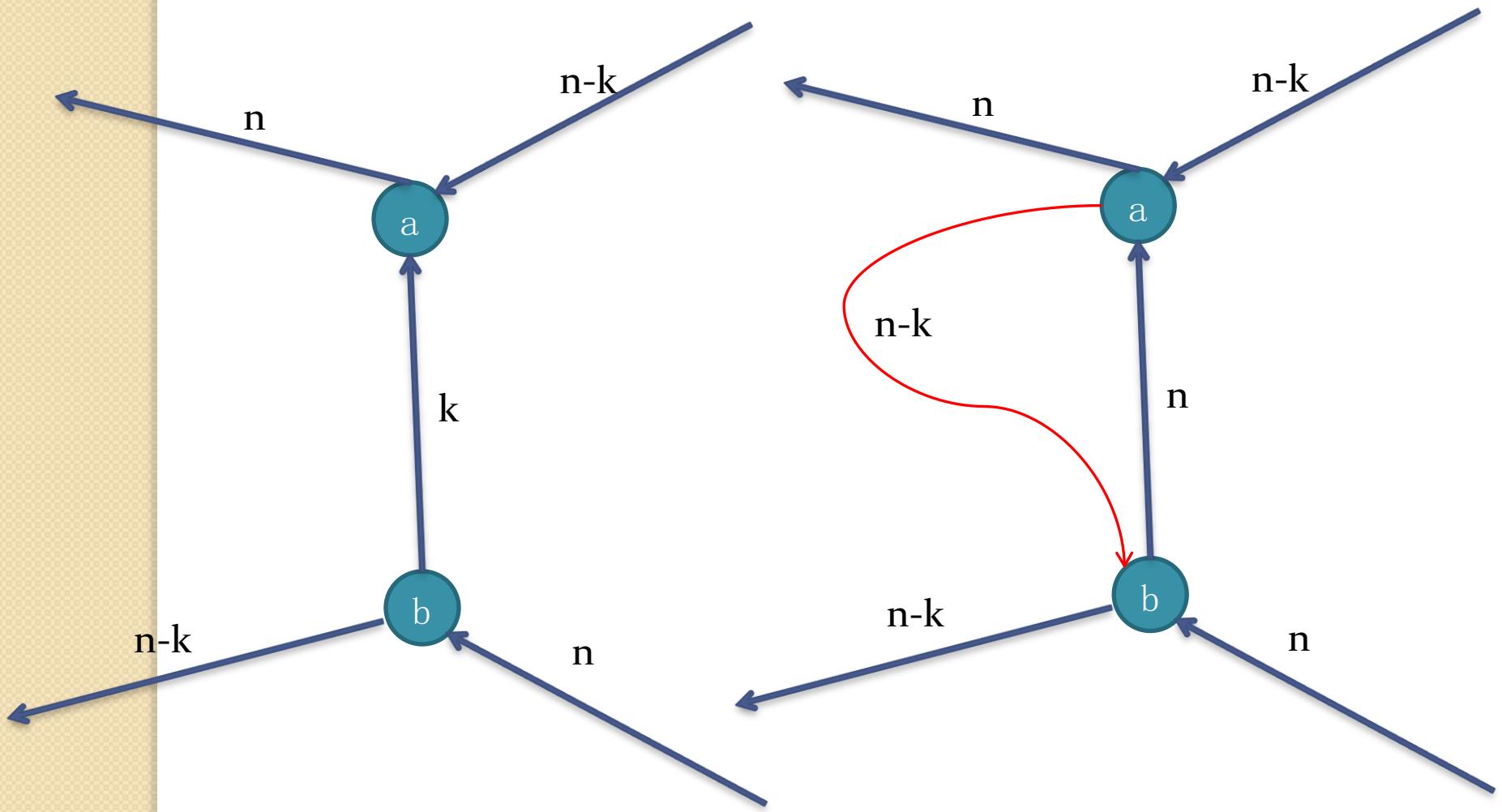
现要证明这 $2n-k$ 的从流量，在原图上确实是
可以从源流到汇的。



把流入b的边合并，看做一条，把流出a的边也合并，同理把流入a和流出b的边也都合并。



在原图上可以如下分配流量，则能有 $2n-k$ 从源流到汇点：



没有严格证明，只给一个感性认识。几条反向边相连的各种复杂情况也可以类似分析。

Ford-Fulkerson算法

- 求最大流的过程，就是不断找到一条源到汇的路径，然后构建残余网络，再在残余网络上寻找新的路径，使总流量增加，然后形成新的残余网络，再寻找新路径…直到某个残余网络上找不到从源到汇的路径为止，最大流就算出来了。
- 每次寻找新流量并构造新残余网络的过程，就叫做寻找流量的“增广路径”，也叫“增广”

现在假设每条边的容量都是整数

这个算法每次都能将流至少增加1

由于整个网络的流量最多不超过 图中所有的边的容量和C， 从而算法会结束

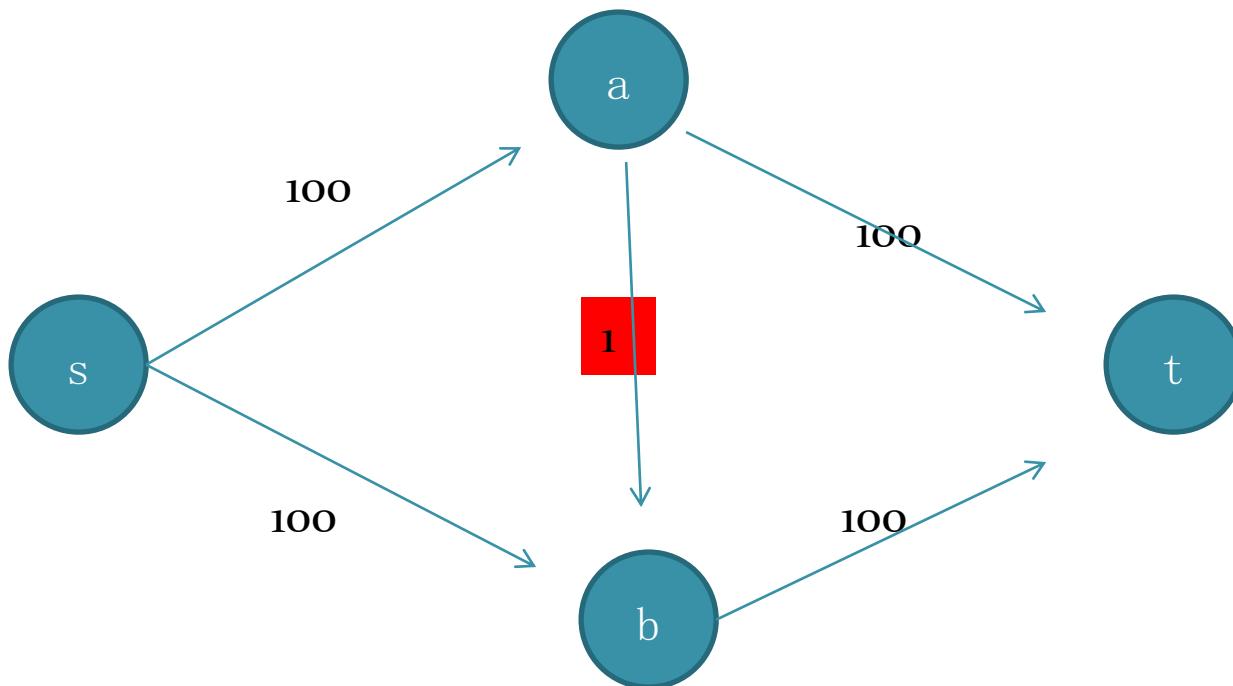
现在来看复杂度

找增广路径的算法可以用dfs， 复杂度为边数m+顶点数n

Dfs 最多运行C次

所以时间复杂度为 $C * (m+n) = C * n^2$

这个算法实现很简单
但是注意到在图中C可能很大很大
比如说下面这张图



如果运气不好 这种图会让你的程序执行200次dfs
虽然实际上最少只要2次我们就能得到最大流

如何避免上述的情况发生？

在每次增广的时候，选择从源到汇的具有最少边数的增广路径,即不是通过dfs寻找增广路径，而是通过bfs寻找增广路径。

这就是Edmonds-Karp 最短增广路算法

已经证明这种算法的复杂度上限为 nm^2 (n是点数， m是边数)

PoJ 1273 Drainage Ditches

赤裸裸的网络流题目。给定点数，边数，每条边的容量，以及源点，汇点，求最大流。

Sample Input

```
5 4
1 2 40
1 4 20
2 4 20
2 3 30
3 4 10
```

Sample Output

```
50
```

```
#include <iostream>
#include <queue>
using namespace std;
int G[300][300];
int Prev[300]; //路径上每个节点的前驱节点
bool Visited[300];
int n,m; //m是顶点数目， 顶点编号从1开始 1是源， m是汇, n是边数
unsigned Augment()
{
    int v;
    int i;
    deque<int> q;
    memset(Prev,0,sizeof(Prev));
    memset(Visited,0,sizeof(Visited));
    Prev[1] = 0;
    Visited[1] = 1;
    q.push_back(1);
    bool bFindPath = false;
    //用bfs寻找一条源到汇的可行路径
```

```
while( ! q.empty() ) {  
    v = q.front();  
    q.pop_front();  
    for( i = 1;i <= m;i ++) {  
  
        if( G[v][i] > 0 && Visited[i] == 0) {  
            //必须是依然有容量的边，才可以走  
            Prev[i] = v;  
            Visited[i] = 1;  
            if( i == m ) {  
                bFindPath = true;  
                q.clear();  
                break;  
            }  
            else  
                q.push_back(i);  
        }  
    }  
}
```

```
if( ! bFindPath)
    return 0;
int nMinFlow = 999999999;
v = m;
//寻找源到汇路径上容量最小的边， 其容量就是此次增加的总流量
while( Prev[v] ) {
    nMinFlow = min( nMinFlow,G[Prev[v]][v]);
    v = Prev[v];
}
//沿此路径添加反向边， 同时修改路径上每条边的容量
v = m;
while( Prev[v] ) {
    G[Prev[v]][v] -= nMinFlow;
    G[v][Prev[v]] += nMinFlow;
    v = Prev[v];
}
return nMinFlow;
}
```

```
int main()
{
    while (cin >> n >> m ) {
        //m是顶点数目， 顶点编号从1开始
        int i,j,k;
        int s,e,c;
        memset( G,0,sizeof(G));
        for( i = 0;i < n;i ++ ) {
            cin >> s >> e >> c;
            G[s][e] += c; //两点之间可能有多条边
        }
        unsigned int MaxFlow = 0;
        unsigned int aug;
        while( aug = Augment() )
            MaxFlow += aug;
        cout << MaxFlow << endl;
    }
    return 0;
}
```

