第 6 章　网络流问题

网络流问题是图论中一类常见的问题。许多系统都包含了流量，例如，公路系统中有车辆流，控制系统中有信息流，供水系统中有水流，金融系统中有现金流等。从问题求解的需求出发，网络流问题可以分为：网络最大流，流量有上下界网络的最大流和最小流，最小费用最大流，流量有上下界网络的最小费用最大流等。网络流算法也是求解其他一些图论问题的基础，如求解图的顶点连通度和边连通度、匹配问题等。本章介绍各种网络流问题及求解方法。

6.1　网络最大流

先看一个运输方案设计的例子。图 6.1(a)是连接产品产地 V_s(称为源点)和销售地 V_t(称为汇点)的交通网，每一条弧<u, v>代表从 u 到 v 的运输线，产品经这条弧由 u 输送到 v，弧旁边的数字表示这条运输线的最大通过能力，以后简称**容量(**Capacity)，单位为百吨。产品经过交通网从 V_s 输送到 V_t。现在要求制定一个运输方案，使得从 V_s 运输到 V_t 的产品数量最多。

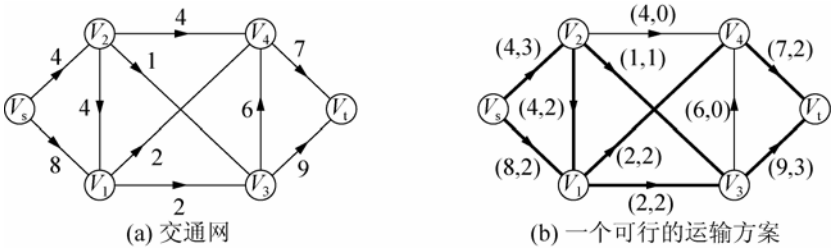


图 6.1　交通网及一个可行的运输方案

图 6.1(b)给出了一个可行的运输方案(粗线所表示的弧为运输方案中的弧)。

(1) 200 吨物资沿着有向路径 P_1(V_s, V_2, V_1, V_4, V_t)运到销售地。

(2) 200 吨物资沿着有向路径 P_2(V_s, V_1, V_3, V_t)运到销售地。

(3) 100 吨物资沿着有向路径 P_3(V_s, V_2, V_3, V_t)运到销售地。

总的运输量为 500 吨。在图 6.1(b)中，每条弧旁边的两个数字，如(4, 3)，分别代表弧的容量和实际运输量。

一个可行的运输方案应满足以下条件。

(1) 实际运输量不能是负的。

(2) 每条弧的实际运输量不能大于该弧的容量。

(3) 除了源点 V_s 和汇点 V_t 外，对其他顶点 u 来说，所有流入 u 的弧上的运输量总和应该等于所有从 u 出发的弧上的运输量总和。

现在的问题如下。

(1) 从 V_s 到 V_t 的运输量是否可以增多？

(2) 从 V_s 到 V_t 的最大运输量是多少？

6.1.1　基本概念

网络最大流、增广路、残留网络、最小割这几个概念是构成最大流最小割定理(定理 6.5)的基本概念，而该定理是网络流理论的基础，本节介绍这几个概念，在 6.1.2 节里介绍最大流最小割定理。

1. 容量网络和网络最大流

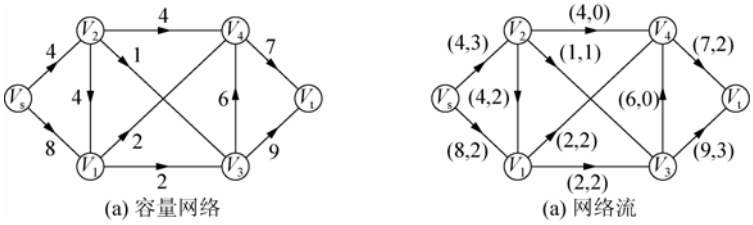
容量网络(Capacity Network)：设 $G(V, E)$是一个有向网络，在 V 中指定了一个顶点，称为源点(记为 V_s)，以及另一个顶点，称为汇点(记为 V_t)；对于每一条弧$<u, v>\in E$，对应有一个权值 $c(u, v)>0$，称为**弧的容量**(Capacity)。通常把这样的有向网络 G 称为容量网络。

例如，图 6.2(a)所示的有向网络就是一个容量网络，每条弧上的数值表示弧的容量。

弧的流量(Flow Rate)：通过容量网络 G 中每条弧$<u, v>$上的实际流量(简称**流量**)，记为 $f(u, v)$。

网络流(Network Flow)：所有弧上流量的集合 $f = \{ f(u, v) \}$，称为该容量网络 G 的一个网络流。

在图 6.2(b)中，每条弧旁边括号内的两个数值($c(u, v), f(u, v)$)，第 1 个数值表示弧容量，第 2 个数值表示通过该弧的流量。例如，弧$<V_s, V_1>$上的两个数字(8, 2)，前者是弧容量，表示通过该弧最大流量为 8，后者表示目前通过该弧的实际流量为 2。



(a) 容量网络　　(a) 网络流

图 6.2　容量网络与网络流

从图 6.2(b)中可见如下几点。

(1) 通过每弧的流量均不超过弧容量。

(2) 源点 V_s 流出的总量为 3 + 2 = 5，等于流入汇点 V_t 的总量 2 + 3 = 5。

(3) 其他中间顶点的流出流量等于其流入流量。例如，中间顶 V_2 的流入流量为 3，流出流量为：2 + 1 = 3。

可行流(Feasible Flow)：在容量网络 $G(V, E)$中，满足以下条件的网络流 f，称为**可行流**。

(1) **弧流量限制条件**：

$$0 \leqslant f(u, v) \leqslant c(u, v),\ <u, v>\in E \tag{6-1}$$

(2) **平衡条件**：

$$\sum_v f(u,v) - \sum_v f(v,u) = \begin{cases} |f| & \text{当} u = V_s \\ 0 & \text{当} u \neq V_s, V_t \\ -|f| & \text{当} u = V_t \end{cases} \tag{6-2}$$

式中：$\sum_{v} f(u,v)$ 表示从顶点 u 流出的流量总和；$\sum_{v} f(v,u)$ 表示流入顶点 u 的流量总和；$|f|$ 为该**可行流的流量**，即源点的净流出流量，或汇点的净流入流量。

对于任何一个容量网络，可行流总是存在的，如 $f=\{0\}$，即每条弧上的流量为 0，该网络流称为**零流**(Zero Flow)。

伪流(Pseudoflow)：如果一个网络流只满足弧流量限制条件(式 6-1)，不满足平衡条件，则这种网络流称为伪流，或称为**容量可行流**。伪流的概念在 6.1.3 和 6.1.7 节中介绍预流推进算法时要用到。

最大流(Maximum Flow)：在容量网络 $G(V, E)$中，满足弧流量限制条件和平衡条件、且具有最大流量的可行流，称为**网络最大流**，简称最大流。

2. 链与增广路

在容量网络 $G(V, E)$中，设有一可行流 $f=\{f(u, v)\}$，根据每条弧上流量的多少以及流量和容量的关系，可将弧分为以下 4 种类型。

(1) **饱和弧**，即 $f(u, v)=c(u, v)$。

(2) **非饱和弧**，即 $f(u, v)<c(u, v)$。

(3) **零流弧**，即 $f(u, v)=0$。

(4) **非零流弧**，即 $f(u, v)>0$。

例如，在图 6.2(b)中，弧$<V_1, V_4>$、$<V_1, V_3>$是饱和弧；弧$<V_s, V_2>$、$<V_2, V_1>$等是非饱和弧；弧$<V_2, V_4>$、$<V_3, V_4>$是零流弧；弧$<V_1, V_4>$、$<V_3, V_t>$等是非零流弧等。

不难看出，饱和弧与非饱和弧，零流弧与非零流弧这两对概念是交错的，饱和弧一般也是非零流弧，零流弧一般也是非饱和弧。

链(Chain)：在容量网络中，称顶点序列$(u, u_1, u_2, \ldots, u_n, v)$为一条链，要求相邻两个顶点之间有一条弧，如$<u, u_1>$或$<u_1, u>$为容量网络中的一条弧。

设 P 是 G 中从 V_s 到 V_t 的一条链，约定从 V_s 指向 V_t 的方向为该链的正方向。注意，链的概念不等同于有向路径的概念，在链中，并不要求所有的弧都与链的正方向同向。

沿着 V_s 到 V_t 的一条链，各弧可分为两类。

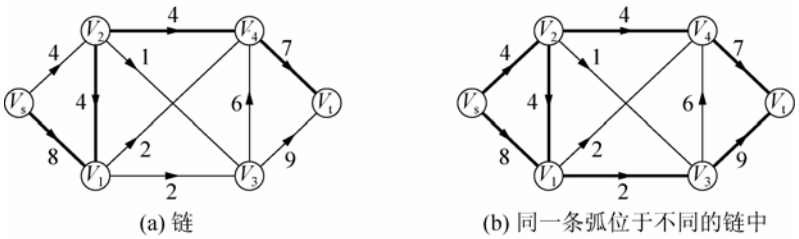
(1) **前向弧**(方向与链的正方向一致的弧)，其集合记为 $P+$。

(2) **后向弧**(方向与链的正方向相反的弧)，其集合记为 $P-$。

注意，前向弧和后向弧是相对的，即相对于指定链的正方向。

例如，在图 6.3(a)中，指定的链为：$P=\{V_s, V_1, V_2, V_4, V_t\}$，这条链在图 6.3(a)中用粗线标明。则 $P+$和 $P-$分别为：

$P+=\{<V_s, V_1>, <V_2, V_4>, <V_4, V_t>\}$，$P-=\{<V_2, V_1>\}$。



(a) 链　　(b) 同一条弧位于不同的链中

图 6.3　前向弧和后向弧

注意，同一条弧可能在某条链中是前向弧，而在另外一条链中是后向弧。例如，如图 6.3(b)所示，弧<V_2, V_1>在链 P_1 = { V_s , V_1 , V_2 , V_4 , V_t }是后向弧，而在链 P_2 = { V_s , V_2 , V_1 , V_3 , V_t }是前向弧。这一点在 6.4 节中求最小费用最大流时要用到。

增广路(Augmenting Path)。设 f 是一个容量网络 G 中的一个可行流，P 是从 V_s 到 V_t 的一条链，若 P 满足下列条件。

(1) 在 P 的所有前向弧<u, v>上，$0 \leqslant f(u, v) < c(u, v)$，即 P+中每一条弧都是非饱和弧。

(2) 在 P 的所有后向弧<u, v>上，$0 < f(u, v) \leqslant c(u, v)$，即 P–中每一条弧是非零流弧。

则称 P 为关于可行流 f 的一条增广路，简称为**增广路**(或称为**增广链**、**可改进路**)。

那么，为什么将具有上述特征的链 P 称为增广路呢？原因是可以通过修正 P 上所有弧的流量 $f(u, v)$来把现有的可行流 f 改进成一个值更大的流 f_1。

沿着增广路改进可行流的操作称为**增广**(augmenting)。

下面具体地给出一种方法，利用这种方法就可以把 f 改进成一个值更大的流 f_1。这种方法是：

(1) 不属于增广路 P 的弧<u, v>上的流量一概不变，即 $f_1(u, v) = f(u, v)$；

(2) 增广路 P 上的所有弧<u, v>上的流量按下述规则变化：(始终满足可行流的 2 个条件)

① 在前向弧<u, v>上，$f_1(u, v) = f(u, v) + \alpha$ ；

② 在后向弧<u, v>上，$f_1(u, v) = f(u, v) - \alpha$ 。

称 α 为**可改进量**，它应该按照下述原则确定：

α 既要取得尽量大；

又要使变化后 f_1 仍满足可行流的两个条件——容量限制条件和平衡条件。

不难看出，按照这个原则，α 既不能超过每条前向弧的 $c(u, v) - f(u, v)$，也不能超过每条后向弧的 $f(u, v)$。因此 α 应该等于每条前向弧上的 $c(u, v) - f(u, v)$与每条后向弧上的 $f(u, v)$的最小值。即：

$$\alpha = \min\{\min_{P+}\{c(u,v) - f(u,v)\}, \min_{P-} f(u,v)\} \tag{6-3}$$

图 6.4(a)给出了一条增广路 P(V_s, V_1, V_2, V_4, V_t)。现在就按照上面讲的方法将流 f 改进成一个更大的流。首先应该确定改进量 α，先看 P 的前向弧集合：

$$P+ = \{ <V_s, V_1>, <V_2, V_4>, <V_4, V_t> \}$$

$$C_{s1} - f_{s1} = 8 - 2 = 6，\ C_{24} - f_{24} = 4 - 0 = 4，\ C_{4t} - f_{4t} = 7 - 3 = 4$$

再看 P 的后向弧集合：$P-$ = { <V_2, V_1> }，在这条弧上 $f_{21} = 2$。

因此 α 最多取 2，这样既可以使改进后的每条前向弧上的流量有所增加，又可以使改进后的每条后向弧上的流量在减少 α 之后不至于变成负数。改进后的流如图 6.4(b)所示，改进后的流，其流量为 7。

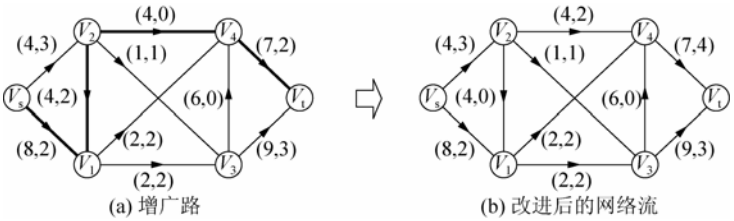


图 6.4　增广路及改进方法

3. 残留容量与残留网络

残留容量(residual capacity)：给定容量网络 $G(V, E)$及可行流 f，弧$<u, v>$上的残留容量记为 $c'(u, v)= c(u, v) - f(u, v)$。每条弧的残留容量表示该弧上可以增加的流量。因为，从顶点 u 到顶点 v 流量的减少，等效于顶点 v 到顶点 u 流量增加，所以每条弧$<u, v>$上还有一个反方向的残留容量 $c'(v, u) = - f(u, v)$。

残留网络(residual network)：设有容量网络 $G(V, E)$及其上的网络流 f，G 关于 f 的残留网络(简称残留网络)记为 $G'(V', E')$，其中 G'的顶点集 V'和 G 的顶点集 V 相同，即 $V'=V$，对于 G 中的任何一条弧$<u, v>$，如果 $f(u, v) < c(u, v)$，那么在 G'中有一条弧$<u, v>\in E'$，其容量为 $c'(u, v) = c(u, v) - f(u, v)$，如果 $f(u, v) > 0$，则在 G'中有一条弧$<v, u>\in E'$，其容量为 $c'(v, u) = f(u, v)$。残留网络也称为**剩余网络**。

从残留网络的定义可以看出，原容量网络中的每条弧在残留网络中都化为一条或两条弧。例如图 6.5(a)所示的容量网络 G，其残留网络 G'为图 6.5(b)。

残留网络中每条弧都表示在原容量网络中能沿其方向增广，弧$<u, v>$的容量 $c'(u, v)$表示原容量网络能沿着 u 到 v 的方向增广大小为 $c'(u, v)$的流量。因此，在残留网络中，从源点到汇点的任意一条简单路径都对应一条增广路，路径上每条弧容量的最小值即为能够一次增广的最大流量。

例如，在图 6.5(b)中，源点到汇点的一条路径为(V_s, V_2, V_4, V_t)，这条路径有 3 条弧，容量分别为 1、4、2，因此沿着这条路径增广可以增加 1 个单位的流量。

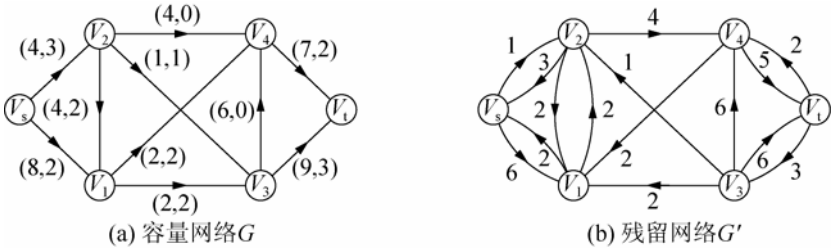


图 6.5　残留网络

残留网络与原网络有如下关系。

定理 6.1(残留网络与原网络的关系)　设 f 是容量网络 $G(V, E)$的可行流，f'是残留网络 G'的可行流，则 $f+f'$仍是容量网络 G 的一个可行流。($f+f'$表示对应弧上的流量相加。)

4. 割与最小割

割(Cut)：在容量网络 $G(V, E)$中，设 $E'\subseteq E$，如果在 G 的基图中删去 E'后不再连通，则称 E'是 G 的割。割将 G 的顶点集 V 划分成两个子集 S 和 $T = V—S$。将割记为(S, T)。

S—t 割：更进一步，如果割所划分的两个顶点子集满足源点 $V_s\in S$，汇点 $V_t\in T$，则称该割为 S—t 割。S—t 割(S, T)中的弧$<u, v>(u\in S, v\in T)$称为割的前向弧，弧$<u, v>(u\in T, v\in S)$称为割的反向弧。(注意，在本章中，如无特别说明，所说的割均指 S—t 割。)

割的容量：设(S, T)为容量网络 $G(V, E)$的一个割，其容量定义为所有前向弧的容量总和，用 $c(S, T)$表示。即：

$$c(S, T)=\sum c(u, v) \quad u\in S，v\in T，<u, v>\in E。 \tag{6-4}$$

例如在图 6.6(a)中，如果选定 $S=\{V_s, V_1, V_2, V_3\}$，则 $T=\{V_4, V_t\}$，(S, T)就是一个 S—t 割。其容量 $c(S, T)$为图中粗线边$<V_2, V_4>$, $<V_1, V_4>$, $<V_3, V_4>$, $<V_3, V_t>$的容量总和，即：

$$c(S, T)=C_{24}+C_{14}+C_{34}+C_{3t}=4+2+6+9=21。$$

最小割(Minimum Cut)：容量网络 $G(V, E)$的最小割是指容量最小的割。

割的净流量：设 f 是容量网络 $G(V, E)$的一个可行流，(S, T)是 G 的一个割，定义割的净流量 $f(S, T)$为：

$$f(S, T)=\sum f(u, v)\quad u\in S，v\in T，<u, v>\in E \text{ 或} <v, u>\in E \tag{6-5}$$

注意：

(1) 在统计割的净流量时：在式(6-5)中，反向弧的流量为负值，即如果$<v, u>\in E$，那么在统计割的净流量时 $f(u, v)$是一个负值。

(2) 在统计割的容量时：在式(6-4)中，不统计反向弧的容量。

例如，在图 6.6(b)中，$S=\{V_s, V_1\}$，则 $T=\{V_2, V_3, V_4, V_t\}$，割(S, T)的容量 $c(S, T)$为：

$$c(S, T)=C_{s2}+C_{14}+C_{13}=4+2+2=8$$

割(S, T)的净流量为：$f(S, T)=f_{s2}+f_{21}+f_{14}+f_{13}=3+(-2)+2+2=5$

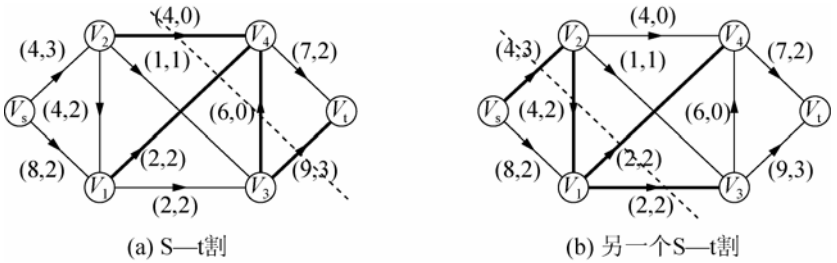


图 6.6　割的容量与净流量

定理 6.2(网络流流量与割的净流量之间的关系)　在一个容量网络 $G(V, E)$中，设其任意一个流为 f，关于 f 的任意一个割为(S, T)，则有 $f(S, T)=|f|$，即网络流的流量等于任何割的净流量。