Dinic 快速网络流算法

前面的网络流算法，每进行一次增广，都要做一遍BFS，十分浪费。能否少做几次BFS？

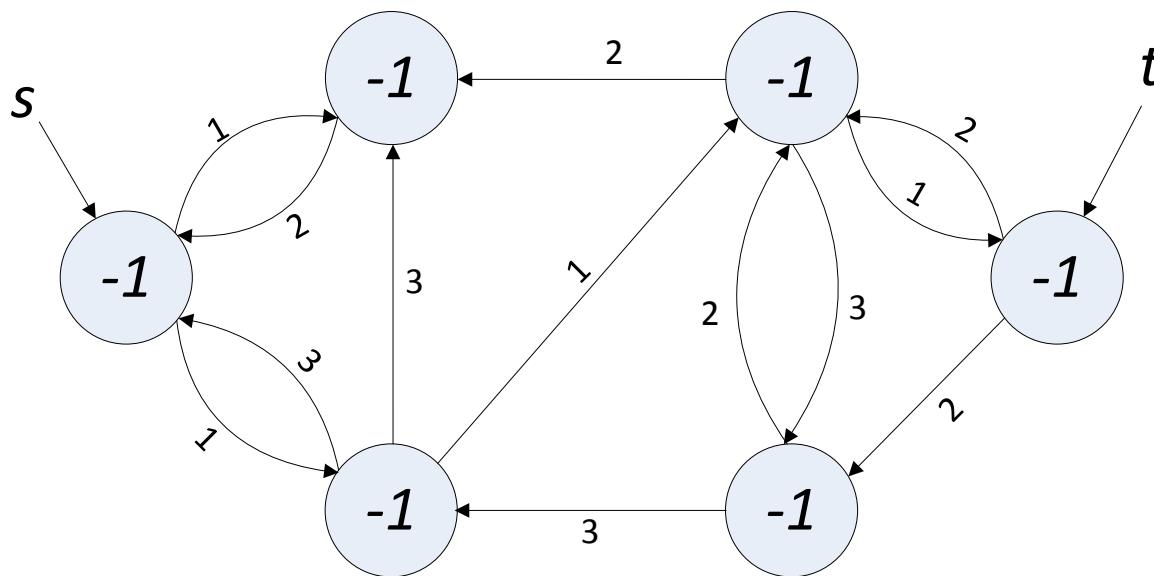
这就是Dinic算法要解决的问题

Dinic 算法

- *Edmonds-Karp*的提高余地：需要多次从 s 到 t 调用 BFS ，可以设法减少调用次数。
- 亦即：使用一种代价较小的高效增广方法。
- 考虑：在一次增广的过程中，寻找多条增广路径。
- DFS

Dinic 算法

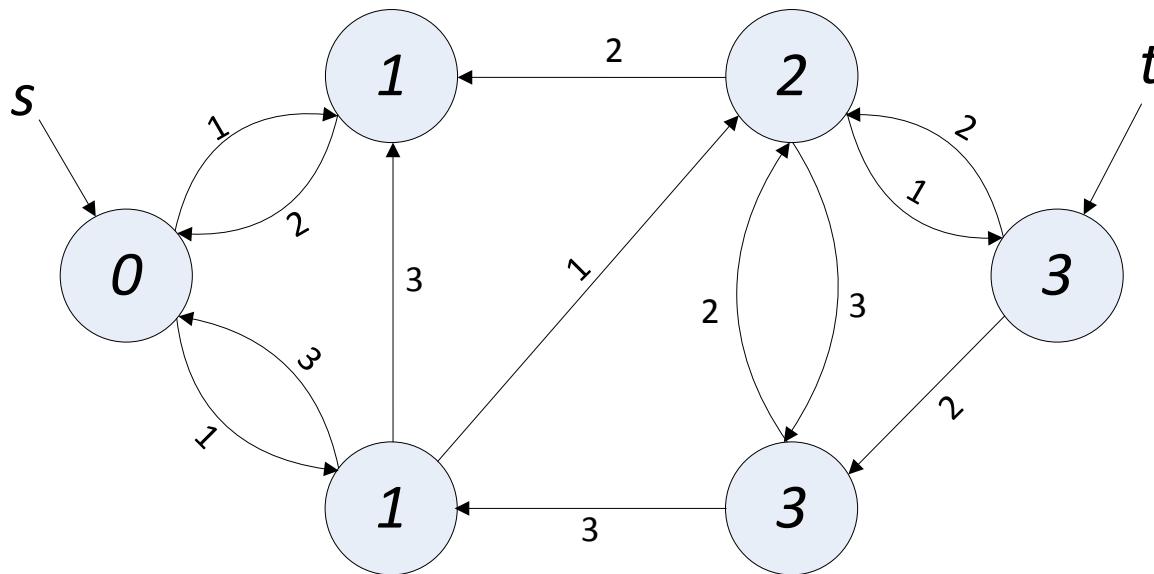
- 先利用 BFS 对残余网络分层



一个节点的“层”数，就是源点到它最少要经过的边数。

Dinic 算法

- 利用 BFS 对残余网络分层，分完层后，利用 DFS 从前一层向后一层反复寻找增广路。



一个节点的“层”数，就是源点到它最少要经过的边数。

Dinic 算法

分完层后，从源点开始，用DFS从前一层向后一层反复寻找增广路（即要求DFS的每一步都必须要走到下一层的节点）。

因此，前面在分层时，只要进行到汇点的层数被算出即可停止，因为按照该DFS的规则，和汇点同层或更下一层的节点，是不可能走到汇点的。

DFS过程中，要是碰到了汇点，则说明找到了一条增广路径。此时要增加总流量的值，消减路径上各边的容量，并添加反向边，即所谓的进行增广。

DFS找到一条增广路径后，并不立即结束，而是回溯后继续DFS寻找下一个增广路径。

回溯到哪个节点呢？

回溯到的节点 u 满足以下条件：

- 1) DFS搜索树的树边 (u, v) 上的容量已经变成0。即刚刚找到的增广路径上所增加的流量，等于 (u, v) 本次增广前的容量。(DFS的过程中，是从 u 走到更下层的 v 的)
- 2) u 是满足条件 1) 的最上层的节点

如果回溯到源点而且无法继续往下走了，DFS结束。

因此，一次DFS过程中，可以找到多条增广路径。

DFS结束后，对残余网络再次进行分层，然后再进行DFS

当残余网络的分层操作无法算出汇点的层次（即BFS到达不了汇点）时，算法结束，最大流求出。

一般用栈实现DFS，这样就能从栈中提取出增广路径。

Dinic 复杂度是 n^*n^*m (n 是点数， m 是边数)

要求出最大流中每条边的流量，怎么办？

要求出最大流中每条边的流量，怎么办？

将原图备份，原图上的边的容量减去做完最大流的残余网络上的边的剩余容量，就是边的流量。

```
#include <iostream> #include <queue> #include <vector> #include <algorithm> #include <deque> using namespace std;
#define INFINITE 999999999 //Poj 1273 Drainage Ditches 的 Dinic算法
int G[300][300];
bool Visited[300];
int Layer[300]; int n,m; //1是源点， m是汇点
bool CountLayer() {
    int layer = 0; deque<int>q;
    memset(Layer,0xff,sizeof(Layer)); //都初始化成-1
    Layer[1] = 0; q.push_back(1);
    while( ! q.empty() ) {
        int v = q.front();
        q.pop_front();
        for( int j = 1; j <= m; j ++ ) {
            if( G[v][j] > 0 && Layer[j] == -1 ) {
                //Layer[j] == -1 说明j还没有访问过
                Layer[j] = Layer[v] + 1;
                if( j == m ) //分层到汇点即可
                    return true;
                else
                    q.push_back(j);
            }
        }
    }
    return false;
}
```

```
int Dinic()
```

```
{  
    int i;      int s;  
    int nMaxFlow = 0;  
    deque<int> q;//DFS用的栈  
    while( CountLayer() ) { //只要能分层  
        q.push_back(1); //源点入栈  
        memset(Visited,0,sizeof(Visited)); Visited[1] = 1;  
        while( !q.empty() ) {  
            int nd = q.back();  
            if( nd == m ) { // nd是汇点  
                //在栈中找容量最小边  
                int nMinC = INFINITE;  
                int nMinC_vs; //容量最小边的起点  
                for( i = 1;i < q.size(); i ++ ) {  
                    int vs = q[i-1];  
                    int ve = q[i];  
                    if( G[vs][ve] > 0 ) {  
                        if( nMinC > G[vs][ve] ) {  
                            nMinC = G[vs][ve];  
                            nMinC_vs = vs;  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
//增广，改图
nMaxFlow += nMinC;
for( i = 1;i < q.size(); i ++ ) {
    int vs = q[i-1];
    int ve = q[i];
    G[vs][ve] -= nMinC; //修改边容量
    G[ve][vs] += nMinC; //添加反向边
}
//退栈到 nMinC_vs成为栈顶，以便继续dfs
while( !q.empty() && q.back() != nMinC_vs ) {
    Visited[q.back()] = 0; //没有这个应该也对
    q.pop_back();
}

}
else { //nd不是汇点
    for( i = 1;i <= m; i ++ ) {
        if( G[nd][i] > 0 && Layer[i] == Layer[nd] + 1 &&
            ! Visited[i]) {
                //只往下一层的没有走过的节点走
                Visited[i] = 1;
                q.push_back(i);
                break;
            }
        }
}
```

```
        if( i > m) //找不到下一个点
            q.pop_back(); //回溯
    }
}

return nMaxFlow;
}

int main()
{
    while (cin >> n >> m ) {
        int i,j,k;
        int s,e,c;
        memset( G,0,sizeof(G));
        for( i = 0;i < n;i ++ ) {
            cin >> s >> e >> c;
            G[s][e] += c; //两点之间可能有多条边
        }
        cout << Dinic() << endl;
    }

    return 0;
}
```