例如，在图 6.6(b)中，$f(S, T)=5$，$|f|=5$，两者相等。

定理 6.3(网络流流量与割的容量之间的关系)　在一个容量网络 $G(V, E)$中，设其任意一个流为 f，任意一个割为(S, T)，则必有 $f(S, T)\leqslant c(S, T)$，即网络流的流量小于或等于任何割的容量。

根据下面的定理 6.5 可知，定理 6.3 中的关系式当且仅当 f 为最大流，(S, T)为最小割时取等号。例如，在图 6.6(b)中，$c(S, T)=8$，该图所示的割实际上是一个最小割，在后面的讨论中可以看到，该容量网络的最大流为 8。

6.1.2　最大流最小割定理

如何判定一个网络流是否是最大流？有以下两个定理。

定理 6.4(增广路定理)　设容量网络 $G(V, E)$的一个可行流为 f，f 为最大流的充要条件是在容量网络中不存在增广路。

定理 6.5(最大流最小割定理)　对容量网络 $G(V, E)$，其最大流的流量等于最小割的容量。

根据定理 6.4 和 6.5，可以总结出以下 4 个命题是等价的(设容量网络 $G(V, E)$的一个可行流为 f)。

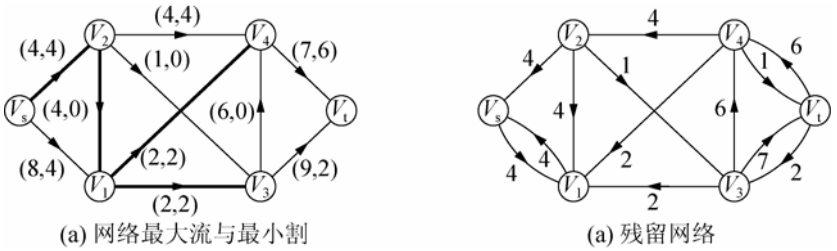
(1) f 是容量网络 G 的最大流。

(2) $|f|$等于容量网络最小割的容量。

(3) 容量网络中不存在增广路。

(4) 残留网络 G'中不存在从源点到汇点的路径。

例如，图 6.7(a)所示的网络流是容量网络中的最大流，其流量为 8。粗线所表示的弧组成了一个最小割，其容量也为 8。在图 6.7(a)中，不存在增广路，而在图 6.7(b)所示的残留网络中，也不存在从源点到汇点的路径。



(a) 网络最大流与最小割　　(a) 残留网络

图 6.7　网络最大流、最小割、残留网络

6.1.3　网络最大流的求解

给定一个容量网络 $G(V, E)$，如何求其最大流是 6.1 节的重点。网络最大流的求解主要有两大类算法：**增广路算法(**Augmenting Path Algorithm**)**和**预流推进算法(**Preflow-Push Algorithm**)**。

1. 增广路算法

根据增广路定理，为了得到最大流，可以从任何一个可行流开始，沿着增广路对网络流进行增广，直到网络中不存在增广路为止，这样的算法称为**增广路算法**。问题的关键在于如何有效地找到增广路，并保证算法在有限次增广后一定终止。

增广路算法的基本流程如下(如图 6.8 所示)。

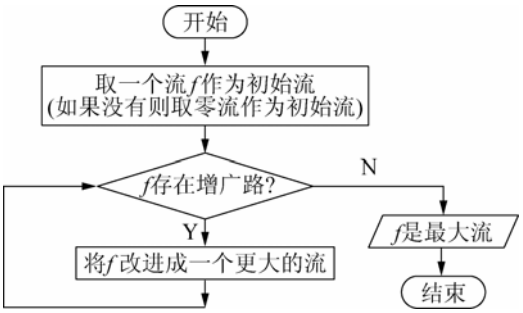


图 6.8　求网络最大流的思路

(1) 取一个可行流 f 作为初始流(如果没有给定初始流，则取零流 $f=\{0\}$作为初始流)。

(2) 寻找关于 f 的增广路 P，如果找到，则沿着这条增广路 P 将 f 改进成一个更大的流。

(3) 重复第(2)步直到 f 不存在增广路为止。

增广路算法的关键是寻找增广路和改进网络流。

将在 6.1.4 节中讨论 Ford 和 Fulkerson 于 1956 年提出的标号算法，该算法的思想是通过一个标号过程来寻找容量网络中的增广路。6.1.5 节讨论最短增广路算法，6.1.6 节讨论连续最短增广路算法(Dinic 算法)，这是两种效率更高的增广路算法。

2. 预流推进算法

预流推进算法是从一个预流出发对活跃顶点沿着允许弧进行流量增广，每次增广称为一次**推进**(Push)。在推进过程中，流一定满足流量限制条件(式 6-1)，但一般不满足流量平衡条件(式 6-2)，因此只是一个伪流。此外，如果一个伪流中，从每个顶点(除源点 V_s、汇点 V_t 外)流出的流量之和总是小于等于流入该顶点的流量之和，称这样的伪流为**预流**(Preflow)。因此这类算法被称为预流推进算法。

将在 6.1.7 节讨论一般预流推进算法，在 6.1.8 节讨论最高标号预流推进算法。

在 6.1.9 节对这两大类算法作对比分析，最后在 6.1.10 节通过几道 ACM/ICPC 例题来讲解这些算法的程序实现。

6.1.4　一般增广路方法——Ford-Fulkerson 算法

在 Ford-Fulkerson 算法中，寻找增广路和改进网络流的方法为**标号法**(Label Method)，接下来先看标号法的两个实例，再介绍标号法的运算过程和程序实现。

1. 标号法实例

以下两个实例分别从初始流为零流和非零流出发采用标号法求网络最大流。

1) 标号法求最大流的实例 1——初始流为零流

如图 6.9(a)所示，各条弧上的流量均为 0，初始可行流 f 为零流。

在图 6.9(b)中，对初始流 f 进行第 1 次标号。每个顶点的**标号**包含以下两个分量。

(1) 第 1 个分量指明它的标号从哪个顶点得到，以便找出可改进量。

(2) 第 2 个分量是为确定可改进量 α 用的。

首先对源点 V_s 进行标号，标号为(0, +∞)。每次标号，源点的标号总是(0, +∞)。其中第 1 个分量为 0，表示该顶点是源点；第 2 个分量为+∞，表示 V_s 可以流出任意多的流量(只要从它发出的弧可以接受)。

源点有标号以后，采用**广度优先搜索(BFS)的思路**从源点出发进行遍历，并对遍历到的每个顶点进行标号。假设在对某个顶点的多个未标号邻接顶点中进行标号时，按顶点序号从小到大的顺序进行标号。例如源点 V_s 有两个邻接顶点：V_1 和 V_2，则先对顶点 V_1 进行标号。对顶点 V_1 的标号为(V_s, 8)。该标号的含义是：第 2 个分量为 8，表示 V_s 可以流出+∞的流量，但弧<V_s, V_1>的容量为 8，所以，顶点 V_s 只能接受 8；第 1 个分量表示流量改进量“8”来自顶点 V_s。按照同样的思路对顶点 V_2 进行标号，标号为(V_s, 4)。

源点 V_s 的邻接顶点检查完毕后，再从顶点 V_1 出发对它的邻接顶点进行标号：对顶点 V_3 的标号为(V_1, 2)，对顶点 V_4 的标号为(V_1, 2)。注意，顶点 V_2 也是 V_1 的“邻接”(通过后向弧“邻接”)顶点，但 V_2 已经有标号了，所以不能通过 V_1 对 V_2 进行标号。

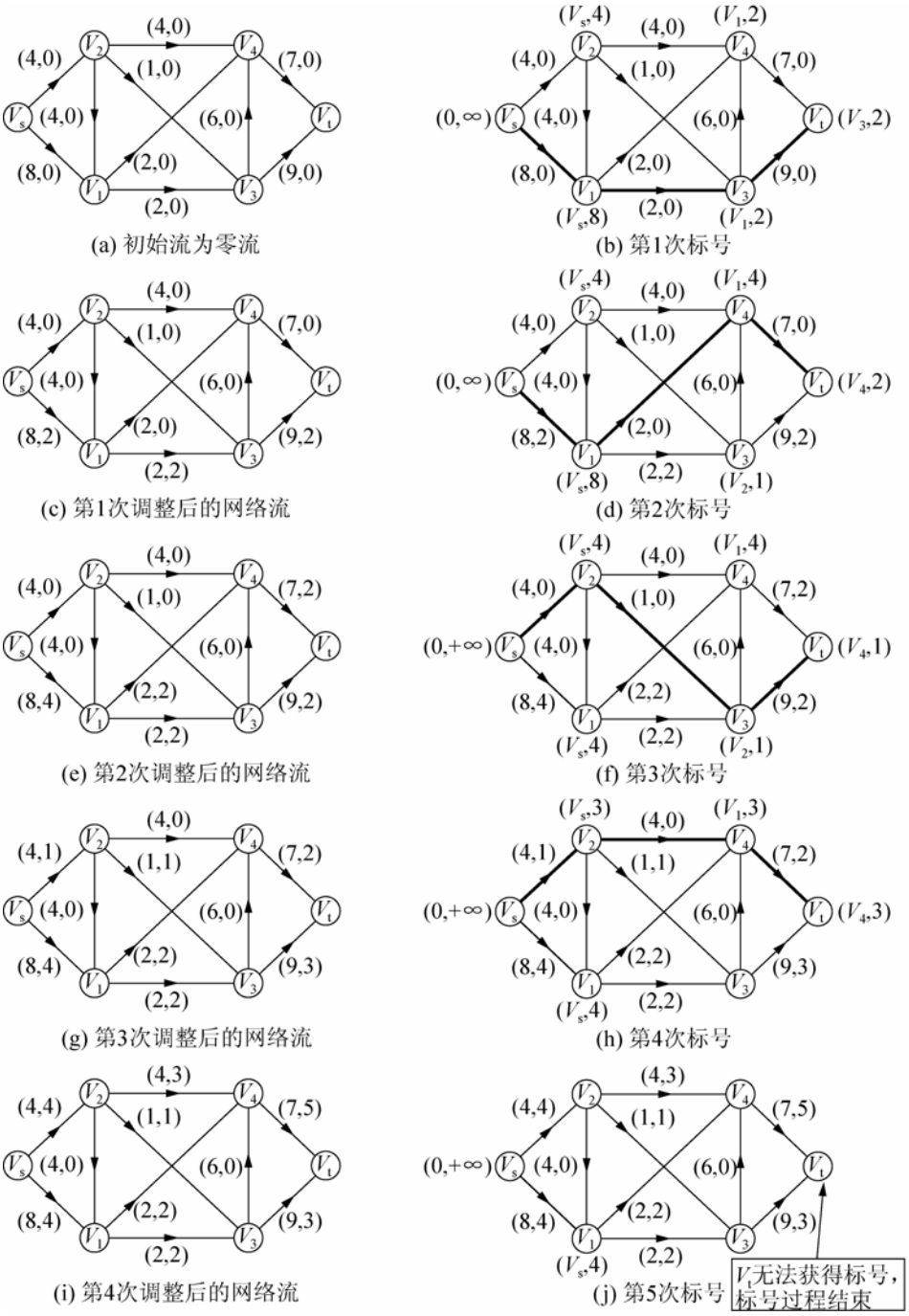


图 6.9　标号法求网络最大流的实例 1——初始流为零流

源点 V_1 的邻接顶点检查完毕后，再从顶点 V_2 出发对它的邻接顶点进行标号，此时顶点 V_2 的邻接顶点中，都已经有标号了。

源点 V_2 的邻接顶点检查完毕后，再从顶点 V_3 出发对它的邻接顶点进行标号，从而通过 V_3 对汇点 V_t 进行标号，标号为$(V_3, 2)$。

一旦汇点 V_t 有标号，且第 2 个分量不为 0，则表示找到一条增广路。确定这条增广路的方法是：从汇点 V_t 标号的第 1 个分量出发，采用“**倒向追踪**”的方法，一直找到源点 V_s。

例如在图 6.9(b)中，汇点 V_t 标号的第 1 个分量为 V_3，表示增广路上汇点 V_t 前面的顶点为 V_3；顶点 V_3 标号的第 1 个分量为 V_1，表示增广路上顶点 V_3 前面的顶点为 V_1；顶点 V_1 标号的第 1 个分量为 V_s，表示增广路上顶点 V_1 前面的顶点为 V_s；因此找到的这条增广路为：$P(V_s, V_1, V_3, V_t)$，增广路中的弧用粗线标明。并且这条增广路的可改进量 α 就是汇点 V_t 标号的第 2 个分量，为 2。沿着这条增广路，可以将流量增加 2，流量变成 2，改进后的流如图 6.9(c)所示。

图 6.9(d)对第 1 次调整后的网络流进行第 2 次标号，求得的增广路为：$P(V_s, V_1, V_4, V_t)$，可改进量 $\alpha = 2$；调整后得到的网络流如图 6.9(e)所示，流量为 4。

图 6.9(f)对第 2 次调整后的网络流进行第 3 次标号，求得的增广路为：$P(V_s, V_2, V_3, V_t)$，可改进量 $\alpha = 1$；调整后得到的网络流如图 6.9(g)所示，流量为 5。

图 6.9(h)对第 3 次调整后的网络流进行第 4 次标号，求得的增广路为：$P(V_s, V_2, V_4, V_t)$，可改进量 $\alpha = 3$；调整后得到的网络流如图 6.9(i)所示，流量为 8。

在图 6.9(j)中，对第 4 次调整后得到的网络流进行第 5 次标号：通过源点 V_s 对顶点 V_1 的标号为$(V_s, 4)$，源点 V_s 无法对顶点 V_2 进行标号，因为弧$<V_s, V_2>$已经饱和了；而顶点 V_1 也无法对它的邻接顶点进行标号。此后汇点 V_t 无法获得标号，或者说汇点 V_t 的可改进量 α 为 0。至此，标号法结束，求得的最大流流量为 8。

2) 标号法求最大流的实例 2——初始流为非零流

如图 6.10(a)所示，初始可行流 f 为非零流，其流量为 5。

图 6.10(b)对初始流进行第 1 次标号，求得的增广路为：$P(V_s, V_2, V_4, V_t)$，可改进量 $\alpha = 1$；调整后得到的网络流如图 6.10(c)所示，流量为 6。

图 6.10(d)所示的第 2 次标号过程要特别注意：顶点 V_2 获得的标号中，第 1 个分量为$-V_1$。第 2 次标号过程为：通过源点 V_s 对顶点 V_1 的标号为$(V_s, 6)$，源点 V_s 无法对顶点 V_2 进行标号。然后，顶点 V_1 的邻接顶点中，V_3 和 V_4 都无法从 V_1 获得标号，因为对应的弧已经饱和了；但这时要注意，顶点 V_1 通过后向弧$<V_2, V_1>$与顶点 V_2“邻接”，且顶点 V_2 还没有标号。所以给 V_2 顶点标号为$(-V_1, 2)$，第 1 个分量前的负号表示在找到的增广路中，弧$<V_2, V_1>$是后向弧，在改进当前可行流时它的流量应该减少，相当于这个改进量实际上是由顶点 V_2 提供给顶点 V_1 的。第 2 次标号后，求得的增广路为：$P(V_s, V_1, V_2, V_4, V_t)$，可改进量 $\alpha = 2$；调整后得到的网络流如图 6.10(e)所示，流量为 8。

在图 6.10(f)中，对得到的网络流进行第 3 次标号：通过源点 V_s 对顶点 V_1 的标号为$(V_s, 4)$，源点 V_s 无法对顶点 V_2 进行标号，因为弧$<V_s, V_2>$已经饱和了；而顶点 V_1 也无法对它的邻接顶点进行标号。此后汇点 V_t 无法获得标号，或者说汇点 V_t 的可改进量 α 为 0。至此，标号法结束，求得的最大流流量为 8。

注意：以上两个实例中，容量网络是相同的，只是初始流不同；求解的结果表明，得到的网络最大流的流量是相同的，都为 8。

2. 标号法的运算过程

标号法的具体运算过程如下：从一个可行流 f 出发(若网络中没有给定每条弧的流量，则可以设 f 为零流)，进入标号过程和调整过程。

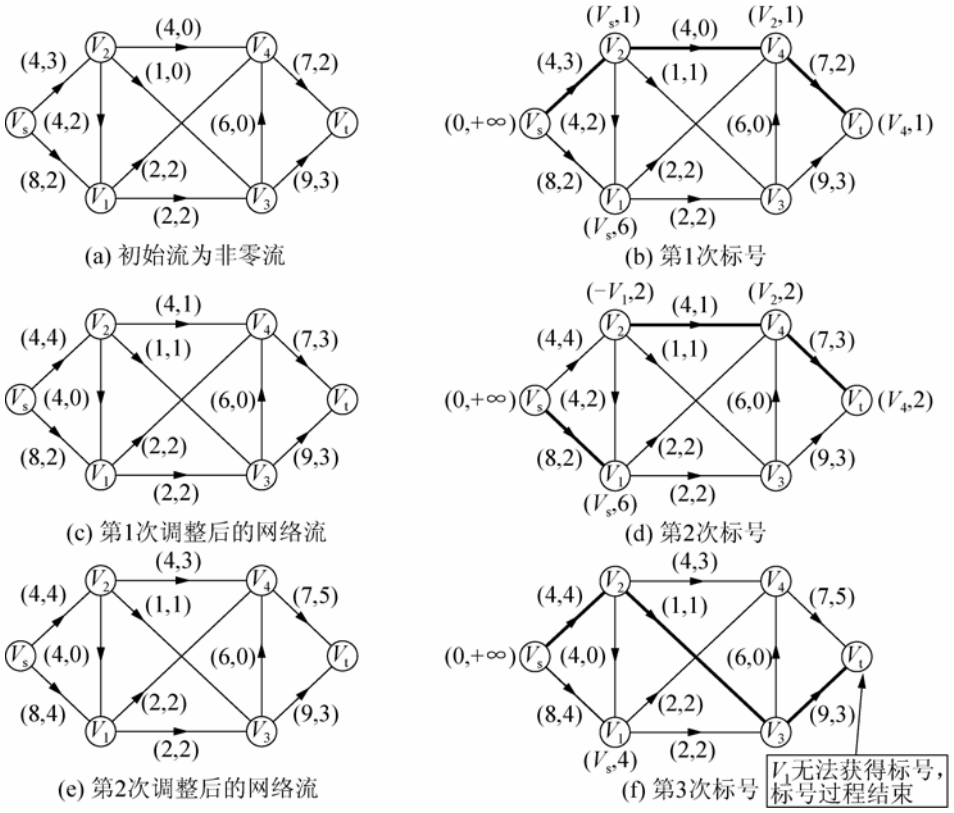


图 6.10　标号法求网络最大流的实例 2——初始流为非零流

1) 标号过程

在标号过程，容量网络中的顶点可以分为如下 3 类。

(1) 未标号顶点。

(2) 已标号，未检查邻接顶点。

(3) 已标号，且已检查邻接顶点(即已经检查它的所有邻接顶点，看是否能标号)。

每个标号顶点的标号包含两个分量。

(1) 第 1 个分量指明它的标号从哪个顶点得到，以便找出增广路。

(2) 第 2 个分量是为确定可改进量 α 用的。

标号过程开始时，总是先给 V_s 标上$(0, +\infty)$，0 表示 V_s 是汇点，$+\infty$表示 V_s 可以流出任意多的流量(只要从它发出的弧可以接受)。这时 V_s 是已标号而未检查的顶点，其余都是未标号点。然后从源点 V_s 出发，对它的每个邻接顶点进行标号。