POJ 3436 ACM Computer Factory

电脑公司生产电脑有N个机器， 每个机器单位时间产量为Qi。

电脑由P个部件组成， 每个机器工作时只能把有某些部件的半成品电脑(或什么都没有的空电脑) 变成有另一些部件的半成品电脑或完整电脑(也可能移除某些部件)。求电脑公司的单位时间最大产量， 以及哪些机器有协作关系， 即一台机器把它的产品交给哪些机器加工。

POJ 3436 ACM Computer Factory

Sample input

3	4					
15	0	0	0	0	1	0
10	0	0	0	0	1	1
30	0	1	2	1	1	1
3	0	2	1	1	1	1

Sample output

25	2	
1	3	15
2	3	10

输入：电脑由3个部件组成，共有4台机器，1号机器产量15，能给空电脑加上2号部件，2号机器能给空电脑加上2号部件和3号部件，3号机器能把有1个2号部件和3号部件有无均可的电脑变成成品（每种部件各有一个）

输出：单位时间最大产量25，有两台机器有协作关系，
1号机器单位时间内要将15个电脑给3号机器加工
2号机器单位时间内要将10个电脑给3号机器加工

建模分析：

每个工厂有三个动作：

1)接收原材料

2)生产

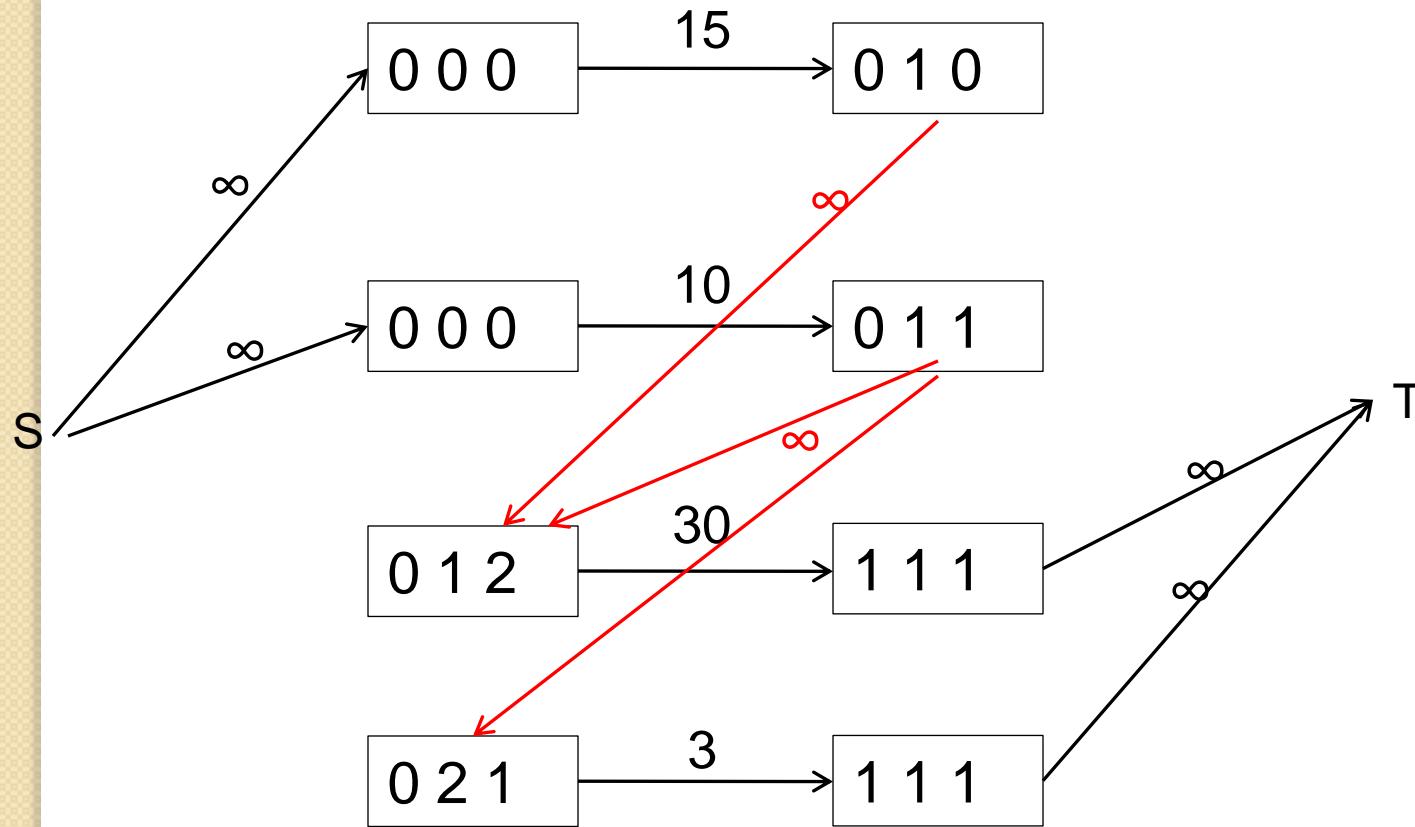
3)将其产出的半成品给其他机器，或产出成品。

这三个过程都对应不同的流量。

网络流模型：

- 1) 添加一个原点 **S**, **S** 提供最初的原料 **00000...**
- 2) 添加一个汇点 **T**, **T** 接受最终的产品 **11111....**
- 3) 将每个机器拆成两个点：编号为 **i** 的接收节点，和编号为 **i+n** 的产出节点（**n** 是机器数目），前者用于接收原料，后者用于提供加工后的半成品或成品。这两个点之间要连一条边，容量为单位时间产量 **Qi**
- 4) **S** 连边到所有接收 "**0000...**" 或 "若干个0及若干个2" 的机器，容量为无穷大
- 5) 产出节点连边到能接受其产品的接收节点，容量无穷大
- 6) 能产出成品的节点，连边到 **T**，容量无穷大。
- 7) 求 **S** 到 **T** 的最大流

15	0	0	0	0	1	0
10	0	0	0	0	1	1
30	0	1	2	1	1	1
3	0	2	1	1	1	1



Trick: S 可以连边到222

poj 2112 Optimal Milking

有K台挤奶机器和C头牛(统称为物体)，每台挤奶机器只能容纳M头牛进行挤奶。现在给出 $dis[K + C][K + C]$ 的矩阵， $dis[i][j]$ 若不为0则表示第i个物体到第j个物体之间有路， $dis[i][j]$ 就是该路的长度。 $(1 \leq K \leq 30, 1 \leq C \leq 200)$

现在问你怎么安排这C头牛到K台机器挤奶，使得需要走最长路程到挤奶机器的奶牛所走的路程最少，求出这个最小值。

Sample Input

2 3 2 // K C M

0 3 2 1 1

3 0 3 2 0

2 3 0 1 0

1 2 1 0 2

1 0 0 2 0

Sample Output

2

利用Floyd算法求出每个奶牛到每个挤奶机的最短距离。

则题目变为：

已知C头奶牛到K个挤奶机的距离，每个挤奶机只能有M个奶牛，每个奶牛只能去一台挤奶机，求这些奶牛到其要去的挤奶机距离的最大值的最小值。

网络流模型：

每个奶牛最终都只能到达一个挤奶器，每个挤奶器只能有M个奶牛，可把奶牛看做网络流中的流。

每个奶牛和挤奶器都是一个节点，添加一个源，连边到所有奶牛节点，这些边容量都是1。

添加一个汇点，每个挤奶器都连边到它。这些边的容量都是M。

网络流模型：

先假定一个最大距离的的最小值 maxdist , 在上述图中, 如果奶牛节点*i*和挤奶器节点*j*之间的距离 $\leq \text{maxdist}$, 则从*i*节点连一条边到*j*节点, 表示奶牛*i*可以到挤奶器*j*去挤奶。该边容量为1。该图上的最大流如果是C(奶牛数) , 那么就说明假设的 maxdist 成立, 则减小 maxdist 再试

总之, 要二分 maxdist , 对每个 maxdist 值, 都重新构图, 看其最大流是否是C, 然后再决定减少或增加 maxdist

Poj 1149 pigs

- 题目大意
- Mirko养着一些猪 猪关在一些猪圈里面 猪圈是锁着的 他自己没有钥匙（汗）
- 只有要来买猪的顾客才有钥匙
- 顾客依次来 每个顾客会用他的钥匙打开一些猪圈 买走一些猪 然后锁上
- 在锁上之前 Mirko有机会重新分配这几个已打开猪圈的猪
- 现在给出一开始每个猪圈的猪数 每个顾客所有的钥匙和要买走的猪数 问Mirko最多能卖掉几头猪

● 数据规模

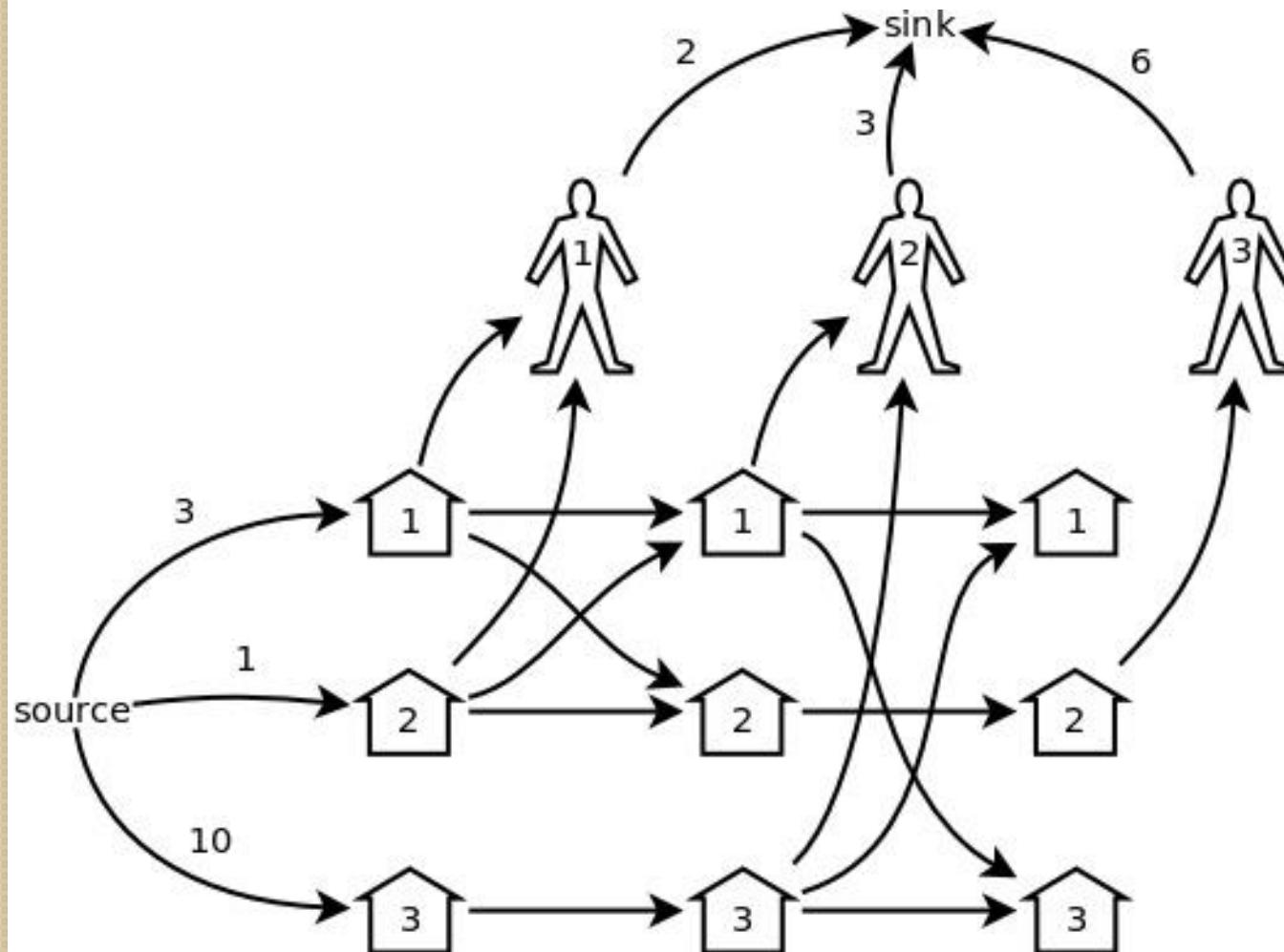
- 猪圈数 $n \leq 1000$
- 顾客数 $m \leq 100$
- 每个猪圈的猪数不超过 1000
- 假设猪圈容量无限

样例

- 3 3
- 3 1 10
- 2 1 2 2
- 2 1 3 3
- 1 2 6
- 3个猪圈分别有3, 1, 10头猪
- 第一个顾客有1, 2猪圈的钥匙要买2头
- 第二个顾客有1, 3猪圈的钥匙要买3头
- 第三个顾客有2猪圈的钥匙要买6头

- 以下本题内容引自
- <http://imlazy.ycool.com/post.2059102.html>

上面的样例可以构造出下面的模型，（图中凡是没有标数字的边，容量都是 $+\infty$ ）：



三个顾客，就有三轮交易，每一轮分别都有 3 个猪圈和 1 个顾客的节点。

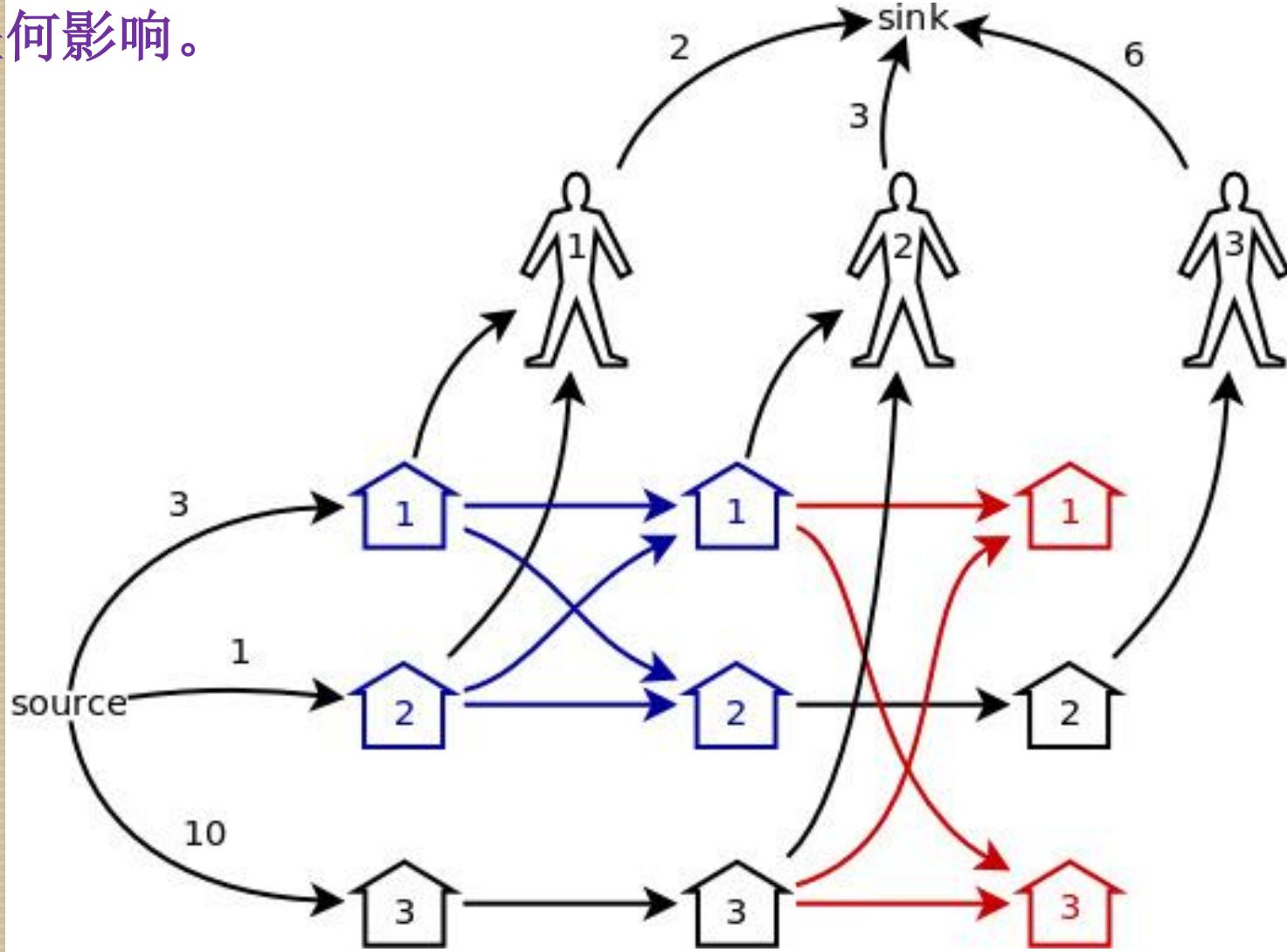
- 1) 三个顾客，就有三轮交易，每一轮分别都有 3 个猪圈和 1 个顾客的节点。
- 2) 从源点到第一轮的各个猪圈各有一条边，容量就是各个猪圈里的猪的初始数量。
- 3) 从各个顾客到汇点各有一条边，容量就是各个顾客能买的数量上限。
- 4) 在某一轮中，从该顾客打开的所有猪圈都有一条边连向该顾客，容量都是 $+\infty$ 。
- 5) 最后一轮除外，从每一轮的 i 号猪圈都有一条边连向下一轮的 i 号猪圈，容量都是 $+\infty$ ，表示这一轮剩下的猪可以留到下一轮。
- 6) 最后一轮除外，从每一轮被打开的所有猪圈，到下一轮的同样这些猪圈，两两之间都要连一条边，容量 $+\infty$ 表示它们之间可以任意流通。