一般，取一个已标号而未检查的顶点 u，对一切未标号顶点 v，进行如下步骤。

(1) 若 v 与 u“正向”邻接，且在弧$<u, v>$上 $f(u, v)<c(u, v)$，则给 v 标号$(u, L(v))$，这里 $L(v) = \min\{ L(u), c(u, v) - f(u, v) \}$，$L(u)$是顶点 u 能提供的标号，$c(u, v) - f(u, v)$是弧$<u, v>$能接受的标号，取二者中的较小者。这时顶点 v 成为已标号而未检查的顶点。

(2) 若 v 与 u“反向”邻接，且在弧$<v, u>$上 $f(v, u)>0$，则给 v 标号$(-u, L(v))$，这里 $L(v)= \min\{ L(u), f(v, u) \}$。这时顶点 v 成为已标号而未检查的顶点。

当 u 的全部邻接顶点都已检查后，u 成为已标号且已检查过的顶点。

重复上述步骤直至汇点获得标号，一旦汇点 V_t 被标号并且汇点标号的第 2 个分量大于 0，则表明得到一条从 V_s 到 V_t 的增广路 P，转入调整过程；若所有已标号未检查的顶点都检查完毕但标号过程无法继续，从而汇点 V_t 无法获得标号，或者得到的可改进量 $\alpha = 0$，则算法结束，这时的可行流即为最大流。

2) 调整过程

采用“倒向追踪”的方法，从 V_t 开始，利用标号顶点的第 1 个分量逐条弧地找出增广路 P，并以 V_t 的第 2 个分量 $L(V_t)$作为改进量 α，改进 P 路上的流量。

例如，设 V_t 标号的第 1 个分量为 V_k，则弧<V_k, V_t>是增广路 P 上的弧。接下来检查 V_k 标号的第 1 个分量，若为 V_i(或−V_i)，则找到弧<V_i, V_k>(或相应的<V_k, V_i>)。再检查 V_i 标号的第 1 个分量，…，一直到 V_s 为止。这时被找出的弧就构成了增广路 P。改进量 α 是 $L(V_t)$，即 V_t 标号的第 2 个分量。对这条增广路上各条弧的流量作如下调整：

$$f(u,v)=\begin{cases} f(u,v)+\alpha & <u,v>\in P^{+} \\ f(u,v)-\alpha & <u,v>\in P^{-} \\ f(u,v) & <u,v>\notin P \end{cases} \tag{6-6}$$

去掉所有的标号，对新的可行流进行重新标号过程和调整过程。

3. 标号法的程序实现

例 6.1　利用前面介绍的标号法求图 6.9(a)及图 6.10(a)所示的容量网络的最大流，输出各条弧及其流量，以及求得的最大流流量。

假设数据输入时采用如下的格式进行输入：首先输入顶点个数 n 和弧数 m，然后输入每条弧的数据。规定源点为第 0 个顶点，汇点为第 n−1 个顶点。每条弧的数据格式为：u v c f，分别表示这条弧的起点、终点、容量和流量。顶点序号从 0 开始计起。

分析：

在下面的程序中，以邻接矩阵存储容量网络，但邻接矩阵中的元素为结构体 ArcType 类型变量。该结构体描述了网络中弧的结构，包含容量 c 和流量 f 两个成员。

在程序中，还定义了 3 个数组：flag[n], prev[n], alpha[n]，其中：

(1) flag[n]表示顶点状态，其元素取值及含义为：−1——未标号，0——已标号未检查，1——已标号已检查。

(2) prev[n]为标号的第 1 个分量：指明标号从哪个顶点得到，以便找出可改进量。

(3) alpha[n]为标号的第 2 个分量：用以确定增广路的可改进量 α 。

另外，如前所述，从一个已标号未检查的顶点出发，对它的邻接顶点进行标号时，采用的是广度优先搜索的策略，因此，在程序中，定义了一个数组 queue[n]来模拟队列；并定义了两个相关变量 q_s 和 q_e，分别表示队列头位置和队列尾位置，约定从队列头取出结点，从队列尾插入结点；当 q_s<q_e 时表示队列非空。

每一次标号过程如下。

(1) 先将 flag、prev 和 alpha 这 3 个数组各元素都初始化−1。

(2) 将源点初始化为已标号未检查顶点，即 flag[0] = 0, prev[0] = 0, alpha[0] = INF，INF 表示无穷大；并将源点入队列。

(3) 当队列非空并且汇点没有标号，从队列头取出队列头顶点，设这个顶点为 v，v 肯

定是已标号未检查顶点；因此，检查顶点 v 的正向和反向“邻接”顶点，如果没有标号并当前可以进行标号，则对这些顶点进行标号并入队列，这些顶点都是已标号未检查顶点；此后顶点 v 为已标号已检查顶点。反复执行这一步直至队列为空或汇点已获得标号。

图 6.11 描述了图 6.10(b)所示的从非零流出发进行第 1 次标号过程，在图 6.11(e)中，检查完顶点 4 后，汇点(即顶点 5)已经有标号了，可以进行调整了，因此这一次标号过程就结束了。

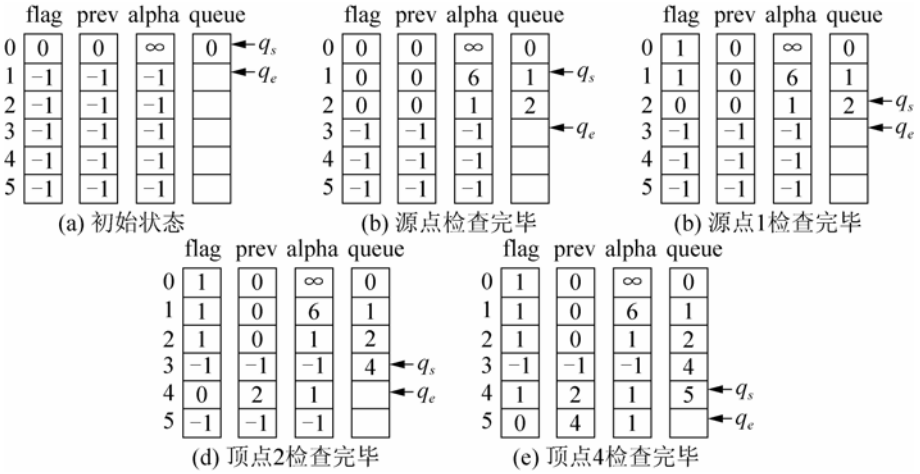


图 6.11　标号法的实现过程

标号完毕后，要进行调整，调整方法是：从汇点出发，通过标号的第 1 个分量，即 prev[5]，采用“倒向追踪”方法，一直找到源点为止，这个过程途经的顶点和弧就构成了增广路。可改进量为汇点标号的第 2 个分量，即 alpha[5]。

代码如下：

```
#define MAXN 1000       //顶点个数最大值
#define INF 1000000     //无穷大
#define MIN(a,b) ((a)<(b)?(a):(b))
struct ArcType          //弧结构
{
    int c, f;           //容量，流量
};
ArcType Edge[MAXN][MAXN]; //邻接矩阵(每个元素为 ArcType 类型)
int n, m;           //顶点个数和弧数
int flag[MAXN];     //顶点状态：-1——未标号,0——已标号未检查,1——已标号已检查
int prev[MAXN];     //标号的第 1 个分量：指明标号从哪个顶点得到，以便找出可改进量
int alpha[MAXN];    //标号的第 2 个分量：可改进量 α
int queue[MAXN];    //相当于 BFS 算法中的队列
int v;              //从队列里取出来的队列头元素
int qs, qe;         //队列头位置，队列尾位置
int i, j;           //循环变量
void ford( )
{
    while( 1 )  //标号直至不存在可改进路
    {
```

```
        //标号前对顶点状态数组初始化
        memset( flag, 0xff, sizeof(flag) );    //将3个数组各元素初始化为-1
        memset( prev,0xff,sizeof(prev));memset(alpha,0xff,sizeof(alpha) );
        flag[0]=0;  prev[0]=0;  alpha[0]=INF;   //源点为已标号未检查顶点
        qs=qe=0;
        queue[qe]=0;  qe++; //源点(顶点0)入队列
        //qs<qe表示队列非空, flag[n-1]==-1表示汇点未标号
        while( qs<qe && flag[n-1]==-1 )
        {
            v=queue[qs];  qs++; //取出队列头顶点
            for( i=0; i<n; i++ )    //检查顶点v的正向和反向"邻接"顶点
            {
                if( flag[i]==-1 )  //顶点i未标号
                {
                    //"正向"且未"满"
                    if( Edge[v][i].c<INF && Edge[v][i].f<Edge[v][i].c )
                    {
                        flag[i]=0; prev[i]=v;   //给顶点i标号(已标号未检查)
                        alpha[i]=MIN(alpha[v],Edge[v][i].c-Edge[v][i].f );
                        queue[qe]=i;  qe++;     //顶点i入队列
}
                    //"反向"且有流量
                    else if( Edge[i][v].c<INF && Edge[i][v].f>0 )
                    {
                        flag[i]=0;prev[i]=-v;   //给顶点i标号(已标号未检查)
                        alpha[i]=MIN( alpha[v], Edge[i][v].f );
                        queue[qe]=i;  qe++;     //顶点i入队列
                    }
                }
            }
            flag[v]=1;  //顶点v已标号已检查
        }//end of while( qs<qe && flag[n-1]==-1 )
        //当汇点没有获得标号，或者汇点的调整量为0，应该退出while循环
        if( flag[n-1]==-1 || alpha[n-1]==0 )  break;
        //当汇点有标号时,应该进行调整了
        int k1=n-1, k2=abs( prev[k1] );
        int a=alpha[n-1];   //可改进量
        while( 1 )
        {
            if( Edge[k2][k1].f<INF )                    //正向
                Edge[k2][k1].f=Edge[k2][k1].f+a;
            else  Edge[k1][k2].f=Edge[k1][k2].f-a;      //反向
            if( k2==0 )  break; //调整一直到源点v0
            k1=k2;  k2=abs( prev[k2] );
        }//end of while( 1 )
    }//end of while( 1 )
    //输出各条弧及其流量，以及求得的最大流量
    int maxFlow=0;
    for( i=0; i<n; i++ )
    {
        for( j=0; j<n; j++ )
```

```
        {
            if( i==0 && Edge[i][j].f<INF ) //求源点流出量，即最大流
                maxFlow+=Edge[i][j].f;
            if(Edge[i][j].f<INF) printf("%d->%d:%d\n",i,j,Edge[i][j].f);
        }
    }
    printf( "maxFlow:%d\n", maxFlow );
}
void main( )
{
    int u, v, c, f;             //弧的起点、终点、容量、流量
    scanf( "%d%d", &n, &m );    //读入顶点个数 n 和弧数 m
    for( i=0; i<n; i++ )        //初始化邻接矩阵中各元素
    {
//INF 表示没有直接边连接
        for( j=0; j<n; j++ )  Edge[i][j].c=Edge[i][j].f=INF;
    }
    for( i=0; i<m; i++ )    //读入每条弧
    {
        scanf( "%d%d%d%d", &u, &v, &c, &f );   //读入边的起点和终点
        Edge[u][v].c=c;  Edge[u][v].f=f;        //构造邻接矩阵
    }
    ford( );    //标号法求网络最大流
}
```

该程序的运行示例如下。

输入：	输出：
6 10	0->1:4
0 1 8 2	0->2:4
0 2 4 3	1->3:2
1 3 2 2	1->4:2
1 4 2 2	2->1:0
2 1 4 2	2->3:1
2 3 1 1	2->4:3
2 4 4 0	3->4:0
3 4 6 0	3->5:3
3 5 9 3	4->5:5
4 5 7 2	maxFlow:8

4. Ford-Fulkerson 算法的复杂度分析

很明显，如果容量网络中各弧的容量和初始流量均为正整数，则 Ford-Fulkerson 算法每增广一次，流量至少会增加 1 个单位，因此 Ford-Fulkerson 算法肯定能在有限的步骤内使得网络流达到最大。类似的理由可以说明：当所有弧上的容量为有理数时，也可在有限的步骤内使得网络流达到最大。但是如果弧上的容量可以是无理数，则 Ford-Fulkerson 算法不一定在有限步内终止。

Ford-Fulkerson 算法并没有明确应该按照怎样的顺序来给顶点进行标号(在前面的实例

分析中，为描述方便，以广度优先搜索的顺序给各顶点标号)，因此可能会出现如图 6.12 所示的最坏情形。在图 6.12 中，交替地使用 $V_sV_1V_2V_t$ 和 $V_sV_2V_1V_t$ 作为增广路，很明显每次只能使流量增加 1，这样就需要 2×2^{100} 次增广才能最终求得最大流。

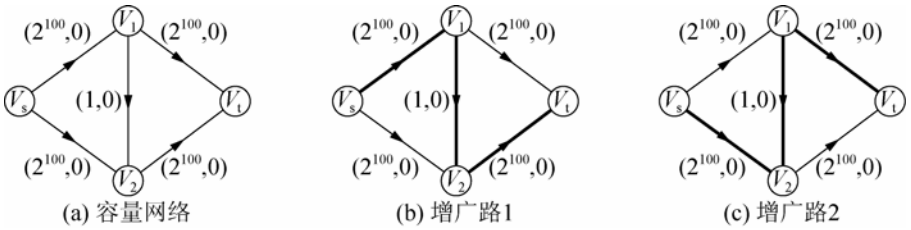


图 6.12　Ford-Fulkerson 算法的最坏情形

由于割($\{V_s\}, V-\{V_s\}$)中前向弧的条数最多为 n 条，因此最大流流量$|F|$的上界为 nU(U 表示网络中各条弧的最大容量)。此外，由于每次增广最多需要对所有弧检查一遍，所以 Ford-Fulkerson 算法的时间复杂度为 $O(mnU)$。因此，Ford-Fulkerson 算法的时间复杂度不仅依赖于容量网络的规模(顶点数和弧数)，还和各条弧的容量有关。(n 和 m 分别为顶点数和边数。)

6.1.5　最短增广路算法

1. 顶点的层次与层次网络

顶点的**层次**(Level)：在残留网络中，把从源点到顶点 u 的最短路径长度(该长度仅仅是值路径上边的数目，与容量无关)，称为顶点 u 的层次，记为 level(u)。源点 V_s 的层次为 0。

将残留网络中所有顶点的层次标注出来的过程称为**分层**。

例如，对图 6.5(b)所示的残留网络进行分层后，得到图 6.13(a)，顶点旁的数值表示顶点的层次。为了让读者更清晰地观察顶点的层次，在图 6.13(b)中还特意将顶点按层次递增的顺序排列。

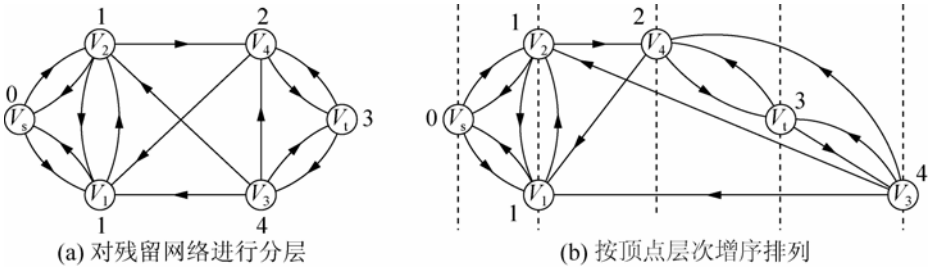


图 6.13　顶点的层次

注意：

(1) 对残留网络进行分层后，弧有 3 种可能的情况。

① 从第 i 层顶点指向第 i+1 层顶点。

② 从第 i 层顶点指向第 i 层顶点。

③ 从第 i 层顶点指向第 j 层顶点($j<i$)。

(2) 不存在从第 i 层顶点指向第 $i+k$ 层顶点的弧($k \geqslant 2$)。

(3) 并非所有网络都能分层。例如图 6.14 所示的网络就不能分层。

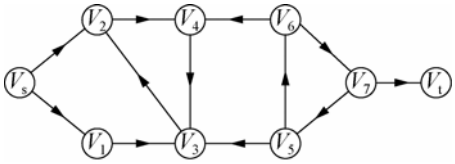


图 6.14　不能分层的网络

层次网络(Level Network)：对残留网络分层后，删去比汇点 V_t 层次更高的顶点和与汇点 V_t 同层的顶点(保留 V_t)，并删去与这些顶点关联的弧，再删去从某层顶点指向同层顶点和低层顶点的弧，所剩的各条弧的容量与残留网络中的容量相同，这样得到的网络是残留网络的子网络，称为层次网络，记为 $G''(V'', E'')$。

根据层次网络的定义可知，层次网络中任意一条弧$<u, v>$，都满足 level(u)+1 = level(v)。这种弧也称为**允许弧**(Admissible Arc)。

例如，对图 6.5(b)所示的残留网络分层并构造层次网络，得到图 6.15 所示的层次网络。

直观地说，层次网络是建立在残留网络基础之上的一张“最短路径图”。从源点开始，在层次网络中沿着边不管怎么走，到达一个终点后，经过的路径一定是终点在残留网络中的最短路径。

阻塞流(Blocking Flow)：设容量网络中的一个可行流为 f，当该网络的层次网络 G''中不存在增广路(即从源点 V_s 到汇点 V_t 的路径)时，称该可行流 f 为层次网络 G''的阻塞流。

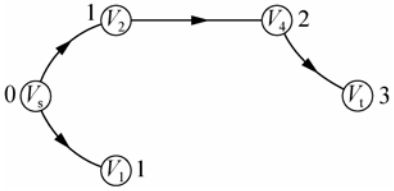


图 6.15　层次网络

2. 最短增广路算法思想

最短增广路算法的思路是：每次在层次网络中找一条含弧数最少的增广路进行增广。最短增广路算法的具体步骤如下。

(1) 初始化容量网络和网络流。

(2) 构造残留网络和层次网络，若汇点不在层次网络中，则算法结束。

(3) 在层次网络中不断用 BFS 增广，直到层次网络中没有增广路为止；每次增广完毕，在层次网络中要去掉因改进流量而导致饱和的弧。

(4) 转步骤(2)。

在最短增广路算法中，第(2)、(3)步被循环执行，将执行(2)、(3)步的一次循环称为一个阶段。每个阶段中，首先根据残留网络建立层次网络，然后不断用 BFS 在层次网络内增广，直到出现阻塞流。注意每次增广后，在层次网络中要去掉因改进流量而导致饱和的弧。该阶段的增广完毕后，进入下一个阶段。这样不断重复，直到汇点不在层次网络内出现为