这个网络模型的最大流量就是最多能卖出的数量。图中最多有 $2 + N + M \times N \approx 100,000$ 个节点。

这个模型虽然很直观，但是节点数太多了，计算速度肯定会很慢。其实不用再想别的算法，就让我们继续上面的例子，用合并的方法来简化这个网络模型。

首先，最后一轮中没有打开的猪圈就可以从图中删掉了，也就是下面图中红色的部分，显然它们对整个网络的流量没有任何影响。

首先，最后一轮中没有打开的猪圈就可以从图中删掉了，也就是下面图中红色的部分，显然它们对整个网络的流量没有任何影响。



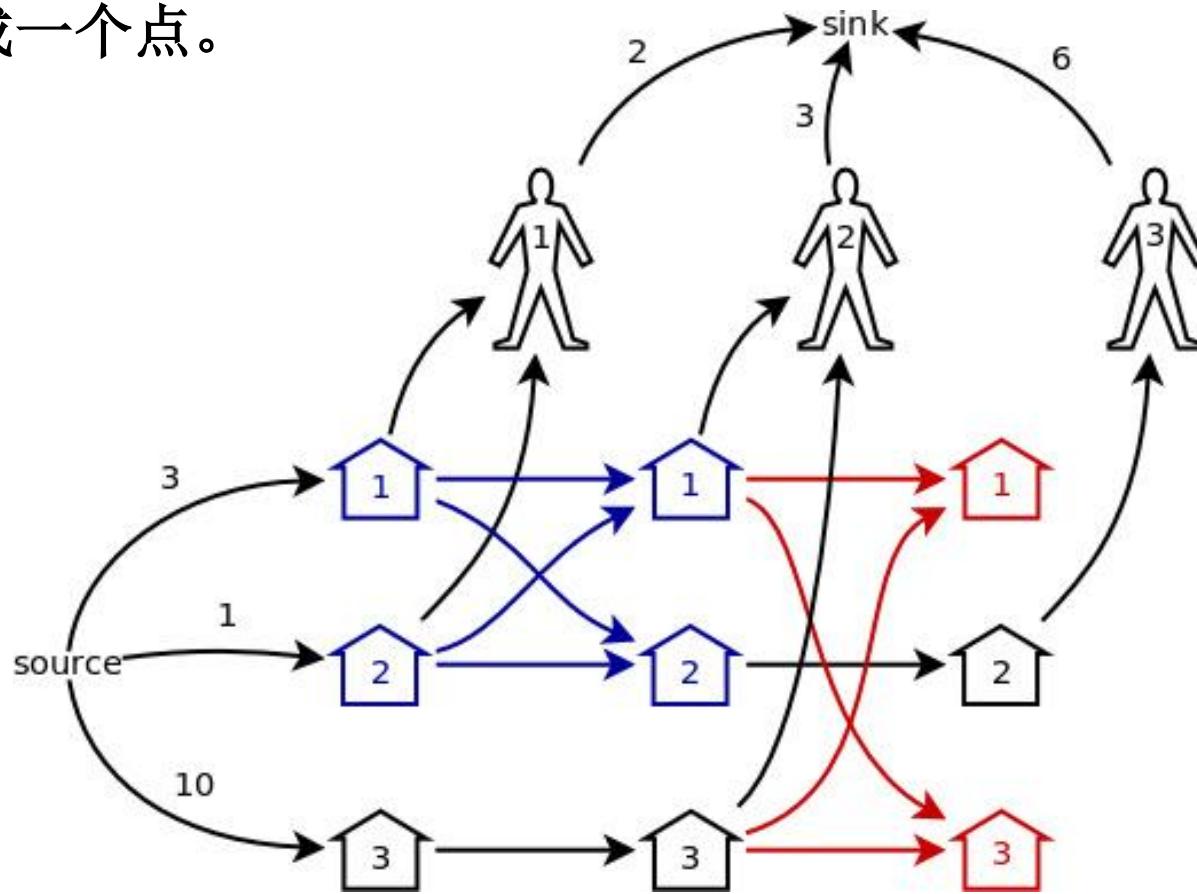
用以下3条规律合并节点：

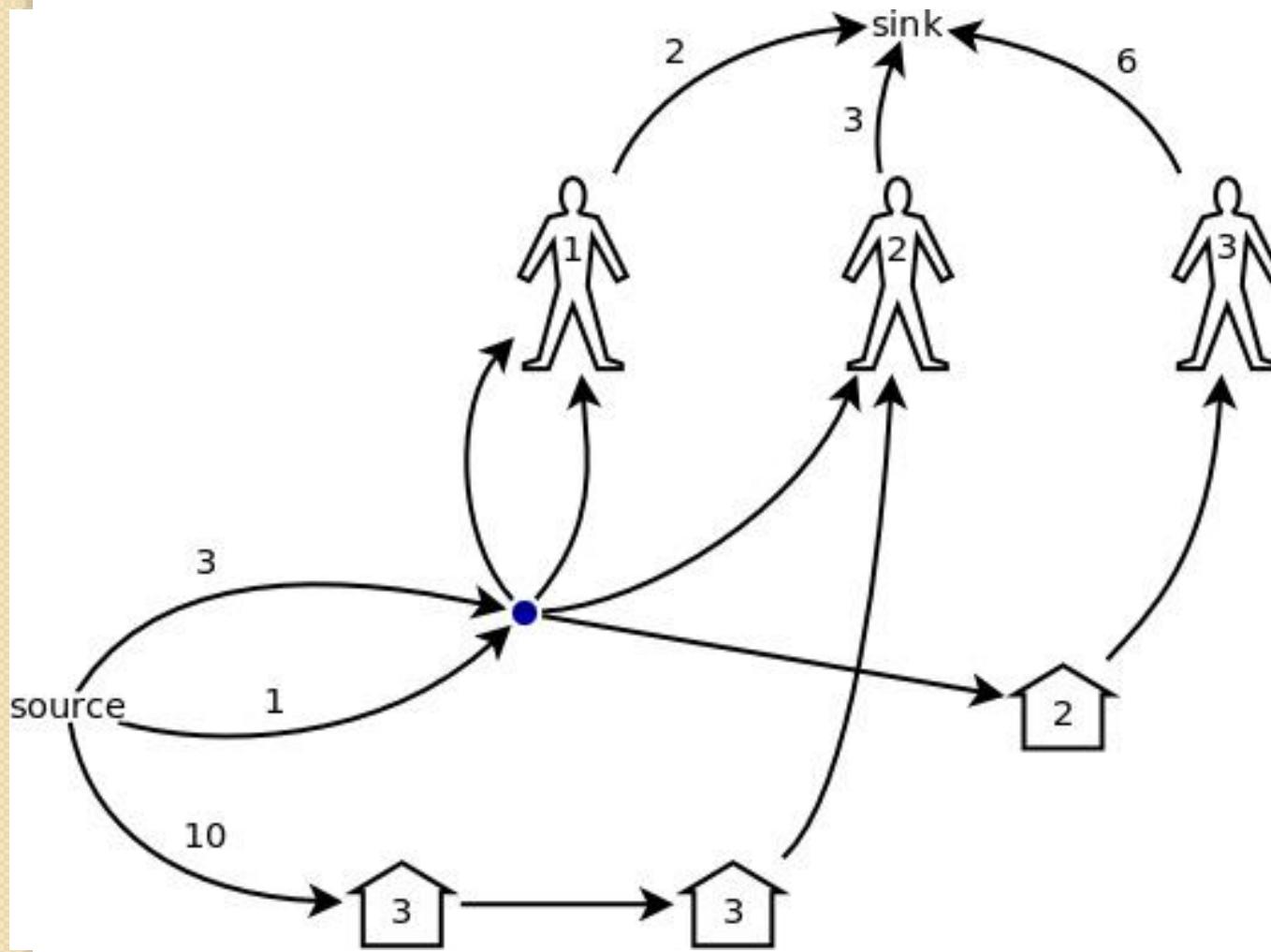
规律 1. 如果几个节点的流量的来源完全相同，则可以把它们合并成一个。

规律 2. 如果几个节点的流量的去向完全相同，则可以把它们合并成一个。

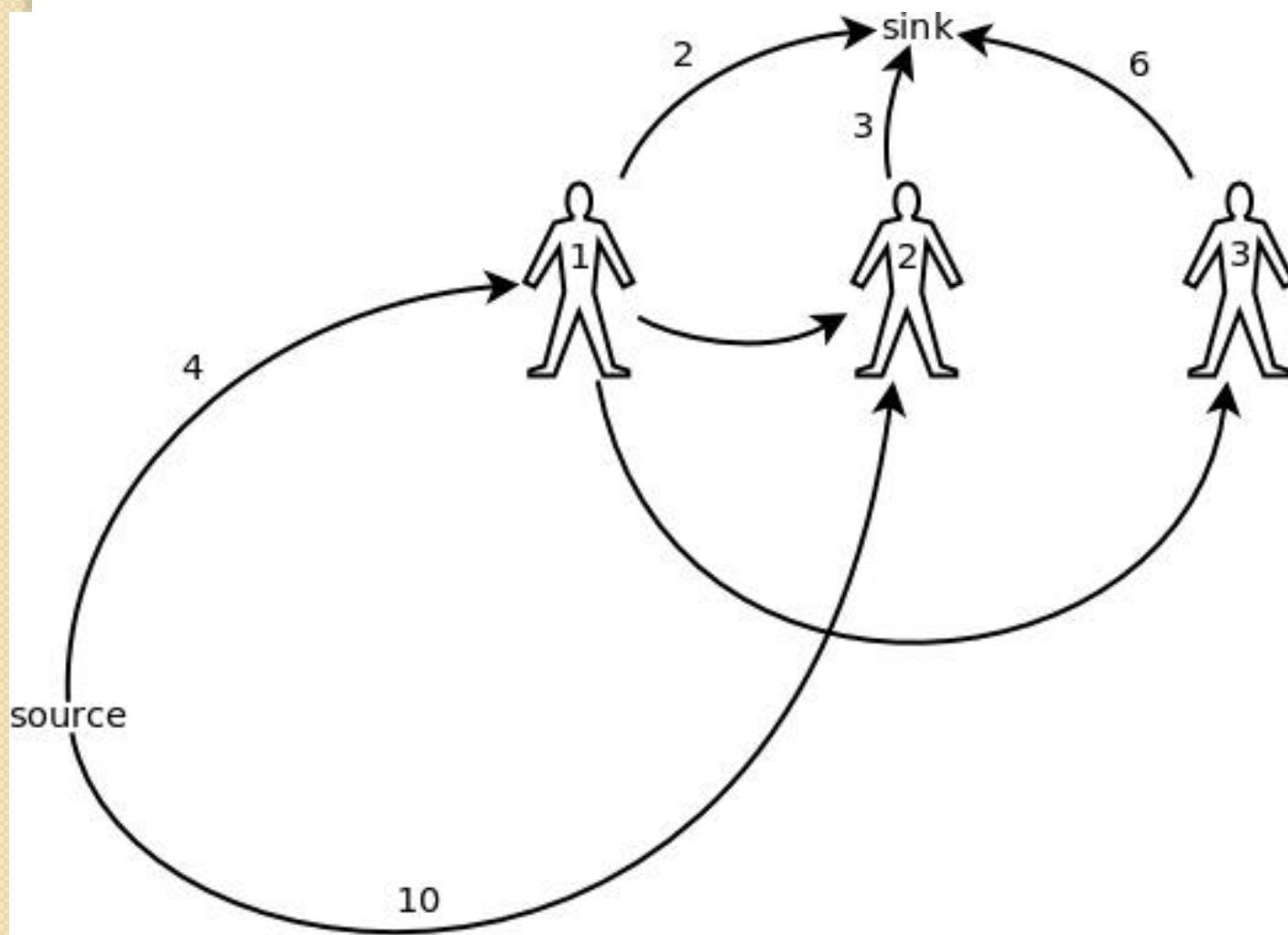
规律 3. 如果从点 u 到点 v 有一条容量为 $+\infty$ 的边，并且 u 是 v 的唯一流量来源，或者 v 是 u 的唯一流量去向，则可以把 u 和 v 合并成一个节点。

根据规律1，可以把蓝色部分右边的1、2号节点合并成一个；根据规律2，可以把蓝色部分左边的1、2号节点合并成一个；最后，根据规律3，可以把蓝色部分的左边和右边（已经分别合并成了一个节点）合并成一个节点。于是，会得到下页的图。也就是说，最后一轮除外，每一轮被打开的猪圈和下一轮的同样这些猪圈都可以被合并成一个点。





接着，根据规律3，此中的蓝色节点、2号猪圈和1号顾客这三点可以合并成一个；两个3号猪圈和2号顾客也可以合并成一个点。当然，如果两点之间有多条同向的边，则这些边可以合并成一条，容量相加。最终，上例中的网络模型被简化成了下页图的样子。