止。汇点不在层次网络内意味着在残留网络中不存在一条从源点到汇点的路径，即没有增广路。

在程序实现的时候，并不需要真正“构造”层次网络，只需对每个顶点标记层次，增广的时候，判断边是否满足 level(v) = level(u) + 1 这一约束条件即可。

3. 最短增广路算法实例

接下来以图 6.16(a)所示的容量网络 G 和初始流为例讲解最短增广路算法的执行过程。

图 6.16(b)～图 6.16(d)为第 1 个阶段，其过程如下。

(1) 首先构造残留网络，如图 6.16(b)所示；其中顶点旁的数字为顶点的层次。

(2) 然后构造层次网络，如图 6.16(c)所示。在层次网络中利用 BFS 算法找到一条增广路，在图 6.16(c)中用粗线标明了增广路，沿着这条增广路可以将网络流流量增加 1。增广完毕后，<V_s, V_1>这条弧因为饱和了，所以在层次网络中将被去掉，而<V_1, V_4>、<V_4, V_t>这两条弧的容量将减 1。这样层次网络中就不存在增广路了。该阶段的增广完毕。

(3) 第 1 个阶段增广后的网络流如图 6.16(d)所示。

图 6.16(e)～图 6.16(h)为第 2 个阶段，其过程如下。

(1) 首先构造残留网络，如图 6.16(e)所示。

(2) 然后构造层次网络，如图 6.16(f)所示。在层次网络中利用 BFS 算法找到一条增广路，在图 6.16(f)中用粗线标明了增广路，沿着这条增广路可以将网络流流量增加 1。增广完毕后，在<V_2, V_1>这条弧因为饱和了，所以在层次网络中将被去掉，而<V_s, V_2>、<V_1, V_4>、<V_4, V_t>这 3 条弧的容量将减 1，如图 6.16(g)所示。然后继续用 BFS 算法从源点开始寻找增广路，在图 6.16(g)中又找到一条增广路，沿着这条增广路可以将网络流流量增加 4。此后，层次网络中就不存在增广路了。该阶段的增广完毕。

(3) 第 2 个阶段增广后的网络流如图 6.16(h)所示。

图 6.16(i)～图 6.16(j)为第 3 个阶段，其过程为：在图 6.16(i)中构造残留网络，在图 6.16(j)中构造层次网络，构造完毕后，发现汇点不在层次网络中，因此不存在增广路了。

至此，最短增广路算法结束，求得的网络最大流流量为 12。

4. 最短增广路算法复杂度分析

最短增广路算法的复杂度包括建层次网络和寻找增广路两部分。

在最短增广路算法中，最多建 n 个层次网络，每个层次网络用 BFS 一次遍历即可得到。一次 BFS 遍历的复杂度为 $O(m)$，所以建层次图的总复杂度为 $O(n\times m)$。

现在分析在每一阶段中寻找增广路的复杂度。注意到每增广一次，层次网络中必定有一条边会被删除。层次网络中最多有 m 条边，所以可以认为最多增广 m 次。在最短增广路算法中，用 BFS 来增广，一次增广的复杂度为 $O(m+n)$，其中 $O(m)$为 BFS 的花费，$O(n)$为修改流量的花费。所以在每一阶段寻找增广路的复杂度为 $O(m\times(m+n)) = O(m^2)$。因此 n 个阶段寻增广路总的复杂度为 $O(n\times m^2)$。

最短增广路算法的总复杂度即为建层次网络的总复杂度与寻找增广路的总复杂度之和，为 $O(n\times m^2)$。

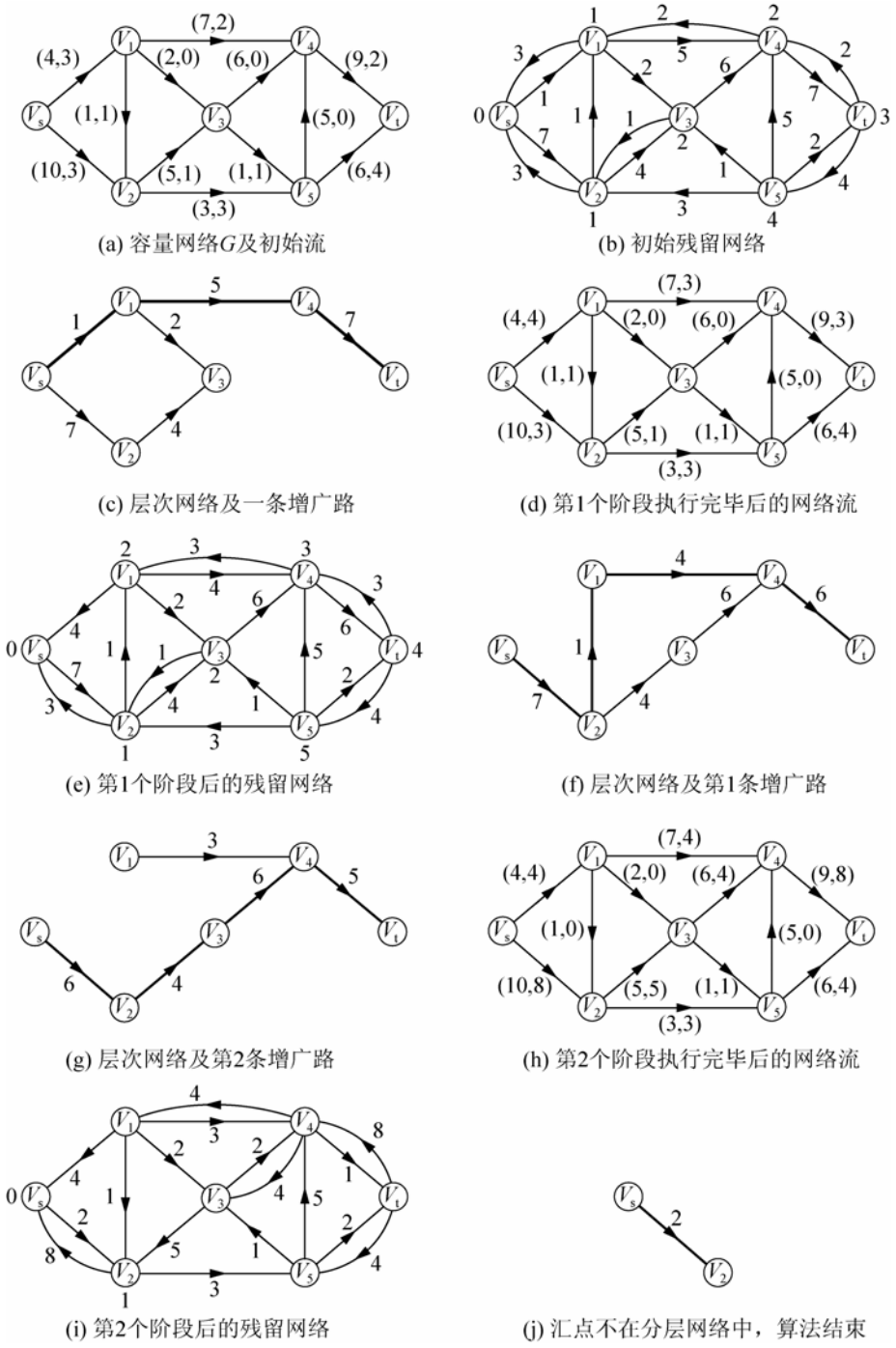


图 6.16　最短增广路算法实例

6.1.6　连续最短增广路算法——Dinic 算法

1. Dinic 算法思路

Dinic 算法的思想也是分阶段地在层次网络中增广。它与最短增广路算法不同之处是：

最短增广路算法每个阶段执行完一次 BFS 增广后，要重新启动 BFS 从源点 V_s 开始寻找另一条增广路；而在 Dinic 算法中，只需一次 DFS 过程就可以实现多次增广，这是 Dinic 算法的巧妙之处。Dinic 算法的具体步骤如下。

(1) 初始化容量网络和网络流。

(2) 构造残留网络和层次网络，若汇点不在层次网络中，则算法结束。

(3) 在层次网络中用一次 DFS 过程进行增广，DFS 执行完毕，该阶段的增广也执行完毕。

(4) 转步骤(2)。

在 Dinic 的算法步骤中，只有第(3)步与最短增广路算法不同。在下面的实例中，将会发现 DFS 过程将会使算法的效率较之最短增广路算法有非常大的提高。

2. Dinic 算法实例

接下来以图 6.16(e)所示的第 2 阶段开始分析 Dinic 算法的一个阶段执行过程，图 6.17(a)就是图 6.16(e)。

图 6.17(a)～图 6.17(f)演示了 Dinic 执行的第 2 个阶段，其过程如下。

(1) 首先构造好残留网络，如图 6.17(a)所示。

(2) 构造好层次网络后，然后从源点开始执行 DFS 过程，约定 DFS 过程中在多个未访问过的邻接顶点中进行选择时，按顶点序号从小到大的顺序进行选择，那么将找到一条增广路(V_s, V_2, V_1, V_4, V_t)，如图 6.17(b)所示，沿着这条增广路进行增广，流量增加 1。

(3) 此后，DFS 过程沿着图 6.17(c)所示的虚线回退，一直回退到从源点可到达的最远顶点为止，即顶点 V_2，如图 6.17(c)所示。

(4) 然后从 V_2 继续 DFS，找到第 2 条增广路(V_s, V_2, V_3, V_4, V_t)，如图 6.17(d)所示，沿着这条增广路进行增广，流量增加 4。

(5) 此后，DFS 过程沿着图 6.17(e)所示的虚线回退，一直回退到源点，至此，DFS 执行完毕，该阶段的增广也执行完毕，增广后网络流如图 6.17(f)所示。

3. Dinic 算法复杂度分析

与最短增广路算法一样，Dinic 算法最多被分为 n 个阶段，每个阶段包括建层次网络和寻找增广路两部分，其中建立层次网络的复杂度仍是 $O(n\times m)$。

现在来分析 DFS 过程的总复杂度。在每一阶段，将 DFS 分成两部分分析。

(1) 修改增广路的流量并后退的花费。在每一阶段，最多增广 m 次，每次修改流量的费用为 $O(n)$。而一次增广后在增广路中后退的费用也为 $O(n)$。所以在每一阶段，修改增广路以及后退的复杂度为 $O(m\times n)$。

(2) DFS 遍历时的前进与后退。在 DFS 遍历时，如果当前路径的最后一个顶点能够继续扩展，则一定是沿着第 i 层顶点指向第 i+1 层顶点的边向汇点前进了一步。因为增广路径长度最长为 n，所以最多连续前进 n 步后就会遇到汇点。在前进的过程中，可能会遇到没有边能够沿着继续前进的情况，这时将路径中的最后一个点在层次图中删除。

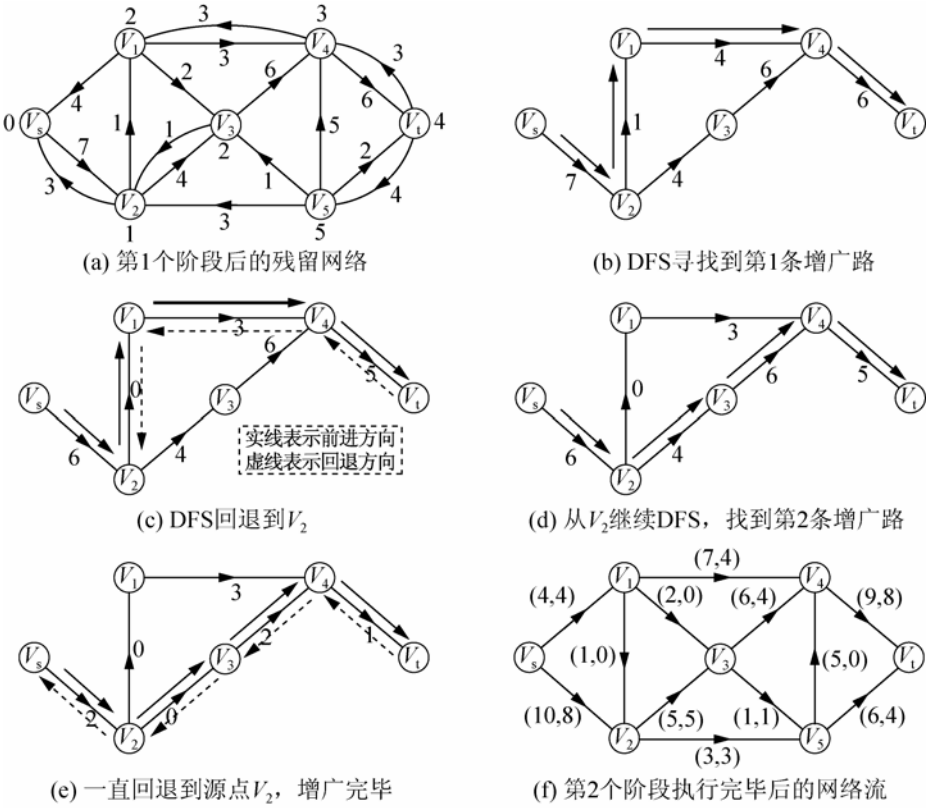
注意到每后退一次必定会删除一个点，所以后退的次数最多为 n 次。在每一阶段中，后退的复杂度为 $O(n)$。

假设在最坏情况下，所有的点最后均被删除，一共后退了 n 次，这也就意味着，有 n 次的前进被“无情”地退了回来，这 n 次前进操作都没有起到“寻找增广路”的作用。除

去这 n 次前进和 n 次后退，其余的前进都对最后找到增广路做了贡献。增广路最多找 m 次，每次最多前进 n 个点。所以所有前进操作最多为 $n+m\times n$ 次，复杂度为 $O(m\times n)$。

于是得到，在每一阶段中，DFS 遍历时前进与后退的花费为 $O(m\times n)$。

综合以上两点，一次 DFS 的复杂度为 $O(m\times n)$。因为最多进行 n 次 DFS，所以在 Dinic 算法中找增广路的总复杂度为 $O(m\times n^2)$。因此，Dinic 算法的总复杂度即为建层次网络的总复杂度与寻找增广路的总复杂度之和，为 $O(m\times n^2)$。



(a) 第1个阶段后的残留网络　(b) DFS寻找到第1条增广路

(c) DFS回退到V_2　(d) 从V_2继续DFS，找到第2条增广路

(e) 一直回退到源点V_2，增广完毕　(f) 第2个阶段执行完毕后的网络流

图 6.17　Dinic 算法实例

6.1.7　一般预流推进算法

1. 增广路算法的缺点

增广路算法的特点是找到增广路后，立即沿增广路对网络流进行增广。每一次增广可能需要对最多 $n-1$ 条弧进行操作，因此，每次增广的复杂度为 $O(n)$，在有些情况下，这个代价是很高的。如图 6.18 所示是一个极端的例子。在图 6.18 所示的容量网络中，无论采用何种增广路算法，都会找到 10 条增广路，每条路长为 10，容量为 1。因此，总共需要 10 次增广，每次增广 1 个单位，每次增广时需要对 10 条弧进行操作。

通过观察发现，10 条增广路中的前 9 个顶点(前 8 条弧)是完全一样的，能否直接将前 8 条弧的流量增广 10 个单位，而只对后面长度为 2 的、不同的有向路单独操作呢？这就是预流推进算法(Preflow Push Algorithm)的思想。也就是说，预流推进算法关注于对每一条弧的操作和处理，而不必一次一定处理一条增广路。

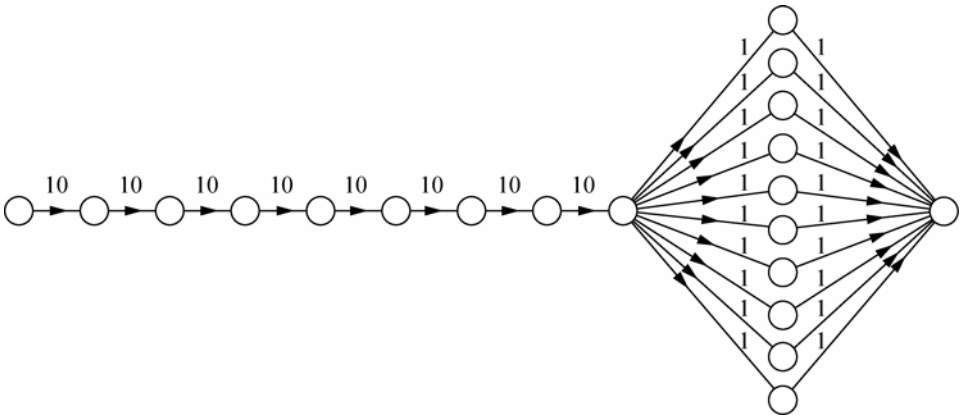


图 6.18　增广路算法的缺点

2. 距离标号

设容量网络为 $G(V, E)$，f 是其可行流，对于一个残留网络 $G'(V, E)$，如果一个函数 d 将顶点集合 V 映射到非负整数集合，则称 d 是关于残留网络 G'的**距离函数**(Distance Function)。$d(u)$称为顶点 u 的**距离标号**(Distance Label)。

如果距离函数 d 满足：① $d(V_t) = 0$；② 对 G'中的任意一条弧$<u, v>$，有 $d(u) \leqslant d(v)+1$。则称距离函数 d 关于流 f 是**有效的**(Valid)，或称距离标号(函数)d 是有效的。

如果任意一个顶点的距离标号正好等于残留网络中从该顶点到汇点 V_t 的最短有向路径距离(指路径上弧的数目)，则称距离函数 d 关于流 f 是**精确**(Exact)的，或称距离标号是精确的。精确的距离标号一定是有效的。

例如，图 6.19 对图 6.5(b)所示的残留网络进行距离标号，这些标号都是精确的。

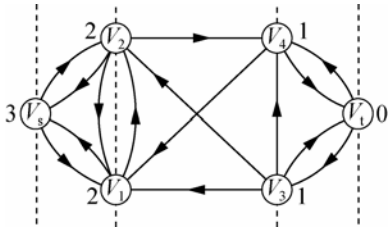


图 6.19　精确的距离标号

如果对残留网络 G'中某一条弧$<u, v>$，有 $d(u)=d(v)+1$，则称弧$<u, v>$为**允许弧**(Admissible Arc)。如果从源点到汇点的一条有向路径完全由允许弧组成，则该有向路称为**允许路**(Admissible Path)。

根据有效距离函数的定义，可以得到以下结论：若距离函数 d 是有效的，则有下面的结论。

(1) $d(u)$是残留网络 G'中从顶点 u 到汇点 V_t 的最短有向路径长度的下界。

(2) 如果 $d(V_s) \geqslant n$，则残留网络 G'中从源点 V_s 到汇点 V_t 没有有向路(即增广路)。

(3) 允许路是残留网络 G'中的最短增广路。

对于一个残留网络 G'，如何确定其精确的距离标号呢？可以从汇点 V_t 开始，对 G' 沿反

向弧进行广度优先搜索，这一过程的复杂度为 $O(m)$。可以看出，一个顶点精确的距离标号实际上表示的是从该顶点到汇点 V_t 的最短路径长度，也就是说对所有顶点按照最短路径长度进行了层次划分。例如，图 6.19 将残留网络中的顶点划分成 4 个层次，用虚线标明。

关于层次和距离标号的说明：从顶点层次(6.1.5 节)和距离标号的定义和讨论可以看出，顶点层次实质上就是一种精确的“距离标号”，只不过这种距离是从 $d(V_s)=0$ 开始进行标号的，顶点 u 的层次 level(u)为源点 V_s 到顶点 u 的最短路径长度。

3. 一般预流推进算法思想

盈余(Excess)：设 u 是容量网络 $G(V, E)$中的顶点，定义顶点 u 的盈余为流入顶点 u 的流量之和减去从顶点 u 流出的流量之和，记为 $e(u)$，即：

$$e(u)=\sum_v f(v,u)-\sum_v f(u,v) \tag{6-7}$$

式中：$\sum_v f(v,u)$ 表示流入顶点 u 的流量总和；$\sum_v f(u,v)$ 表示从顶点 u 流出的流量总和。

活跃顶点(Active Vertex)：容量网络 G 中，$e(u)>0$ 的顶点 $u(u\neq V_s、V_t)$称为活跃顶点。

预流(Preflow)：设 $f=\{f(u,v)\}$是容量网络 $G(V, E)$上的一个网络流，如果 G 的每一条弧$<u, v>$都满足：

$$0\leqslant f(u,v)\leqslant c(u,v),\ <u,v>\in E \tag{6-8}$$

另外，除源点 V_s、汇点 V_t 外每个顶点 u 的盈余 $e(u)$都满足：

$$e(u)\geqslant 0,\ u\neq V_s、V_t \tag{6-9}$$

则称该网络流 f 为 G 的预流。

对容量网络 G 的一个预流 f，如果存在活跃顶点，则说明该预流不是可行流。预流推进算法就是要选择活跃顶点，并通过它把一定的流量推进到它的邻接顶点，尽可能将正的盈余减少为 0。如果当前活跃顶点有多个邻接顶点，那么首先应推进到哪个邻接顶点呢？由于算法最终目的是尽可能将流量推进到汇点 V_t，因此算法总是首先寻求将流量推进到距离汇点 V_t 最近的邻接顶点中。由于每个顶点的距离标号可以表示顶点到汇点 V_t 的距离，因此算法总是将流量沿着允许弧推进。如果从当前活跃顶点出发没有允许弧，则增加该顶点的距离标号，使得从当前活跃顶点出发至少有一条允许弧。

预流推进算法的基本框架如下。

(1) (预处理)取零流作为初始可行流，即 $f=\{0\}$，对源点 V_s 发出的每条$<V_s, u>$，令 $f(V_s, u)=c(V_s, u)$；对任意的顶点 $v\in V$，计算精确的距离标号 $d(v)$；令 $d(V_s)=n$。

(2) 如果残留网络 $G'(V', E')$中不存在活跃顶点，则算法结束，已经求得最大流，否则进入第(3)步。

(3) 在残留网络中选取活跃顶点 u；如果存在顶点 u 的某条出弧$<u, v>$为允许弧，则将 $\min\{e(u), c'(u, v)\}$流量的流从顶点 u 推进到顶点 v；否则令 $d(u)=\min\{d(v)+1 \mid <u, v>\in E'$，且 $c'(u, v)>0\}$。转第(2)步。$c'(u, v)$为残留网络中弧$<u, v>$的容量。

可见，算法的每次迭代是一次推进操作(第(3)步的前面部分)，或者一次重新标号操作(第(3)步的后面部分)。对于推进操作，如果推进的流量等于弧上的残留容量，则称为**饱和推进**(Saturating Push)，否则称为**非饱和推进**(Nonsaturating Push)。当算法终止时，网络中不含有活跃顶点，因此源点 V_s 和汇点 V_t 的盈余不为 0。所以，此时得到的预流实际上已经是

一个可行流。又由于在算法预处理已经令源点的距离标号 $d(V_s)=n$，而距离标号在计算过程中不会减少，因此算法在计算过程中可以保证网络中永远不会有增广路存在。根据增广路定理，算法终止时一定得到了最大流。

4. 一般预流推进算法求解实例

本小节以图 6.20(a)所示的容量网络 G 为例演示一般预流推进算法的求解过程。在该容量网络中，源点为 V_s，汇点为 V_t，只有顶点 V_1 和 V_2 可能为活跃顶点。

首先进行预处理，得到图 6.20(b)所示的残留网络(注意在预处理中将源点 V_s 发出的每条弧的流量设置为其容量)。顶点旁的两个数字分别表示顶点的盈余 $e(u)$和距离标号 $d(u)$。在图 6.20(b)中，存在两个活跃顶点 V_1 和 V_2，假设选择 V_1，沿弧<V_1, V_t >推进 1 个单位流量，得到残留网络 6.20(c)。

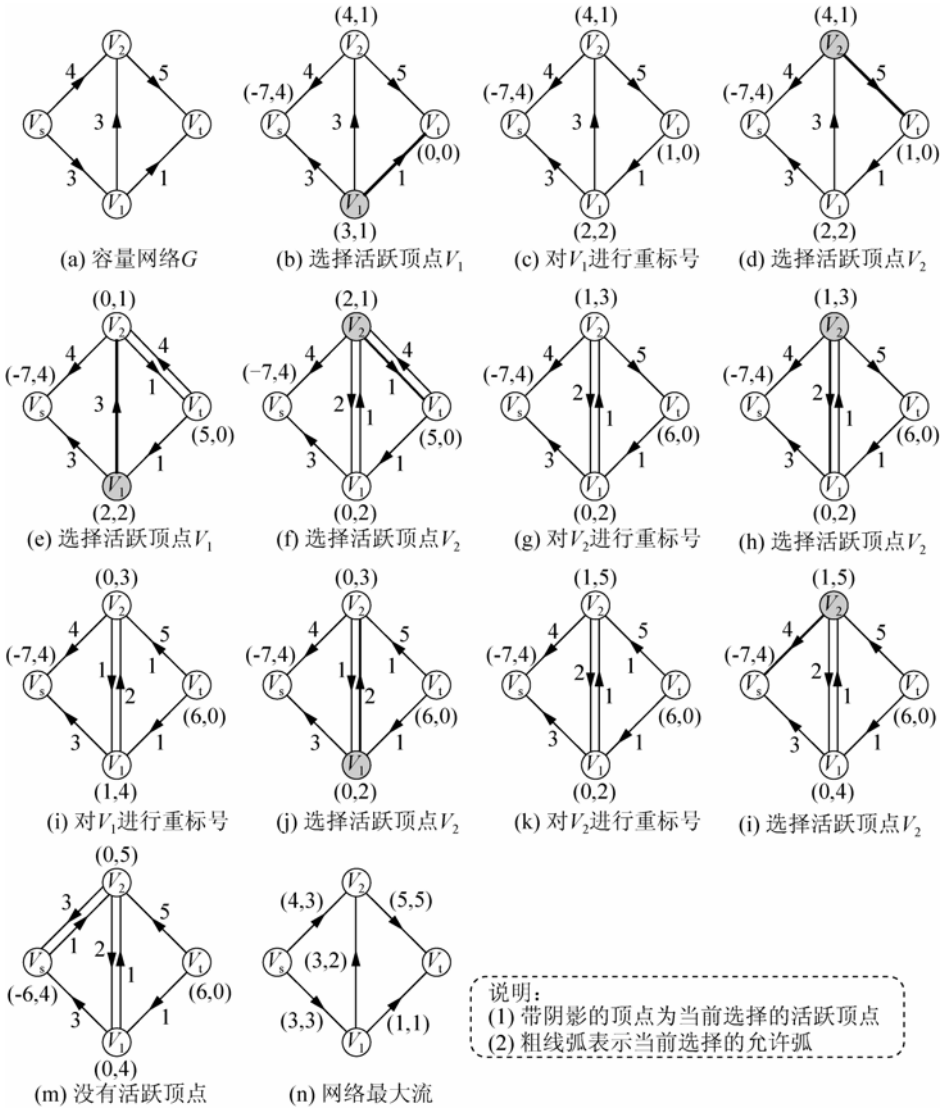


图 6.20　一般预流推进算法实例

在图 6.20(c)中，顶点 V_1 和 V_2 仍是活跃顶点，假设继续选择顶点 V_1，由于没有从顶点 V_1 发出的允许弧，因此将 V_1 的标号修改为 2。

在图 6.20(d)中，此时顶点 V_1 和 V_2 仍是活跃顶点，假设选择顶点 V_2，沿弧<V_2, V_t>推进 4 个单位流量，得到残留网络如图 6.20(e)所示。

在图 6.20(e)中，只有顶点 V_1 是活跃顶点，沿弧<V_1, V_2>推进 2 个单位流量，得到残留网络如图 6.20(f)所示。

在图 6.20(f)中，只有顶点 V_2 是活跃顶点，沿弧<V_2, V_t>推进 1 个单位流量，得到残留网络如图(g)所示。

在图 6.20(g)中，只有顶点 V_2 是活跃顶点，由于没有从顶点 V_2 发出的允许弧，因此将 V_2 的标号修改为 3。

在图 6.20(h)中，只有顶点 V_2 是活跃顶点，沿弧<V_2, V_1>推进 1 个单位流量，得到残留网络如图 6.20(i)所示。

在图 6.20(i)中，只有顶点 V_1 是活跃顶点，由于没有从顶点 V_1 发出的允许弧，因此将 V_2 的标号修改为 4。

在图 6.20(j)中，只有顶点 V_1 是活跃顶点，沿弧<V_1, V_2>推进 1 个单位流量，得到残留网络如图 6.20(k)所示。

在图 6.20(k)中，只有顶点 V_2 是活跃顶点，由于没有从顶点 V_2 发出的允许弧，因此将 V_2 的标号修改为 5。

在图 6.20(l)中，只有顶点 V_2 是活跃顶点，沿弧<V_2, V_s>推进 1 个单位流量，得到残留网络如图 6.20(m)所示。

在图 6.20(m)中，除源点 V_s、汇点 V_t 外没有活动顶点了，至此，算法结束，求得的网络最大流如图 6.20(n)所示，流量为 6。

5. 一般预流推进算法的复杂度

一般预流推进算法的时间复杂度为 $O(n^2m)$。证明略。

6.1.8 最高标号预流推进算法

从前面对一般预流推进算法的演示实例可以看出，该算法的瓶颈在于非饱和推进：每当选定一个活跃顶点后，如果执行的是一次非饱和推进，则该顶点仍然是活跃顶点，但紧接着的下一次迭代可能选择另一个新的活动顶点。最高标号预流推进算法的思想是从具有最大距离标号的活跃节点开始预流推进。之所以按照这样的顺序做，一个直观的想法是：使得距离标号较小的活跃顶点累积尽可能多地来自距离标号较大的活跃顶点的流量，然后对累积的盈余进行推进，可能会减少非饱和推进的次数。

最高标号预流推进算法的时间复杂度为 $O(n^2m^{1/2})$。证明略。

6.1.9 网络最大流算法总结

前面介绍了求解网络最大流的两大类共 5 种算法，表 6-1 对这 5 种算法做了总结。

表 6-1 网络最大流算法总结

	算法名称	复杂度	算法概要
增广路方法(Augmenting Path Method)	一般增广路算法(Generic Augmenting Path Algorithm)	$O(nmU)$	采取标号法每次在容量网络中寻找一条增广路进行增广(或者在残留网络中，每次任意找一条增广路径增广)，直至不存在增广路为止
	最短增广路算法(Shortest Augmenting Path)	$O(nm^2)$	每个阶段：在层次网络中，不断用 BFS 算法进行增广直至不存在增广路为止。如果汇点不在层次网络中，则算法结束
	连续最短增广路算法－Dinic 算法(Successive Shortest Augmenting path Algorithm)	$O(n^2m)$	在最短增广路算法的基础上改造：在每个阶段，用一个 DFS 过程实现多次增广。如果汇点不在层次网络中，则算法结束
预流推进方法(Preflow-Push Method)	一般预流推进算法(Generic Preflow-Push Algorithm)	$O(n^2m)$	维护一个预流，不断地对活跃顶点执行推进(Push)操作或重标号(Relabel)操作来调整这个预流，直到不能操作
	最高标号预流推进算法(Highest-Label Preflow-Push Algorithm)	$O(n^2m^{1/2})$	每次检查具有最高标号的活跃结点

其中：n 为顶点数，m 为弧的数目，U 表示网络中各条弧的最大容量。

6.1.10 例题解析

以下通过 5 道例题的分析，详细介绍容量网络建模方法、各种求解网络最大流的算法思想及实现方法。

例 6.2 迈克卖猪问题(PIGS)

题目来源：

Croatia 2002 Final Exam—First day, POJ1149

题目描述：

迈克在一个养猪场工作，养猪场里有 M 个猪圈，每个猪圈都上了锁。由于迈克没有钥匙，所以他不能打开任何一个猪圈。要买猪的顾客一个接一个来到养猪场，每个顾客有一些猪圈的钥匙，而且他们要买一定数量的猪。某一天，所有要到养猪场买猪的顾客，他们的信息是要提前让迈克知道的。这些信息包括：顾客所拥有的钥匙(详细到有几个猪圈的钥匙、有哪几个猪圈的钥匙)、要购买的数量。这样对迈克很有好处，他可以安排销售计划以便卖出的猪的数目最大。

更详细的销售过程为：当每个顾客到来时，他将那些他拥有钥匙的猪圈全部打开；迈克从这些猪圈中挑出一些猪卖给他们；如果迈克愿意，迈克可以重新分配这些被打开的猪圈中的猪；当顾客离开时，猪圈再次被锁上。注意：猪圈可容纳的猪的数量没有限制。

编写程序，计算迈克这一天能卖出猪的最大数目。

输入描述：

输入格式如下。

(1) 第 1 行是两个整数：M 和 N(1≤M≤1 000，1≤N≤100)。M 是猪圈的数目，N 是顾客的数目。猪圈的编号从 1 到 M，顾客的编号从 1 到 N。

(2) 第二行是 M 个整数，为每个猪圈中初始时猪的数目，范围是[0, 1 000]。

(3) 接下来的 N 行是顾客的信息，第 i 个顾客的信息保存在第 i+2 行。格式为：$A\ K_1\ K_2 \cdots K_A\ B$。$A$ 为拥有钥匙的数目，K_j 表示拥有第 K_j 个猪圈的钥匙，B 为该顾客想买的猪的数目。A、B 均可为 0。

输出描述：

输出有且仅有一行，为迈克能够卖掉的猪的最大数目。

样例输入：

```
3 3
3 1 10
2 1 2 2
2 1 3 3
1 2 6
```

样例输出：

```
7
```

分析：

本题的关键在于如何构造一个容量网络。在本题中，容量网络的构造方法如下。

(1) 将顾客看作除源点和汇点以外的结点，并且另设两个结点：源点和汇点。

(2) 源点和每个猪圈的第 1 个顾客连边，边的权是开始时猪圈中猪的数目。

(3) 若源点和某个结点之间有重边，则将权合并(因此源点流出的流量就是所有的猪圈能提供的猪的数目)。

(4) 顾客 j 紧跟在顾客 i 之后打开某个猪圈，则边<i, j>的权是+∞；这是因为，如果顾客 j 紧跟在顾客 i 之后打开某个猪圈，那么迈克就有可能根据顾客 j 的需求将其他猪圈中的猪调整到该猪圈，这样顾客 j 就能买到尽可能多的猪。

(5) 每个顾客和汇点之间连边，边的权是顾客所希望购买的猪的数目(因此汇点的流入量就是每个顾客所购买的猪的数目)。

例如，对本题样例输入中的测试数据所构造的容量网络如图 6.21(a)所示。其过程如下。

(1) 因为有 3 个顾客，所以除源点和汇点外，还有 3 个顶点 V_1、V_2 和 V_3。

(2) 第 1 个猪圈的第一个顾客是 V_1，第 2 个猪圈的第 1 个顾客是 V_1，第 3 个猪圈的第 1 个顾客是 V_2，因此源点到顶点 V_1 有重边，合并后，权值为 3 + 1 = 4，源点到顶点 V_2 有一条边，权值为 10。

(3) 顾客 V_2 紧跟在 V_1 后面打开第 1 个猪圈，顾客 V_3 紧跟在 V_1 后面打开第 2 个猪圈，因此顶点 V_1 到 V_2、V_1 到 V_3 都有边，其权值为+∞。

(4) 每个顾客 V_1、V_2 和 V_3 到汇点 V_t 都有一条边，其权值分别为 2、3、6。

构造好容量网络后，从初始流(零流)出发进行标号、调整，如图 6.21(b)～6.21(h)所示。其过程如下。

图 6.21(b)对初始网络流进行第 1 次标号，求得的增广路为：$P(V_s, V_1, V_t)$，可改进量 α =2；调整后得到的网络流如图 6.21(c)所示。

图 6.21(d)对第 1 次调整后的网络流进行第 2 次标号，求得的增广路为：$P(V_s, V_2, V_t)$，可改进量 $\alpha = 3$；调整后得到的网络流如图 6.21(e)所示。

图 6.21(f)对第 2 次调整后的网络流进行第 3 次标号，求得的增广路为：$P(V_s, V_1, V_3, V_t)$，可改进量 $\alpha = 2$；调整后得到的网络流如图 6.21(g)所示。