从上图 重新总结一下构造这个网络流模型的规则：

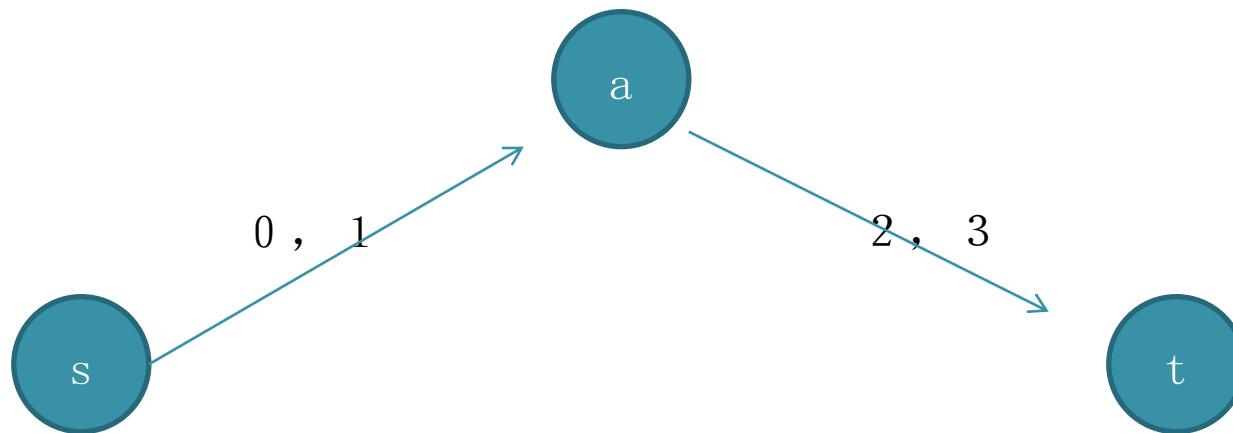
- 1) 每个顾客分别用一个节点来表示。
- 2) 对于每个猪圈的第一个顾客，从源点向他连一条边，容量就是该猪圈里的猪的初始数量。如果从源点到一名顾客有多条边，则可以把它合并成一条，容量相加。
- 3) 对于每个猪圈，假设有 n 个顾客打开过它，则对所有整数 $i \in [1, n]$ ，从该猪圈的第 i 个顾客向第 $i + 1$ 个顾客连一条边，容量为 $+\infty$ 。
- 4) 从各个顾客到汇点各有一条边，容量是各个顾客能买的数量上限。

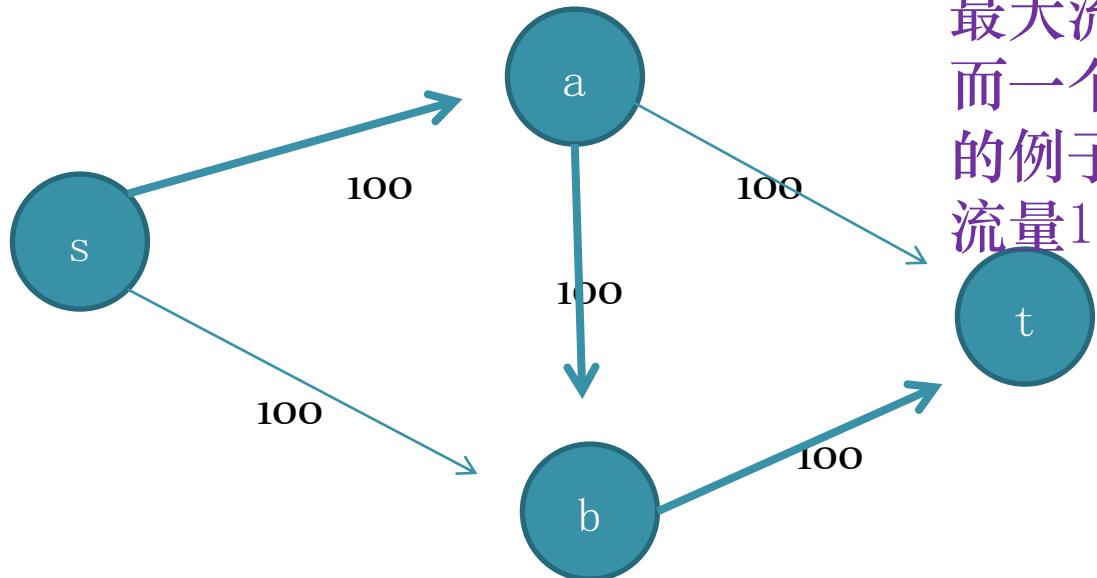
新的网络模型中最多只有 $2 + N = 102$ 个节点，计算速度就可以相当快了。可以这样理解这个新的网络模型：对于某一个顾客，如果他打开了猪圈 h ，则在他走后，他打开的所有猪圈里剩下的猪都有可能被换到 h 中，因而这些猪都有可能被 h 的下一个顾客买走。所以对于一个顾客打开的所有猪圈，从该顾客到各猪圈的下一个顾客，都要连一条容量为 $+\infty$ 的边。

在面对网络流问题时，如果一时想不出很好的构图方法，不如先构造一个最直观的模型，然后再用合并节点和边的方法来简化这个模型。经过简化以后，好的构图思路自然就会涌现出来了。这是解决网络流问题的一个好方法。

有流量下界的网络最大流

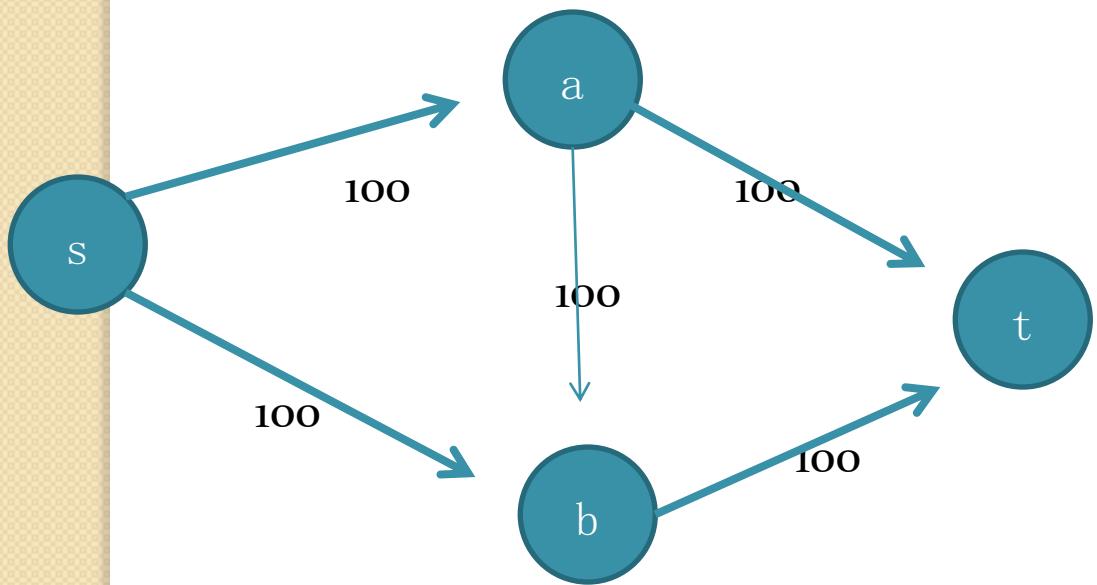
- 如果流网络中每条边 e 对应两个数字 $B(e)$ 和 $C(e)$ ，分别表示该边上的流量至少要是 $B(e)$ ，最多 $C(e)$ ，那么，在这样的流网络上求最大流，就是有下界的最大流问题。
- 这种网络不一定存在可行流，如：