在图 6.21(h)中对第 3 次调整后的网络流进行第 4 次标号时，汇点 V_t 无法获得标号，至此，标号过程结束，求得的网络最大流流量为 7。

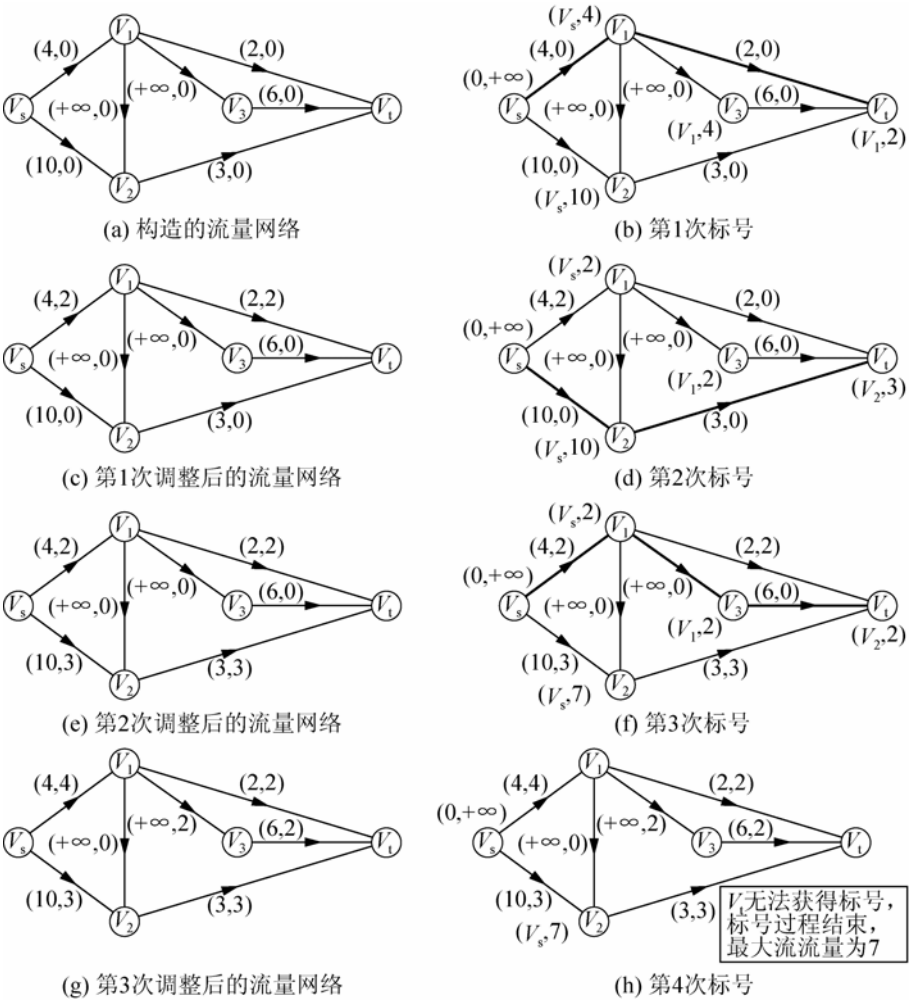


图 6.21　迈克卖猪问题：容量网络的构造及最大流的求解

代码如下：

```
#define INF 300000000           //无穷大
#define MAXM 1000               //猪圈数：1<=M<=1000
#define MAXN 100                //顾客数：1<=N<=100
int s, t;                       //源点，汇点
int customer[MAXN+2][MAXN+2];   //N+2 个结点(包括源点,汇点)之间的容量 Cij
int flow[MAXN+2][MAXN+2];       //结点之间的流量 Fij
int i, j;                       //循环变量
```

```
void init( )                    //初始化函数，构造网络流
{
    int M, N;                   //M 是猪圈的数目，N 是顾客的数目
    int num;                    //每个顾客拥有钥匙的数目
    int k;                      //第 K 个猪圈的钥匙
    int house[MAXM];            //存储每个猪圈中猪的数目
    int last[MAXM];             //存储每个猪圈的前一个顾客的序号
    memset(last,0, sizeof(last)); memset( customer, 0, sizeof(customer) );
    scanf( "%d%d", &M, &N );
    s=0;  t=N+1;                //源点、汇点
    for( i=1; i<=M; i++ )       //读入每个猪圈中猪的数目
        scanf( "%d", &house[i] );
    for( i=1; i<=N; i++ )       //构造网络流
    {
        scanf( "%d", &num );    //读入每个顾客拥有钥匙的数目
        for( j=0; j<num; j++ )
        {
            scanf( "%d", &k );  //读入钥匙的序号
            if( last[k]==0 )    //第 i 个顾客是第 k 个猪圈的第 1 个顾客
                customer[s][i]=customer[s][i]+house[k];
            else    //last[k]!=0,表示顾客 i 紧跟在顾客 last[k]后面打开第 k 个猪圈
                customer[ last[k] ][i]=INF;
            last[k] = i;
        }
        scanf("%d",&customer[i][t]); //每个顾客到汇点的边,权值为顾客购买猪的数量
    }
}
void ford( )
{
    //可改进路径上该顶点的前一个顶点的序号,相当于标号的第 1 个分量,
    //初始为-2 表示未标号,源点的标号为-1
    int prev[ MAXN+2 ];
    int minflow[ MAXN+2 ]; //每个顶点的可改进量 α，相当于标号的第 2 个分量
    //采用广度优先搜索的思想遍历网络，从而对所有顶点进行标号
    int queue[MAXN+2];          //相当于 BFS 算法中的队列
    int qs, qe;                 //队列头位置、队列尾位置
    int v;                      //当前检查的顶点
    int p;                      //用于保存 Cij-Fij
    for( i=0; i<MAXN+2; i++ )   //构造零流:从零流开始标号调整
    {
        for( j=0; j<MAXN+2; j++ )  flow[i][j]=0;
    }
    minflow[0]=INF;     //源点标号的第 2 个分量为无穷大
    while( 1 )          //标号法
    {
        for( i=0; i<MAXN+2; i++ )  //每次标号前，每个顶点重新回到未标号状态
            prev[i]=-2;
```

```
            prev[0]=-1;                         //源点
            qs=0;  queue[qs]=0;  qe=1;  //源点(顶点0)入队列
            //标号过程：如果qe>=qs(相当于队列空)，标号也无法再进行下去
            while( qs<qe && prev[t]==-2 )
            {
                v=queue[qs];  qs++; //取出队列头顶点
                for( i=0; i<t+1; i++ )
                {
                    //如果顶点i是顶点v的"邻接"顶点，则考虑是否对顶点i进行标号
                    //customer[v][i]-flow[v][i]!=0能保证顶点i是v的邻接顶点,且能
                      进行标号
                    //顶点i未标号，并且Cij-Fij>0
                    if( prev[i]==-2 && ( p=customer[v][i]-flow[v][i]) )
                    {
                        prev[i]=v;  queue[qe]=i;  qe++;
                        minflow[i]=(minflow[v]<p) ? minflow[v] : p;
                    }
                }
            }

            if( prev[t]==-2 )  break;   //汇点t没有标号，标号法结束
            for( i=prev[t], j=t; i!=-1; j=i, i=prev[i] )   //调整过程
            {
                flow[i][j]=flow[i][j] + minflow[t];
                flow[j][i]=-flow[i][j];
            }
        }
        for( i=0, p=0; i<t; i++ )  //统计进入汇点的流量，即为最大流的流量
            p=p+flow[i][t];
        printf( "%d\n", p );
    }
    int main( )
    {
        init( );
        ford( );
        return 0;
    }
```

例 6.3 排水沟(Drainage Ditches)

题目来源：

USACO 93, POJ1273

题目描述：

每次下雨的时候，农场主 John 的农场里就会形成一个池塘，这样就会淹没其中一小块土地，在这块土地上种植了 Bessie 最喜欢的苜蓿。这意味着苜蓿要被水淹没一段时间，而后要花很长时间才能重新长出来。因此，John 修建了一套排水系统，这样种植了苜蓿的土地就不会被淹没。雨水被排到了附近的一条小河中。作为一个一流的工程师，John 还在每

条排水沟的起点安装了调节阀门，这样可以控制流入排水沟的水流的速度。

John 不仅知道每条排水沟每分钟能排多少加仑的水，而且还知道整个排水系统的布局，池塘里的水通过这个排水系统排到排水沟；并最终排到小河中，构成一个复杂的排水网络。

给定排水系统，计算池塘能通过这个排水系统排水到小河中的最大流水速度。每条排水沟的流水方向是单方向的，但在排水系统中，流水可能构成循环。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为两个整数 M 和 N，用空格隔开，$0 \leqslant M \leqslant 200$，$2 \leqslant N \leqslant 200$，其中 M 是排水沟的数目，N 是这些排水沟形成的汇合结点数目。结点 1 为池塘，结点 N 为小河。接下来有 M 行，每行描述了一条排水沟，用 3 个整数来描述：S_i，E_i 和 C_i，其中 S_i 和 $E_i(1 \leqslant S_i, E_i \leqslant N)$标明了这条排水沟的起点和终点，水流从 S_i 流向 E_i，C_i $(0 \leqslant C_i \leqslant 10\ 000\ 000)$表示通过这条排水沟的最大流水速度。

输出描述：

对输入文件中的每个测试数据，输出一行，为一个整数，表示整个排水系统可以从池塘排出水的最大速度。

样例输入：

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

样例输出：

```
50
```

分析：

很明显，这道题目就是求容量网络的最大流，样例输入中测试数据所描述的容量网络如图 6.22(a)所示，最终求得的网络最大流如图 6.22(b)所示，最大流的流量为 50，就是题目所要求输出的值。

本题采用一般增广路算法求解，在 solve 函数中调用 find_augment_path 函数求增广路，并通过 augment_flow 函数计算可改进量，然后调用 update_flow 函数更新网络流，如此循环直至网络流中不存在增广路为止。

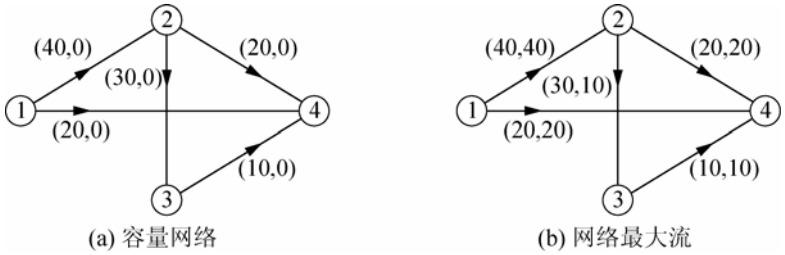


图 6.22　排水沟

代码如下：

```
#define MAXN 210
struct Matrix
{
```

```
    int c, f;               //容量，流量
};
Matrix Edge[MAXN][MAXN];    //流及容量(邻接矩阵)
int M, N;                   //读入的排水沟(即弧)数目和汇合结点(即顶点)数目
int s, t;                   //源点(结点1)、汇点(结点N)
int residual[MAXN][MAXN];   //残留网络
int qu[MAXN*MAXN], qs, qe;  //队列、队列头和尾
int pre[MAXN];              //pre[i]为增广路上顶点i前面的顶点序号
int vis[MAXN];              //BFS算法中各顶点的访问标志
int maxflow, min_augment;   //最大流流量、每次增广时的可改进量
void find_augment_path( )   //BFS求增广路
{
    int i, cu;              //cu为队列头顶点
    memset( vis, 0, sizeof(vis) );
    qs=0;  qu[qs]=s;        //s入队列
    pre[s]=s;  vis[s]=1;  qe=1;
    memset( residual, 0, sizeof(residual) );  memset( pre, 0, sizeof(pre) );
    while( qs<qe && pre[t]==0 )
    {
        cu=qu[qs];
        for( i=1; i<=N; i++ )
        {
            if( vis[i]==0 )
            {
                if( Edge[cu][i].c - Edge[cu][i].f >0 )
                {
                    residual[cu][i]=Edge[cu][i].c-Edge[cu][i].f;
                    pre[i]=cu;  qu[qe++]=i;  vis[i]=1;
                }
                else if( Edge[i][cu].f>0 )
                {
                    residual[cu][i]=Edge[i][cu].f;
                    pre[i]=cu;  qu[qe++]=i;  vis[i]=1;
                }
            }
        }
        qs++;
    }
}
void augment_flow( )    //计算可改进量
{
    int i=t, j;         //t为汇点
    if( pre[i]==0 )
    {
        min_augment=0;  return;
    }
    j=0x7fffffff;
    while( i!=s )       //计算增广路上可改进量的最小值
    {
        if( residual[pre[i]][i]<j )  j=residual[pre[i]][i];
```

```
            i=pre[i];
        }
        min_augment=j;
    }
    void update_flow( )//调整流量
    {
        int i=t;        //t 为汇点
        if( pre[i]==0 )  return;
        while( i!=s )
        {
            if( Edge[pre[i]][i].c-Edge[pre[i]][i].f>0 )
                Edge[pre[i]][i].f+=min_augment;
            else if( Edge[i][pre[i]].f>0 )  Edge[pre[i]][i].f+=min_augment;
            i=pre[i];
        }
    }
    void solve( )
    {
        s=1;  t=N;
        maxflow=0;
        while( 1 )
        {
            find_augment_path( );              //BFS 寻找增广路
            augment_flow( );                   //计算可改进量
            maxflow+=min_augment;              //
            if(min_augment>0) update_flow( );  //更新流
            else return;
        }
    }
    int main( )
    {
        int i;
        int u, v, c;
        while( scanf("%d %d",&M,&N)!=EOF )
        {
            memset( Edge, 0, sizeof(Edge) );
            for( i=0; i<M; i++ )
            {
                scanf( "%d %d %d", &u, &v, &c );
                Edge[u][v].c+=c;
            }
            solve( );
            printf( "%d\n", maxflow );
        }
        return 0;
    }
```

例 6.4　最优的挤奶方案(Optimal Milking)

题目来源：

USACO 2003 US Open, POJ2112

题目描述：

农场主 John 将他的 $K(1\leqslant K\leqslant 30)$个挤奶器运到牧场，在那里有 $C(1\leqslant C\leqslant 200)$头奶牛，在奶牛和挤奶器之间有一组不同长度的路。$K$ 个挤奶器的位置用 1～K 的编号标明，奶牛的位置用 K+1～K+C 的编号标明。

每台挤奶器每天最多能为 $M(1\leqslant M\leqslant 15)$头奶牛挤奶。

试编写程序，寻找一个方案，安排每头奶牛到某个挤奶器挤奶，并使得 C 头奶牛需要走的所有路程中的最大路程最小。每个测试数据中至少有一个安排方案。每头奶牛到挤奶器有多条路。

输入描述：

测试数据的格式如下。

第 1 行为 3 个整数 K、C 和 M。

第 2～K+C+1 行，每行有 K+C 个整数，描述了奶牛和挤奶器(二者合称实体)之间的位置，这 K+C 行构成了一个沿对角线对称的矩阵。第 2 行描述了第 1 个挤奶器距离其他实体的距离，……，第 K+1 行描述了第 K 个挤奶器距离其他实体的距离；第 K+2 行描述了第 1 头奶牛距离其他实体的距离，……这些距离为不超过 200 的正数。实体之间如果没有直接路径相连，则距离为 0。实体与本身的距离(即对角线上的整数)也为 0。

输出描述：

输出一个整数，为所有方案中，C 头奶牛需要走的最大距离的最小值。

样例输入：

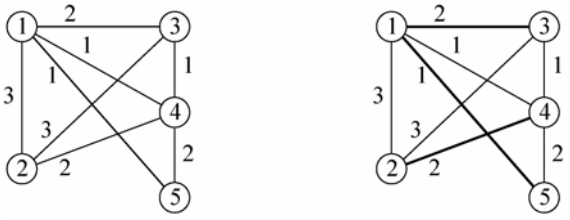
```
2 3 2
0 3 2 1 1
3 0 3 2 0
2 3 0 1 0
1 2 1 0 2
1 0 0 2 0
```

样例输出：

```
2
```

分析：

本题要安排 C 头奶牛到某个挤奶器，使得每头奶牛需要走的路程中最大路程的距离最小。例如，样例输入数据所描绘的挤奶器和奶牛之间的距离如图 6.23(a)所示，图 6.23(b)给出了一个最优的方案，粗线边表示安排奶牛到指定的挤奶器，在这个方案中，3 头奶牛需要走的距离分别为 2、1、2，所以最长距离是 2。



(a) 样例输入数据　　(b) 最优方案

图 6.23　最优挤奶方案

本题的求解算法：先用 Floyd 算法求出能达到的任意两点之间的最短路径，然后用 Dinic 算法求最大流；搜索最大距离的最小值采用二分法进行。

代码如下：

```
#define MAX 300
#define INF 10000000
int dis[MAX][MAX];      //任意连点间的最短路径
int map[MAX][MAX];      //容量网络
bool sign[MAX][MAX];    //层次网络
bool used[MAX];         //标志数组
int K, C, n, M;
int min( int a,  int b )
{
    return a<b ? a:b;
}
void Bulid_Graph( int min_max )        //构建容量网络
{
    int i, j;
    memset( map, 0, sizeof( map ) );   //初始化
    for( i=K+1; i<=n; i++ )  map[0][i]=1;
    for( i=1; i<=K; i++ )  map[i][n+1]=M;
    for( i=K+1; i<=n; i++ )
    {
        for( j=1; j<=K; j++ )
        {
            if( dis[i][j]<=min_max )  map[i][j]=1;
        }
    }
}
bool BFS( )  //BFS 构建层次网络
{
    //初始化
    memset( used, 0, sizeof( used ) );
    memset( sign, 0, sizeof( sign ) );
    int queue[100*MAX]={0};
    queue[0]=0;
    used[0]=1;
    int t=1, f=0;
    while( f<t )
    {
        for( int i=0; i<=n+1; i++ )
        {
            if( !used[i]&&map[queue[f]][i] )
            {
                queue[t++]=i;
                used[i]=1;
                sign[queue[f]][i]=1;
            }
        }
        f++;
    }
```

```
    if( used[n+1] )  return true;   //汇点在层次网络中
    else  return false;             //汇点不在层次网络中
}
int DFS( int v, int sum )  //DFS 增广
{
    int i, s, t;
    if( v==n+1 ) return sum;
    s=sum;
    for( i=0; i<=n+1; i++ )
    {
        if( sign[v][i] )
        {
            t=DFS( i, min( map[v][i], sum ) ); //递归调用
            map[v][i]-=t;
            map[i][v]+=t;
            sum-=t;
        }
    }
    return s-sum;
}
int main( )
{
    int i, j, k, L, R, mid, ans;
    scanf( "%d%d%d", &K, &C, &M );
    n=K+C;
    //Floyd 算法,求任意两点间的最短距离
    for( i=1; i<=n; i++ )
    {
        for( j=1; j<=n; j++ )
        {
            scanf( "%d", &dis[i][j] );
            if( dis[i][j]==0 ) dis[i][j]=INF;
        }
    }
    for( k=1; k<=n; k++ )
    {
        for( i=1; i<=n; i++ )
        {
            if( dis[i][k]!=INF )
            {
                for( j=1; j<=n; j++ )
                dis[i][j]=min( dis[i][k]+dis[k][j], dis[i][j] );
            }
        }
    }
    L=0, R=10000;
    //二分法搜索
    while( L < R )
    {
        mid=( L+R )/2;
        ans=0;
```

```
            //应用 Dinic 算法求最大流
            Bulid_Graph( mid ); //构建容量网络(残余网络)
            while( BFS() ) ans+=DFS( 0, INF ); //构建层次网络,并进行 DFS 增广
            if( ans>=C )  R=mid;
            else  L=mid+1;
        }
        printf( "%d\n", R );
        return 0;
    }
```

例 6.5　电网(Power Network)

题目来源：

Southeastern Europe 2003, ZOJ1734, POJ1459

题目描述：

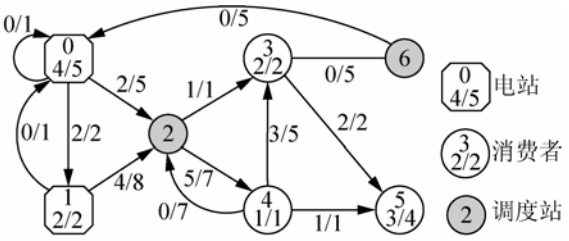
一个电网包含一些结点(电站、消费者、调度站)，这些结点通过电线连接。每个结点 u 可能被供给 $s(u)$的电能，$s(u)\geqslant 0$；同时也可能产生 $p(u)$的电能，$0\leqslant p(u)\leqslant p\max(u)$；站点 u 还有可能消费 $c(u)$电能，$0\leqslant c(u)\leqslant \min(s(u), c\max(u))$；可能传输 $d(u)$的电能，$d(u)=s(u)+p(u)-c(u)$。以上这些量存在以下限制关系：对每个电站，$c(u)=0$；对每个消费者，$p(u)=0$；对每个调度站，$p(u)=c(u)=0$。

在电网中两个结点 u 和 v 之间最多有一条电线连接。从结点 u 到结点 v 传输 $L(u, v)$的电能，$0\leqslant L(u, v)\leqslant L\max(u, v)$。定义 *Con* 为 $c(u)$的总和，表示电网中消费电能的总和。本题的目的是求 *Con* 的最大值。

电网的一个例子如图 6.24 所示。在图 6.24(a)中，电站结点 u 的标记"x/y"代表 $p(u)=x$、$p\max(u)=y$。消费者结点 u 的标记"x/y"代表 $c(u)=x$、$c\max(u)=y$。每条电线所对应的边(u, v)，其标记"x/y"代表 $L(u, v)=x$、$L\max(u, v)=y$。在图 6.24(b)中，消费的最大电能 $Con=6$，其中各结点含义如图 6.24(c)所示；图 6.24(a)列出了在此状态下各个站点的 $s(u)$、$p(u)$、$c(u)$和 $d(u)$。注意，如图 6.24(b)所示的电网中，电能的流动还存在其他状态，但消费的电能总和不超过 6。