最大流不满足限制条件，
而一个非最大流满足条件
的例子（假定ab至少要有
流量100）

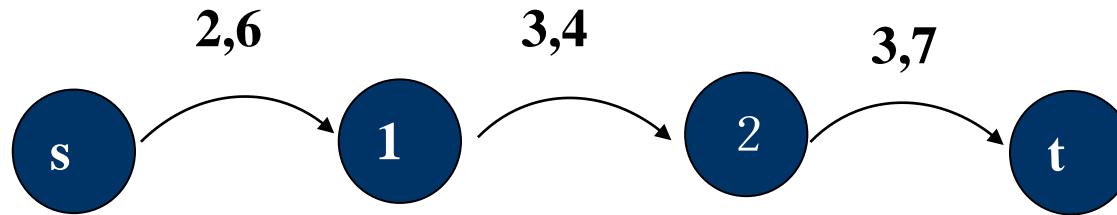
- 如果我们沿着s-a-b-t路线走 仅能得到一个100的流



实际上此图存在流量
为200的流

有流量下界的网络最大流

- 思路：将下界“分离”出去，使问题转换为下界为0的普通网络流问题。

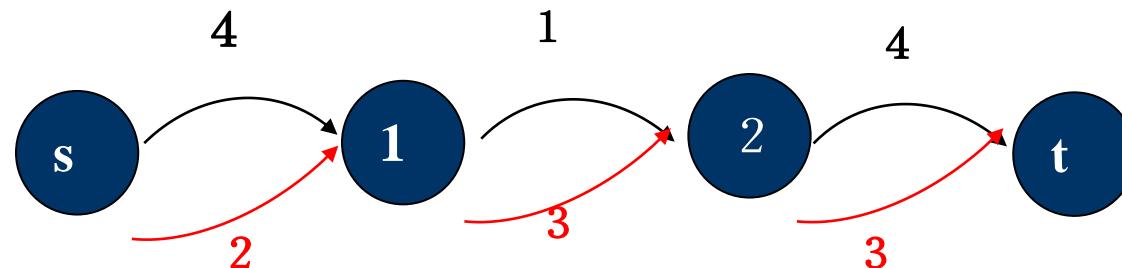


将原弧 (u,v) 分离出一条**必要弧**和一条非必要弧：

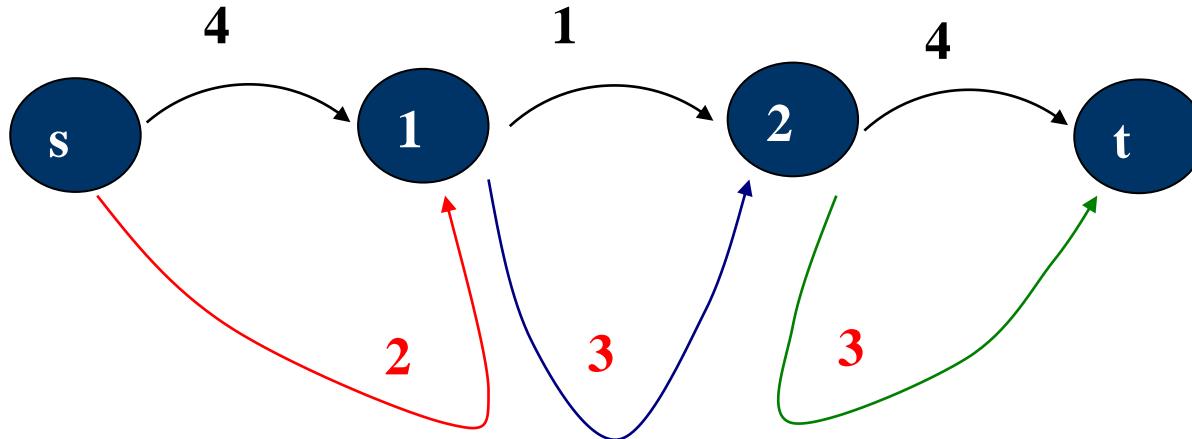
假设 $B(u,v)$ 是下界，则分离出两条弧：

$C_1(u,v) = B(u,v)$ -- 必要弧

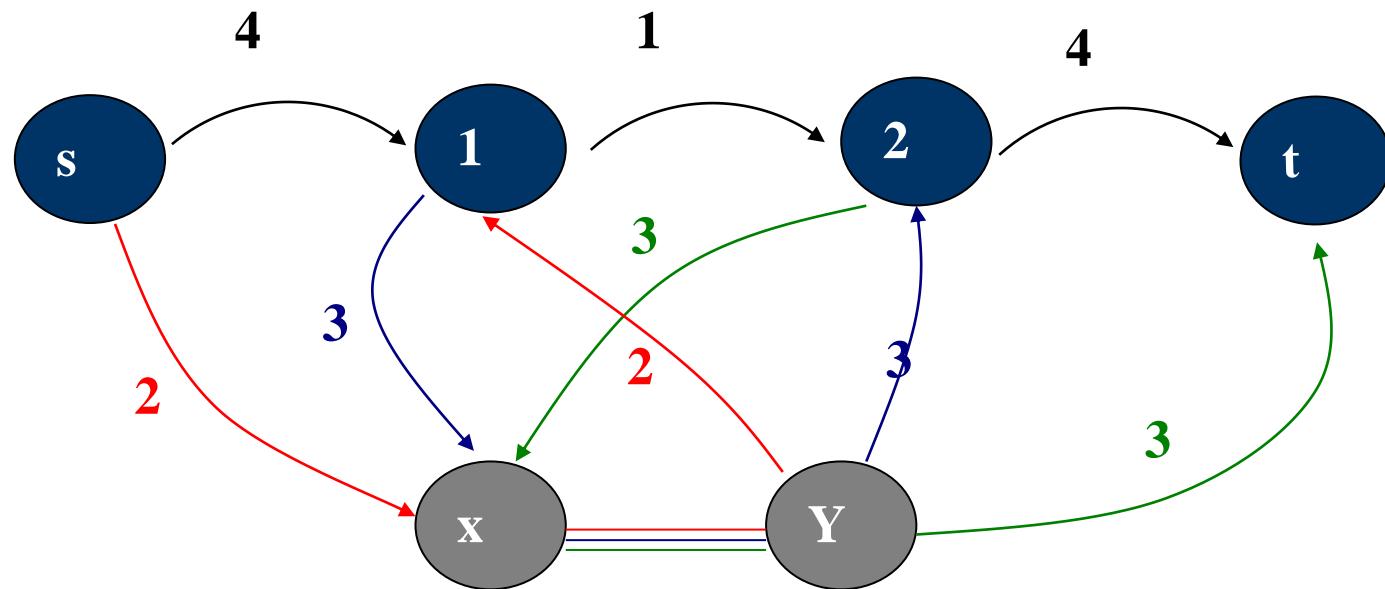
$C_2(u,v) = C(u,v) - B(u,v)$



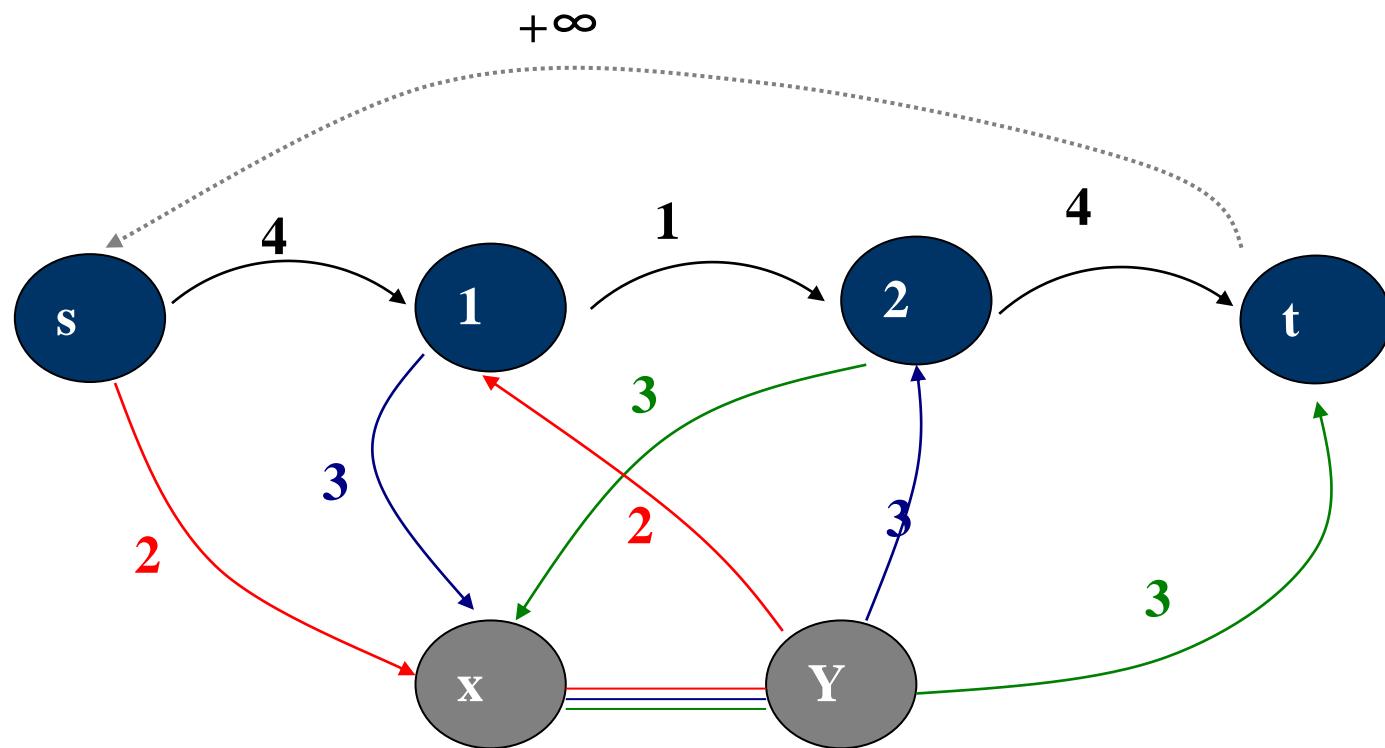
由于必要弧的有一定要满流的限制，将必要弧“拉”出来集中考虑：



添加附加点 x, y 。想像一条不限上界的 (x, y) , 用必要弧将它们“串”起来, 即对于有向必要弧 (u, v) , 添加 $(u, x), (y, v)$, 容量为必要弧容量。这样就建立了一个等价的网络。

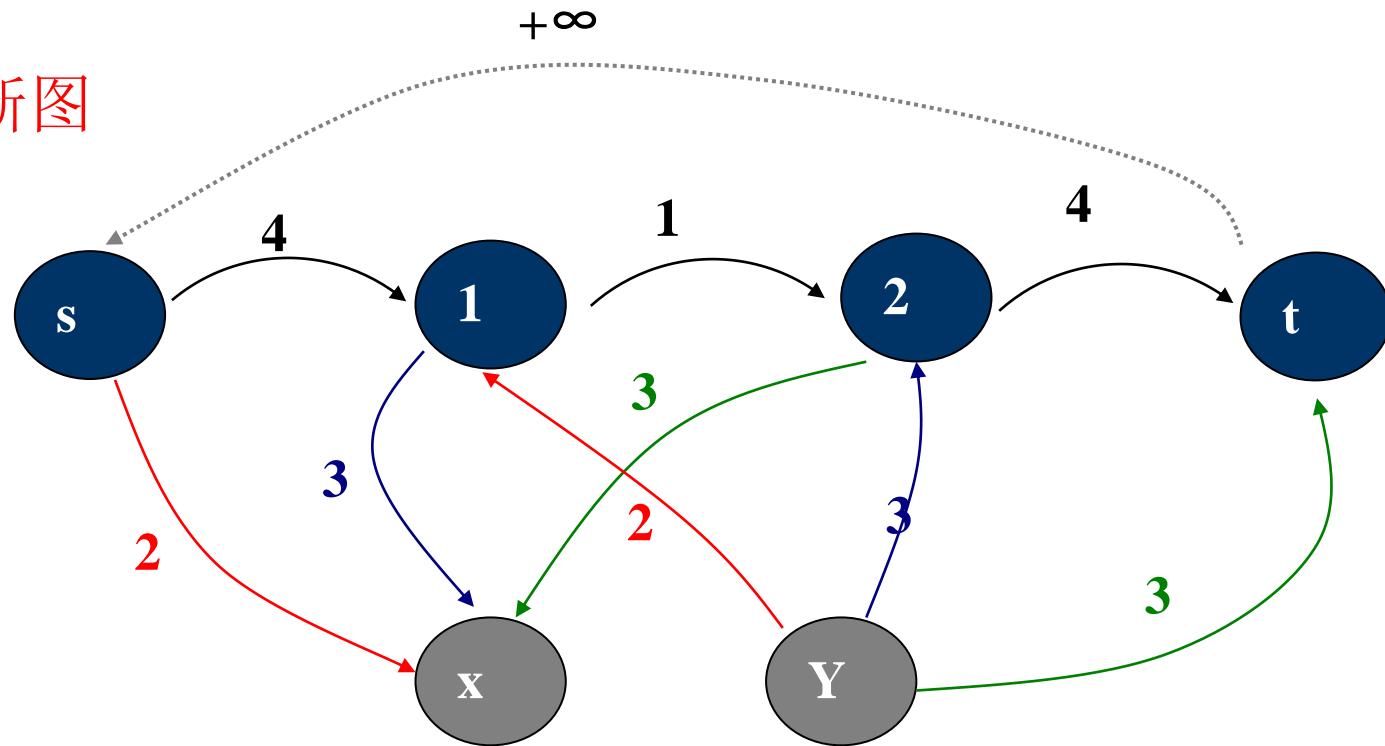


去掉边 (x,y) , 添加由 t 到 s 的容量为正无穷大的边, 使 y 和 x 分别成为新的源和新的汇。



若此图上的最大流能够占满与Y相连的所有边的容量（自然也就会占满所有连到x的边的容量），那么原图上就存在满足上下界条件的可行流。若最大流不能够占满与Y相连的所有边的容量，则原图不存在可行流。