U	类型	$s(u)$	$p(u)$	$p\max(u)$	$c(u)$	$c\max(u)$	$d(u)$
0	电站	0	4	5	\	\	4
1		2	2	2	\	\	4
3	消费者	4	\	\	2	2	2
4		5	\	\	1	1	4
5		3	\	\	3	4	0
2	调度者	6	\	\	\	\	6
6		0	\	\	\	\	0

(a) 电网中的结点



(b) 构图

图 6.24　电网

输入描述：

输入文件中包含多个测试数据。每个测试数据描述了一个电网。每个测试数据的第 1 行为 4 个整数：n np nc m，其中，$0\leqslant n\leqslant 100$，代表结点数目；$0\leqslant np\leqslant n$，代表电站数目；$0\leqslant nc\leqslant n$，代表消费者数目；$0\leqslant m\leqslant n^2$，代表传输电线的数目。接下来有 m 个三元组，$(u, v)z$，其中 u 和 v 为结点序号(结点序号从 0 开始计起)，$0\leqslant z\leqslant 1\,000$，代表 $L\max(u, v)$的值。接下

来有 *np* 个二元组，(*u*)*z*，其中 *u* 为电站结点的序号，0≤*z*≤10 000，代表 *p*max(*u*)的值；每个测试数据的最后是 *nc* 个二元组，(*u*)*z*，其中 *u* 为消费者结点的序号，0≤*z*≤10 000，代表 *c*max(*u*)的值。所有数据都是整数。除三元组(*u*, *v*)*z* 和二元组(*u*)*z* 中不含空格外，输入文件中其他位置允许出现空格。测试数据一直到文件尾。

输出描述：

对输入文件中每个测试数据所描述的电网，输出占一行，为一个整数，表示电网中能消费电能的最大值。

样例输入：

```
2 1 1 2 (0,1)20 (1,0)10 (0)15 (1)20
7 2 3 13 (0,0)1 (0,1)2 (0,2)5 (1,0)1 (1,2)8 (2,3)1 (2,4)7
(3,5)2 (3,6)5 (4,2)7 (4,3)5 (4,5)1 (6,0)5
(0)5 (1)2 (3)2 (4)1 (5)4
```

样例输出：

```
15
6
```

分析：

本题目可以采用网络最大流的模型来求解。网络最大流模型需要有一个源点和汇点，而原图中的电站、调度站和消费者结点都不能作为源点和汇点，因此在原图的基础上添加一个源点和汇点，顶点序号分别为 *n*+2 和 *n*+1。在输入文件中，顶点序号是 0 开始计起的，但在程序中，因为定义了数组 fa[MAXN]用于记录增广路，fa[*i*]为 0 表示顶点 *i* 没有标记过，fa[*i*]为 *v*(或-*v*)表示增广路上顶点 *i* 的前一个顶点为 *v*，因此在程序中，顶点序号必须从 1 开始计起。

引入源点和汇点后，对于每个电站，从源点引一条容量为 *p*max 的弧；从每个消费者，引一条容量为 *c*max 的弧到汇点；对于题目中给的三元组(*u*, *v*)*z*，从顶点 *u* 连一条容量为 *z* 的弧到顶点 *v*。这样每个消费者实际消费的电流量流入汇点，源点提供的最大电流量就是每个电站的 *p*max 之和。显然这样构造的网络最大流模型是符合题目要求的。

例如，对样例输入中的第 2 个测试数据，原图为图 6.24(b)，构造好网络最大流模型后的网络图如图 6.25 所示。

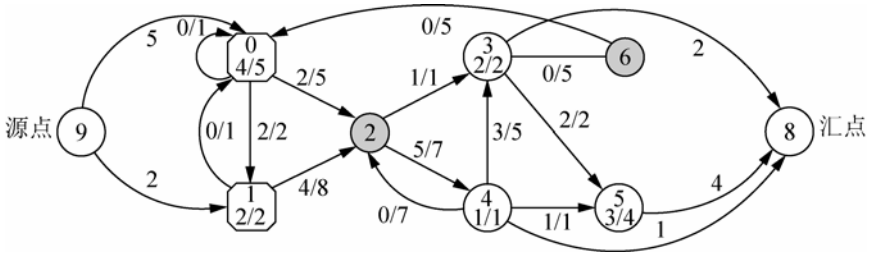


图 6.25　电网：网络最大流模型

构造好网络最大流的模型后，求解这个网络的最大流即可。下面的代码采用了预流推进算法来求解网络最大流。

代码如下：

```
using namespace std;
const int mxn=110;
const int mxf=0x7fffffff;
int n, np, nc, m;
```

```
int resi[mxn][mxn];     //残留网络
deque<int> act;         //活动定点队列
int h[mxn];             //高度
int ef[mxn];            //余流
int s, t;               //源和汇
int V;                  //定点数量
void push_relabel( )
{
    int i, j;
    int sum=0;
    int u, v, p;
    for( i=1; i<=V; i++ )
    h[i]=0;
    h[s]=V;
    memset( ef, 0, sizeof(ef) );
    ef[s]=mxf;  ef[t]=-mxf;
    act.push_front(s);
    while( !act.empty() )
    {
        u=act.back();
        act.pop_back();
        for( i=1; i<=V; i++ )
        {
            v=i;
            if( resi[u][v]<ef[u] )
                p=resi[u][v];
            else p=ef[u];
            if( p>0 && (u==s || h[u]==h[v]+1) )
            {
                resi[u][v]-=p;  resi[v][u]+=p;
                if(v==t) sum+=p;
                ef[u]-=p;  ef[v]+=p;
                if( v!=s && v!=t )
                    act.push_front(v);
            }
        }
        if(u!=s && u!=t && ef[u]>0) //如果余流不为0就将它的高度加1并放入队列
        {
            h[u]++;
            act.push_front(u);
        }
    }
    printf( "%d\n", sum );
}
int main( )
{
    int i, j;
    int u, v, val;
    while( scanf( "%d %d %d %d", &n, &np, &nc, &m )!=EOF )
    {
        s=n+1;  t=n+2;  V=n+2;
```

```
            memset( resi, 0, sizeof(resi) );
            for( i=0; i<m; i++ )
            {
                while( getchar()!='(' )  ;
                scanf( "%d,%d)%d", &u, &v, &val );
                resi[u+1][v+1]=val;
            }
            for( i=0; i<np; i++ )
            {
                while( getchar()!='(' )  ;
                scanf( "%d)%d", &u, &val );
                resi[s][u+1]=val; //把生产力引入源点
            }
            for( i=0; i<nc; i++ )
            {
                while( getchar()!='(' )  ;
                scanf( "%d)%d", &u, &val );
                resi[u+1][t]=val; //把消耗力放到汇点
            }
            push_relabel();
        }
        return 0;
    }
```

练　习

6.1　UNIX 会议室的插座(A Plug for UNIX), ZOJ1157, POJ1087

题目描述：

现在由你负责布置 Internet 联合组织首席执行官就职新闻发布会的会议室。

由于会议室修建时被设计成容纳全世界各地的新闻记者，因此会议室提供了多种电源插座用以满足(会议室修建时期)各国不同插头的类型和电压。不幸的是，会议室是很多年前修建的，那时新闻记者很少使用电子设备，所以会议室对每种插座只提供了一个。新闻发布会时，新闻记者需要使用许多电子设备，如手提电脑、麦克风、录音机、传呼机等。尽管这些设备很多可以使用电池，但是由于发布会时间很长并且是单调乏味的，记者们希望能够使用尽可能多的设备(这些设备需要使用插座)，以打发时间。

在发布会之前，你收集了记者们使用的设备的信息，开始布置会议室。你注意到有些设备的插头没有合适的插座可用。你怀疑这些设备来自那些在修建会议室时不存在的国家。对有些插座来说，有多个设备的插头可以使用。而对另一些插座来说，没有哪些设备的插头可以用得上。

为了试图解决这个问题，你光顾了附近的商店，商店出售转换器，这些转换器可以将一种插头转换成另一种插头。而且转换器可以串联。商店没有足够多的转换器类型，满足所有的插头和插座的组合，但对于已有某种转换器，总是可以提供无限多个。

输入描述：

输入文件包含多个测试数据。输入文件的第 1 行为一个整数 N，然后隔一个空行之后

是 N 个输入块，每个输入块对应一个测试数据。输入块之间用空行隔开。每个输入块格式如下。

每个输入块的第一行为一个正整数 n，$1 \leqslant n \leqslant 100$，表示会议室提供的插座个数；接下来 n 行列出了会议室提供的 n 个插座，每个插座用一个数字字母式字符串描述(至多有 24 个字符)。

接下来一行为一个正整数 m，$1 \leqslant m \leqslant 100$，表示待插入的设备个数；接下来 m 行中，每行首先是设备的名称，然后是它使用的插头的名称；插头的名称跟它所使用的插座的名称是一样的；设备名称是一个至多包含 24 个字母数字式字符的字符串；任何两个设备的名称都不同；设备名称和插头之间用空格隔开。

接下来一行为一个正整数 k，$1 \leqslant k \leqslant 100$，表明可以使用的转换器种数；接下来的 k 行，每行描述了一种转换器：首先是转换器提供的插座类型，中间是一个空格，然后是插头的类型。

输出描述：

对输入文件中的每个测试数据，输出一个非负整数，表明至少有多少个设备无法插入。

样例输入：

```
1

4
A
B
C
D
5
laptop B
phone C
pager B
clock B
comb X
3
B X
X A
X D
```

样例输出：

```
1
```

6.2　不喜欢雨的奶牛(Ombrophobic Bovines), POJ2391

题目描述：

Jack 农场主的奶牛实在是太讨厌被淋湿了。农场主决定在农场设置降雨警报，这样在快要下雨的时候可以让奶牛们都知道。他们设置设计了一个下雨撤退计划，这样在下雨之前每头奶牛都能躲到避雨点。然而，天气预报并不总是准确的。为了使得错误的天气预报影响尽可能小，他们希望尽可能晚地拉响警报，只要保证留有足够的时间让所有的奶牛都能回到避雨点就可以了。

农场有 F 块草地，$1 \leqslant F \leqslant 200$，奶牛们在草地上吃草。这些草地之间有 P 条路相连，$1 \leqslant P \leqslant 1\,500$，这些路足够宽，再多的奶牛也能同时在路上行走。

有些草地上有避雨点，奶牛们可以在此避雨。避雨点的容量是有限的，所以一个避雨点不可能容纳下所有的奶牛。草地与路相比很小，奶牛们通过时不需要花费时间。

计算警报至少需要提前多少时间拉响，以保证所有的奶牛都能到达一个避雨点。

输入描述：

测试数据的格式如下。

(1) 第 1 行为两个整数，*F* 和 *P*。

(2) 第 2～*F*+1 行，每行为两个整数，描述了一块草地，前一个整数(范围为 0～1 000)，表示在该草地吃草的奶牛数量；后一个整数(范围为 0～1 000)该草地的避雨点能容纳的奶牛数量。

(3) 第 *F*+2～*F*+*P*+1 行，每行有 3 个整数，描述了一条路，第 1 个和第 2 个整数(范围为 1～F)为这条路连接的两块草地序号，第 3 个整数(范围为 0～1000 000 000)，表示任何一头奶牛通过这条路是需要花费的时间。

输出描述：

输出所有奶牛回到避雨点所需的最少时间，如果不能保证所有的奶牛都回到一个避雨点，则输出"−1"。

样例输入：

```
3 4
7 2
0 4
2 6
1 2 40
3 2 70
2 3 90
1 3 120
```

样例输出：

```
110
```

6.3　ACM 计算机工厂(ACM Computer Factory), POJ3436

题目描述：

正如你所知道的，ACM 竞赛中所有竞赛队伍使用的计算机必须是相同的，以保证参赛者在公平的环境下竞争。这就是所有这些计算机都是同一个厂家生产的原因。

每台 ACM 计算机包含 *P* 个部件，当所有这些部件都准备齐全后，计算机就可以组装了，组装好以后就可以交给竞赛队伍使用了。计算机的生产过程是全自动的，通过 *N* 台不同的机器来完成。每台机器从一台半成品计算机中去掉一些部件，并加入一些新的部件(去除一些部件在有的时候是必须的，因为计算机的部件不能以任意的顺序组装)。每台机器用它的性能(每小时组装多少台计算机)、输入/输出规格来描述。

输入规格描述了机器在组装计算机时哪些部件必须准备好了。输入规格是由 *P* 个整数组成，每个整数代表一个部件，这些整数取值为 0、1 或 2，其中 0 表示该部件不应该已经准备好了，1 表示该部件必须已经准备好了，2 表示该部件是否已经准备好了无关紧要。

输出规格描述了该机器组装的结果。输出规格也是由 *P* 个整数组成，每个整数取值为 0 或 1，其中 0 代表该部件没有生产好，1 代表该部件生产好了。

机器之间用传输速度非常快的流水线连接，部件在机器之间传送所需的时间与机器生产时间相比是十分小的。

经过多年的运转后，ACM 计算机工厂的整体性能已经远远不能满足日益增长的竞赛需求。因此 ACM 董事会决定升级工厂。升级工厂最好的方法是重新调整流水线。ACM 董事会决定让你来解决这个问题。

输入描述：

输入文件第 1 行为两个整数：P 和 N。接下来有 N 行，描述了每台机器。第 i 台机器用 $2P+1$ 个整数来描述：$Q_i\ S_{i,1}\ S_{i,2} \cdots S_{i,p}\ D_{i,1}\ D_{i,2} \cdots D_{i,p}$，其中 Q_i 指定了机器的性能，$S_{i,j}$ 为第 j 部分的输入规格，$D_{i,k}$ 为第 k 部分的输出规格。$1 \leqslant P \leqslant 10$，$1 \leqslant N \leqslant 50$，$1 \leqslant Q_i \leqslant 10\ 000$。

输出描述：

输出文件的第 1 行为整体的最大可能性能，以及 M，表示为达到最大性能所需的接连数目。然后是 M 行，每行描述了一对连接。每对连接，设为机器 A 和机器 B 之间的连接，必须用 3 个正数来描述：$A\ B\ W$，其中 W 为每小时从机器 A 传送到机器 B 的计算机数目。

如果存在多个解，输出任意一个均可。

样例输入：

```
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
```

样例输出：

```
25 2
1 3 15
2 3 10
```

6.4　观光旅游线(Sightseeing Tour), ZOJ1992, POJ1637

题目描述：

Lund 市的市政委员会想在 Lund 市建设一条公交观光旅游线，使得游客可以游览到这个美丽的城市的各个角落。他们想使得观光旅游线经过每条街一次且仅一次，并且公交车要回到起点。就像其他城市一样，Lund 市的街道有的是单向的，有的是双向的。请帮助市政委员会判断是否能建设这样的一条观光线。

输入描述：

输入文件的第 1 行为一个正整数 n，表示输入文件中测试数据的个数。每个测试数据的第 1 行为两个正整数：m 和 s，$1 \leqslant m \leqslant 200$，$1 \leqslant s \leqslant 1\ 000$，分别表示交叉路口的数目和街道的数目；接下来有 s 行，描述了 s 条街道；每条街道用 3 个整数描述：x_i、y_i 和 d_i，$1 \leqslant x_i, y_i \leqslant m$, $0 \leqslant d_i \leqslant 1$，其中 x_i 和 y_i 为这条街道所连接的两个交叉路口，$d_i = 1$ 表示这条街道是单向的(从 x_i 到 y_i)，否则表示这条街道是双向的；假定存在一个交叉路口，从这个交叉路口可以到达其他每个交叉路口。

输出描述：

对输入文件中的每个测试数据，输出一行，为"possible"或"impossible"，分别表示可以和不可以建设这样的一条观光线。

样例输入：

```
2
5 8
2 1 0
1 3 0
4 1 1
1 5 0
```

样例输出：

```
possible
impossible
```

```
5 4 1
3 4 0
4 2 1
2 2 0
4 4
1 2 1
2 3 0
3 4 0
1 4 1
```

提示：求混合图(某些边为无向边，其他一些边为有向边的图)的欧拉回路，转换成网络流。

6.2　最小割的求解

最小割是最大流的对偶问题。最小割的求解通常有以下两种情形。

(1) 求最小割的容量，根据最大流最小割定理，可以转换成求解网络最大流流量。

(2) 如果还要进一步求出最小割由哪些边组成，或者要求出最小割将顶点集划分成哪两个子集，这些问题要按如下思路求解。

根据网络最大流的求解思路，当在残留网络中从源点 V_s 出发无法遍历汇点时(或者说汇点不在层次网络中)，所求得的网络流就是最大流。此时，从源点 V_s 能遍历到的顶点就构成最小割(S, T)中的顶点集合 S，其余顶点构成顶点集合 T。因此，求最小割的步骤如下。

(1) 先求得网络最大流。

(2) 在残留网络 G'中，从源点 V_s 出发进行深度优先搜索，遍历到的顶点构成集合 S，其余顶点构成顶点集合 T，连接 S 和 T 的所有弧构成容量网络的一个最小割(S, T)。

例如，在图 6.26 中，求得网络最大流后，从源点 V_s 出发只能遍历到顶点 V_s 和 V_1，因此 $S = \{ V_s, V_1 \}$，如图 6.26(c)所示，其余顶点构成集合 T。连接 S 和 T 的所有弧，在图(d)中用粗线标明，构成容量网络的一个最小割(S, T)，其容量为 8，等于网络最大流量。

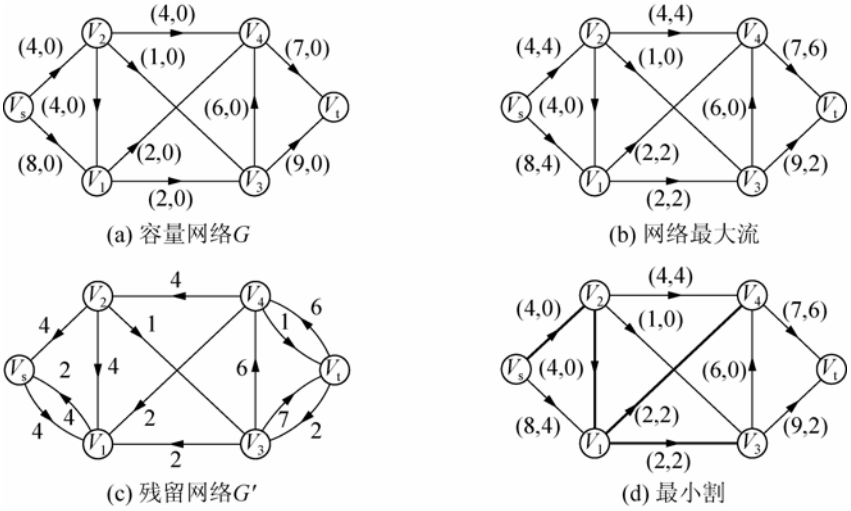


图 6.26　最小割的求解

最小割(S, T)中所有的前向弧在网络最大流中一定是饱和弧，但饱和弧不一定就是最小割中的弧，例如图 6.26(d)中，弧$<V_2, V_4>$是饱和弧，但不是最小割中的弧。

下面通过 3 道 ACM/ICPC 例题，再详细讲解容量网络最小割的求解思路和程序实现。

例 6.6 双核 CPU(Dual Core CPU)