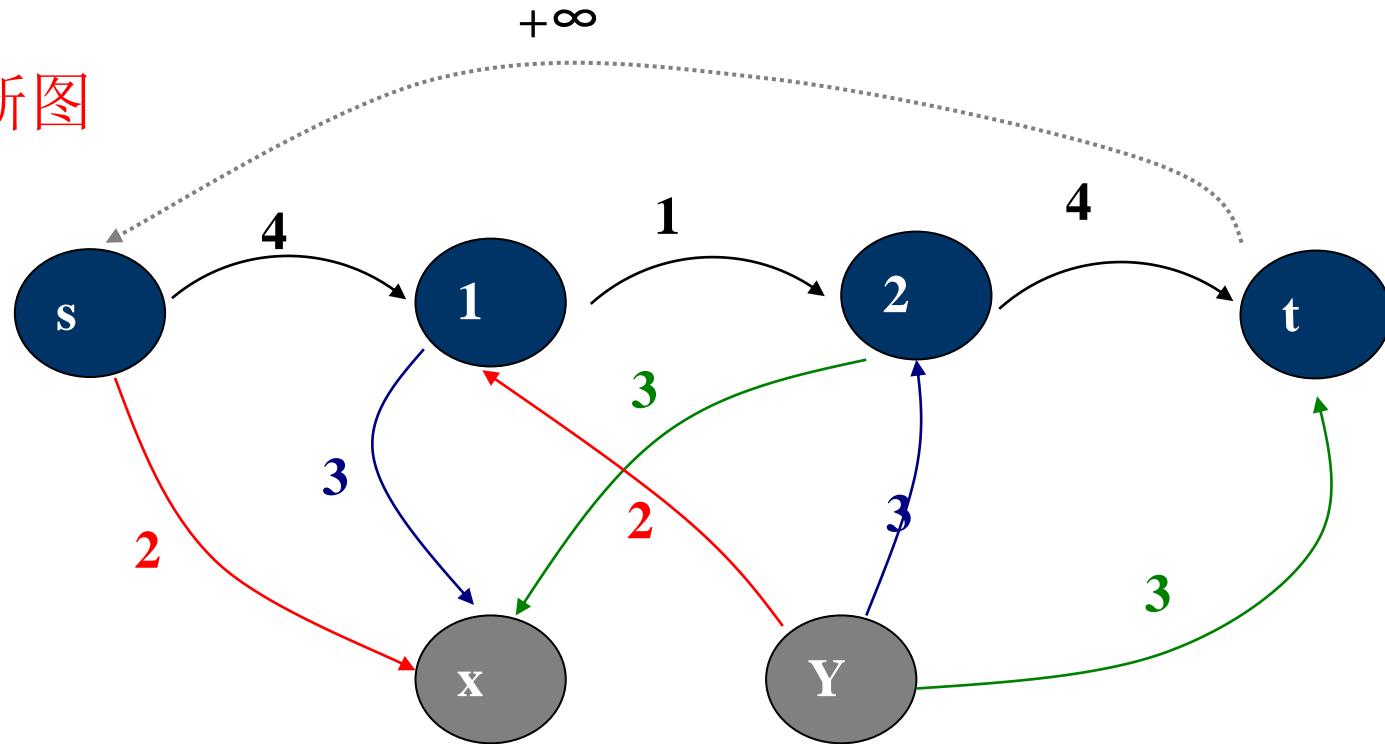
新图



新图最大流若小于新图中x的流入量之和,则原问题无解

在新图的最大流中, 求出s流出的流量之和, 记为
sum1

新图



在做过一遍最大流的**新图**的残余网络中，去掉 $t \rightarrow s$ 以及 $s \rightarrow t$ 的边，然后以 s 为源， t 为汇再做一次最大流，此时得到的流量 $sum2$ ，则 $sum1+sum2$ 就是在原图上满足下界的最大流。

和 x, y 相连的边不用处理，因为 x, y 实际上是只能流入或只能流出的点，在图中不起作用。

要想求出每条边上的流量，怎么办？

要想求出每条边上的流量，怎么办？

在做第二次最大流之前，将图备份到G2

经过两次求最大流后，最后变成的残余网络是G

此时 $G2[i][j] - G[i][j] + LC[i][j]$ 就是 $i \rightarrow j$ 上的流量

$LC[i][j]$ 是 $i \rightarrow j$ 边上的流量下界(下界是被满足的)

处理网络流题目要注意，如果有重边，则要将重边上的容量和下界累加，合并成一条边。

POj 2396 Budget

参考: <http://blog.csdn.net/wsniyufang/article/details/6601162>

现在有一个 $n*m$ 的方阵,方阵里面的数字未知,但是我们知道如下约束条件:

- 1> 每一行的数字的和
- 2> 每一列的数字的和

3> 某些格子里的数, 大小有限制。比如规定第2行第3列的数字必须大于5(或必须小于3,或必须等于10等)

求解是否存在在满足所有的约束的条件下用正数来填充该方阵的方案,若有,输出填充后的方阵,否则输出**IMPOSSIBLE.**

这道题可以转化成容量有上下界的最大流问题,将方阵的行从1.....n编号,列n+1.....n+m编号,添加源点s=0和汇点t=n+m+1.

1>将源点和每一个行节点相连,相连所形成的边的容量和下界置为该行所有数字的和

2>将每一个列节点和汇点相连,相连所形成的边的容量和下界都置为该列所有数字的和

3>从每个行节点到每个列节点连边, 容量为无穷大

4>如果u行v列的数字必须大于w,则边,v+n>流量的下界是w+1

5>如果u行v列的数字必须小于w,则边,v+n>容量改为w-1

6>如果u行v列的数字必须等于w,则边,v+n>流量的下界和容量都是w

找到的可行流（也是最大流）, 就是问题的解

本题trick:

- 1) W 可能为负数，产生流量下界为负数的情况。应处理成
0
- 2) 数据本身可能矛盾。比如前面说了 $(2,1) = 1$, 后面又说
 $(2,1) = 10$

最小费用最大流

以下内容引自<http://web.nuist.edu.cn/courses/dlxxxt/ch5/5.7.3.htm>

<http://jpkc.lzjtu.edu.cn/material3/xxxt/zxfyzdl.htm>

设有一个网络图 $G(V, E)$, , $V=\{s, a, b, c, \dots, s'\}$, E 中的每条边 (i, j) 对应一个容量 $c(i, j)$ 与输送单位流量所需费用 $a(i, j)$ 。如有一个运输方案 (可行流), 流量为 $f(i, j)$, 则最小费用最大流问题就是这样一个求极值问题:

$$\min_{f \in F} a(f) = \min_{f \in F} \sum_{(i, j) \in E} a(i, j) f(i, j)$$

其中 F 为 G 的最大流的集合, 即在最大流中寻找一个费用最小的最大流。

最小费用最大流算法过程

以下内容引自<http://web.nuist.edu.cn/courses/dlxxt/ch5/5.7.3.htm>

<http://jpkc.lzjtu.edu.cn/material3/xxxt/zxfyzdl.htm>

1. 求出从发点到收点的最小费用通路 $\mu(s,t)$
2. 对该通路 $\mu(s,t)$ 分配最大可能的流量: $\bar{f} = \min_{(i,j) \in \mu(s,t)} \{c(i,j)\}$ 并让通路上所有边的容量相应减少 \bar{f} 。这时，对于通路上的饱和边，其单位流费用相应改为 ∞
3. 作该通路 $\mu(s,t)$ 上所有边 (i,j) 的反向边 (j,i) ，令 $c(j,i) = \bar{f}, d(j,i) = -d(i,j)$ 。
4. 在这样构成的新网络中，重复上述步骤 1, 2, 3，直到从发点到收点的全部流量等于 f_v 为止（或者再也找不到从 s 到 t 的最小费用通路）。

反复用spfa算法做源到汇的最短路进行增广，边权值为边上单位费用。反向边上的单位费用是负的。

直到无法增广，即为找到最小费用最大流。

成立原因：每次增广时，每增加1个流量，所增加的费用都是最小的。

因为有负权边（取消流的时候产生的），所以不能用迪杰斯特拉算法求最短路。

- POJ 2135
- 题目描述：
- 有n个景点，一个人要从1号景点走到n号景点，再从n号景点走到1号（回来的路不能重复，不一定走完所有景点，只要求从1到n即可），给你一些景点之间的路的长度（双向），问你最短需要走多少路才能回来？
- 解题报告：
- 最小费用就是路径长度的总和，最大流就是来回的两条路。
- 由于去和回来可以看成：2条从1到n的不同的路。所以转化成求从1到n的两条不同的路。

- 假设a b之间有长度为c的路。按照最小费用流建图：
 - ab之间费用为c， 容量是1。
 - ba之间费用为c， 容量是1.
- 建立一个源点，连接1号景点，无费用，容量为2（表示可以有两条路）
- 同理，建立一个汇点，连接n号景点，无费用，容量为2.
- 这样，如果求的最大流是2，就表示了有两条从1到n的不同的路。（因为中间的点边容量只是1，只能用一次）
- 这样求的最小费用最大流的最小费用就是最短路径长度。