题目来源：

POJ Monthly—2007.11.25, POJ3469

题目描述：

由于越来越多的计算机配置了双核 CPU，TinySoft 公司的首席技术官员 SetagLilb 决定升级他们的产品—SWODNIW。

SWODNIW 包含了 N 个模块，每个模块必须运行在某个 CPU 中。每个模块在每个 CPU 中运行的耗费已经被估算出来了，设为 A_i 和 B_i。同时，M 对模块之间需要共享数据，如果它们运行在同一个 CPU 中，共享数据的耗费可以忽略不计，否则，还需要额外的费用。必须很好地安排这 N 个模块，使得总耗费最小。

输入描述：

测试数据的第 1 行为两个整数 N 和 M，$1 \leqslant N \leqslant 20\,000$，$1 \leqslant M \leqslant 200\,000$。接下来有 N 行，每行为两个整数 A_i 和 B_i。接下来有 M 行，每行为 3 个整数 a, b, w，表示 a 模块和 b 模块如果不是在同一个 CPU 中运行，则需要花费额外的 w 耗费来共享数据。

输出描述：

输出一个整数，为最小耗费。

样例输入：

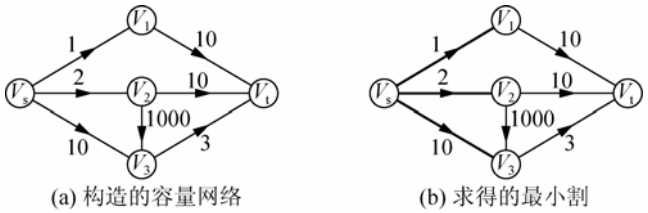
```
3 1
1 10
2 10
10 3
2 3 1000
```

样例输出：

```
13
```

分析：

如果将两个 CPU 分别视为源点和汇点、模块视为顶点，则可以按照以下方式构图：对于第 i 个模块在每个 CPU 中的耗费 A_i 和 B_i，从源点向顶点 i 连接一条容量为 A_i 的弧、从顶点 i 向汇点连接一条容量为 B_i 的弧；对于 a 模块与 b 模块在不同 CPU 中运行造成的额外耗费 w，从顶点 a 向顶点 b 连接一条容量为 w 的弧。此时每个顶点(模块)都和源点及汇点(两个 CPU)相连，即每个模块都可以在任意一个 CPU 中运行。例如，对样例输入数据构造的容量网络如图 6.27(a)所示。



(a) 构造的容量网络　　(b) 求得的最小割

图 6.27　双核 CPU

不难了解到，对于图中的任意一个割，源点与汇点必不连通。因此每个顶点(模块)都

不可能同时和源点及汇点(两个 CPU)相连，即每个模块只在同一个 CPU 中运行。此时耗费即为割的容量。很显然，当割的容量取得最小值时，总耗费最小。故题目转化为求最小割的容量。例如，在图 6.27 中，求得的最小割为图 6.27(b)；在图 6.27(b)中，最小割中的弧用粗线标明，其容量为 13。对于最小割的容量，根据最大流最小割定理，可以通过最大流流量来求解。

下面的代码通过 Dinic 算法来求解网络最大流。

代码如下：

```
#define INT_MAX 10000000
#define NMAX 21000
#define MMAX 1000000
struct EDGE
{
    int u, v, cap, flow;
    int next;
    EDGE( int a=0, int b=0, int c=0, int d=0 )
        : u( a ), v( b ), cap( c ), flow( d ) {}
};
struct EDGELIST
{
    int start[NMAX];
    int last[NMAX];
    int t, i;
    EDGE arc[MMAX];
    void clear( )              //清除操作
    {
        t=0;
        for( i=0; i<NMAX; i++ )  last[i]=-1;
    }
    void push_back( EDGE edge )//追加操作
    {
        edge.next=-1;  arc[t]=edge;
        if ( last[edge.u]!=-1 ) arc[ last[edge.u] ].next=t;
        else start[edge.u]=t;
        last[edge.u]=t;  t++;
    }
    void add_arc( EDGE edge )  //创建双向弧
    {
        push_back( edge );  push_back( EDGE( edge.v,edge.u,edge.cap ) );
    }
}net;
int q[2][NMAX];                //数组模拟滚动队列
int q1[2],q2[2],qnow;
void push_queue( int a )       //入队列
{
    q[qnow][q2[qnow]++]=a;
```

```
}
int pop_queue()                 //出队列
{
    return q[qnow^1][q1[qnow^1]++];
}
void switch_queue( )    //滚动队列
{
    qnow^=1;
    q1[qnow]=0, q2[qnow]=0;
}
bool empty_queue( )     //判断队列是否为空
{
    return q1[qnow^1]>=q2[qnow^1];
}
int size_queue( )       //队列大小
{
    return q2[qnow^1]-q1[qnow^1];
}
int n, m;
int dis[NMAX];          //层次网络(距离标号)
int path[NMAX], deep;   //路径
int cur[NMAX];
bool BFS( )             //BFS 构建层次网络
{
    int i, l, u, v;
    for( i=0; i<NMAX; i++ )  dis[i]=-1;//初始化
    dis[0]=0;  qnow=0;
    switch_queue();
    push_queue( 0 );
    switch_queue();
    while ( !empty_queue() )            //队列不为空
    {
        l=size_queue();
        while( l-- )
        {
            u=pop_queue();          //取出队列结点
            for ( i=net.start[u]; i!=-1; i=net.arc[i].next )
            {
                v=net.arc[i].v;
                if ( dis[v]==-1 && net.arc[i].cap > net.arc[i].flow )
                {
                    push_queue( v );
                    dis[v]=dis[u]+1;
                    if ( v==n ) return true; //汇点在层次网络中
                }
            }
        }
```

```
            switch_queue();     //滚动队列
        }
        return false;           //汇点不在层次网络中
    }
    int Dinic( )                //Dinic 算法求最大流
    {
        int i, u, neck, pos, res;
        int maxflow=0;
        while ( BFS() )
        {
            memcpy( cur, net.start, sizeof( cur ) );//初始化
            deep=0, u=0;
            while( true )       //最短路径增广
            {
                if ( u==n )     //存在增广路则修改残留网络
                {
                    neck=INT_MAX;
                    for ( i=0; i<deep; i++ )
                    {
                        res=net.arc[path[i]].cap-net.arc[path[i]].flow;
                        if ( res<neck )
                        {
                            neck=res;  pos=i;
                        }
                    }
                    maxflow+=neck;
                    for( i=0; i<deep; i++ )
                    {
                        net.arc[path[i]].flow+=neck;
                        net.arc[path[i]^1].flow-=neck;
                    }
                    deep=pos;
                    u=net.arc[path[deep]].u;
                }
                //在层次网络中进行增广
                for( i=cur[u]; i!=-1; i=net.arc[i].next )
                {
                    if( net.arc[i].cap>net.arc[i].flow && dis[u]+1==
                        dis[net.arc[i].v] )
                        break;
                }
                cur[u]=i;
                if( i!=-1 )
                {
                    path[deep++]=i;  u=net.arc[i].v;
                }
                else
```

```
            {
                if( deep==0 ) break;
                dis[u]=-1;
                u=net.arc[path[--deep]].u;
            }
        }
    }
    return maxflow;             //返回最大流
}
int main( )
{
    int i, a, b, w;
    scanf( "%d %d", &n, &m );
    net.clear( );               //初始化
    for( i=1; i<=n; i++ )       //构建网络
    {
        scanf( "%d %d", &a, &b );
        net.add_arc( EDGE( 0,i,a ) );  net.add_arc( EDGE( i,n+1,b ) );
    }
    for( i=0; i<m; i++ )
    {
        scanf( "%d %d %d", &a, &b, &w );
        net.add_arc( EDGE( a,b,w ) );
    }
    n++;
    printf( "%d\n", Dinic() );
    return 0;
}
```

例 6.7　伞兵(Paratroopers)

题目来源：

AUT Contest 1, ZOJ2874, POJ3308

题目描述：

公元 2500 年，地球和火星之间爆发了一场战争。最近，地球军队指挥官获悉火星入侵者将派一些伞兵来摧毁地球的兵工厂，兵工厂是一个 $m \times n$ 大小的网格。他还获悉每个伞兵将着陆的具体位置(行和列)。由于火星的伞兵个个都很强壮，而且组织性强，只要有一个伞兵存活了，就能摧毁整个兵工厂。因此，地球军队必须在伞兵着陆后瞬间全部杀死他们。

为了完成这个任务，地球军队需要利用高科技激光枪。他们能在某行(或某列)安装一架激光枪，一架激光枪能杀死该行(或该列)所有的伞兵。在第 i 行安装一架激光枪的费用是 R_i，在第 i 列安装的费用是 C_i。要安装整个激光枪系统，以便能同时开火，总的费用为这些激光枪费用的乘积。现在，试选择能杀死所有伞兵的激光枪，并使得整个系统的费用最小。

输入描述：

输入文件的第 1 行为整数 T，表示测试数据的数目，接下来有 T 个测试数据。每个测试数据的第 1 行为 3 个整数 m、n 和 L，$1 \leqslant m \leqslant 50$，$1 \leqslant n \leqslant 50$，$1 \leqslant L \leqslant 50$，分别表示网格的行和列以及伞兵的数目；接下来一行为 m 个大于或等于 1.0 的实数，第 i 个实数表示 R_i；

再接下来一行为 n 个大于或等于 1.0 的实数，第 i 个实数表示 C_i；最后 L 行，每行为两个整数，描述了每个伞兵的着陆位置。

输出描述：

对每个测试数据，输出搭建整个激光枪系统的最小费用，精确到小数点后面 4 位有效数字。

样例输入：

```
1
4 4 5
2.0 7.0 5.0 2.0
1.5 2.0 2.0 8.0
1 1
2 2
3 3
4 4
1 4
```

样例输出：

```
16.0000
```

分析：

对于本题，如果把伞兵视为边、行与列视为顶点，则可以通过以下方式来构图：增加两个顶点—源点与汇点；对于第 i 行，从源点向顶点 R_i 连接一条容量为在第 i 行安装激光枪费用的弧；对于第 j 列，从顶点 C_j 向汇点连接一条容量为在第 j 列安装激光枪费用的弧。如果某一点(i, j)有伞兵降落，则从顶点 R_i 向顶点 C_j 连接一条容量为无穷大的弧。例如，对样例输入数据构造的容量网络如图 6.28 所示。

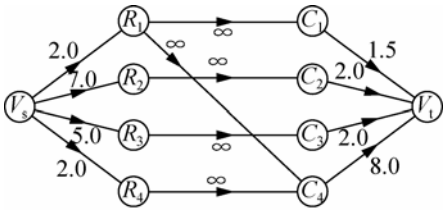


图 6.28　伞兵：容量网络的构造

根据割的性质，源点与汇点必不连通，因此割边集中必定存在 $S\rightarrow R$、$R\rightarrow C$、$C\rightarrow T$ 其一。为了求得最小容量，将 $R\rightarrow C$ 的容量设为无穷大，则其不可能被选中。这样割边集为 $S\rightarrow R$ 与 $C\rightarrow T$ 的集合，也就是选中了行或列。此时求得的最小割即为花费最小的方案。

需要注意的是，花费为行与列的乘积，因此在算法中可以先通过对数运算来把乘法转换为加法。

代码如下：

```
#define MAX 110
#define INF 10000000
struct Node
{
    double c, f;
}map[MAX][MAX];
int pre[MAX];          //pre[i]为增广路径顶点 i 前一个顶点的序号
int queue[MAX];        //数组模拟队列
```

```
int s, t;               //源点;汇点
bool BFS( )             //BFS 求增广路
{
    int i, cur, qs, qe; //队列当前结点;队列头;队列尾
    memset( pre, -1, sizeof(pre) );
    pre[s]=s;
    qs=0; qe=1;
    queue[qs]=s;
    while( qs<qe )
    {
        cur=queue[qs++];
        for( i=0; i<=t; i++ )
        {
            if( pre[i]==-1 && map[cur][i].c-map[cur][i].f>0 )
            {
                queue[qe++]=i;
                pre[i]=cur;
                if( i==t )  return 1;   //汇点在层次网络中
            }
        }
    }
    return 0;       //汇点不在层次网络中
}
double maxflow( )  //求最大流
{
    double max_flow=0, min;
    int i;
    while( BFS( ) )
    {
        min=INF;
        for( i=t; i!=s; i=pre[i] ) //调整网络
        {
            if( map[pre[i]][i].c-map[pre[i]][i].f<min )
                min=map[pre[i]][i].c-map[pre[i]][i].f;
        }
        for( i=t; i!=s; i=pre[i] )
        {
            map[pre[i]][i].f+=min;  map[i][pre[i]].f-=min;
        }
        max_flow+=min;
    }
    return max_flow;    //返回最大流
}
int main( )
{
    int i, j, n, m, l, r, cc, w;
    double c;
    scanf( "%d", &w );
    while( w-- )
    {
```

```
            memset( map, 0, sizeof(map) );
            scanf( "%d %d %d", &n, &m, &l );
            s=0; t=n+m+1;
            //构建网络;用对数运算来将乘法转换为加法
            for( i=1; i<=n; i++ )
            {
                scanf( "%lf", &c );  map[s][i].c=log(c);
            }
            for( i=1; i <= m; i++ )
            {
                scanf( "%lf", &c );  map[i+n][t].c=log(c);
            }
            for( i=1; i<=l; i++ )
            {
                scanf( "%d %d", &r, &cc );  map[r][n+cc].c=10000000;
            }
            printf( "%.4lf\n", exp( maxflow() ) ); //输出时将对数值转换为原值
        }
        return 0;
    }
```

例 6.8　友谊(Friendship)

题目来源：

POJ Monthly, POJ1815

题目描述：

在现代社会，每个人都有自己的朋友。由于每个人都很忙，他们只通过电话联系。你可以假定 A 可以和 B 保持联系，当且仅当：① A 知道 B 的电话号码；② A 知道 C 的号码，而 C 能联系上 B。如果 A 知道 B 的电话号码，则 B 也知道 A 的电话号码。

有时，有人可能会碰到比较糟糕的事情，导致他与其他人失去联系。例如，他可能会丢失了电话簿，或者换了电话号码。

在本题中，告知 N 个人之间的两两联系，这 N 个人的编号为 1～N。给定两个人，比如 S 和 T，如果有些人碰到糟糕的事情，S 可能会与 T 失去联系。计算至少多少人碰到糟糕的事情，会导致 S 与 T 失去联系。假定 S 和 T 不会碰到糟糕的事情。

输入描述：

测试数据的第 1 行为 3 个整数 N、S 和 T，$2 \leqslant N \leqslant 200$，$1 \leqslant S$、$T \leqslant N$，$S$ 不等于 T。接下来有 N 行，每行有 N 个整数，如果 i 知道 j 的电话号码，则第 i+1 行、第 j 列上的数字为 1，否则为 0。假定这 N 行中 1 的数目不超过 5 000。

输出描述：

如果无法使 A 与 B 失去联系，输出"NO ANSWER!"；否则输出的第 1 行为整数 t，表示至少需要 t 个人碰到糟糕的事情，才能导致 A 与 B 失去联系；如果 t 不为 0，则要输出第 2 行，包含 t 个整数，按升序输出这 t 个人的编号，这些整数用空格隔开。

如果存在多个解，则为每个解定义一个分值，输出具有最小分值的解。分值计算方式为：假定解为 A_1，A_2，…，A_t，$1 \leqslant A_1 < A_2 < \cdots < A_t \leqslant N$，则分值为$(A_1-1)\times N$ ^t+$(A_2-1)\times N$^(t–1)+…+$(A_t-1)\times N$。测试数据保证不会出现两个解都具有相同的最小分值。

样例输入：

```
3 1 3
1 1 0
1 1 1
0 1 1
9 1 9
1 1 1 0 0 0 0 0 0
1 1 1 1 1 0 0 0 0
1 1 1 0 1 1 0 0 0
0 1 0 1 0 0 1 0 0
0 1 1 0 1 0 1 1 0
0 0 1 0 0 1 0 1 0
0 0 0 1 1 0 1 1 1
0 0 0 0 1 1 1 1 1
0 0 0 0 0 0 1 1 1
```

样例输出：

```
1
2
2
2 3
```

分析：

本题要求解的是在一个给定的无向图中至少应该去掉几个顶点才能使得 S 和 T 不连通。样例输入中两个测试数据所描绘的无向图如图 6.29 所示。在图 6.29(a)中，至少应该去掉顶点 2，才能使顶点 1 和 3 不连通。在图 6.29(b)中，至少应该去掉顶点 2 和 3 才能使得顶点 1 和 9 不连通(去掉顶点 7 和 8 也可以，但前面的解是题目中所要求的最小分值的解)。

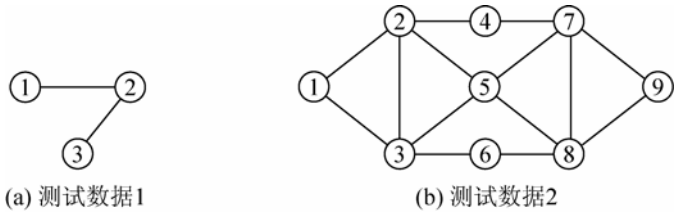


图 6.29　友谊：测试数据

如果把问题扩展一些，对任意一对不相邻顶点，至少应该去掉几个顶点，才能使它们不连通，这是求顶点连通度的问题，详见 8.3.3 节。本题可以采取 8.3.3 节中构造容量网络 N 的方法，具体如下。

(1) 原图中的每个顶点 v 变成网络 N 中的两个顶点 v'和 v''，顶点 v'到 v''有一条弧连接，即<v', v''>，其容量为 1。

(2) 原图 G 中的每条边 $e = (u, v)$，在网络 N 中有两条弧 $e' = <u'', v' >$和 $e'' = <u', v''>$，e'和 e''的容量均为∞。

(3) 令 S'为源点，T''为汇点。

构造好容量网络 N 后，求从 S'到 N''的最大流 F。在最大流中，流出 S'的一切弧的流量和，即为所求的至少要删除的顶点数。

下面的代码首先通过 Dinic 算法来求得最大流(即最小割)，然后枚举顶点，求得使无向图不连通的顶点。枚举时要注意按照从小到大的顺序，这样可以按照字典序得到顶点，即题目要求的分值最小的解。

代码如下：

```
#define INF 1000000
struct NODE
{
    int w, f;
};
NODE net[500][500];
int set[500];           //最小割点集
int N, S, T;
bool flag[500];         //标志数组
int q[500], d[500];     //队列;层次网络(距离标号)
int s, t;               //源点;汇点
int min( int a, int b )
{
    return a<b?a:b;
}
bool BFS( )             //构建层次网络
{
    int head, tail, u, v;
    head=0, tail=0;     //初始化
    memset( flag, 0, sizeof(flag) );
    q[tail++]=t, d[t]=0, flag[t]=1;
    while( head<tail )
    {
        u=q[head++];
        for( v=0; v<=t; v++ )
        {
            if( !flag[v] && net[v][u].w>net[v][u].f )
            {
                d[v]=d[u]+1;
                q[tail++]=v;
                flag[v]=1;
            }
            if( flag[s] )  return 1;
        }
    }
    return 0;
}
int DFS( int v, int low )  //DFS 增广
{
    int i;
    if( v==t )  return low;
    int flow;
    for( i=0; i<=t; i++ )
    {
        if( net[v][i].w>net[v][i].f && d[v]==d[i]+1 )
```

```
            {
                if( flow=DFS( i,min(low,net[v][i].w-net[v][i].f) ) ) //递归调用
                {
                    net[v][i].f+=flow;  //修改流量
                    net[i][v].f=-net[v][i].f;
                    return flow;
                }
            }
        }
        return 0;
    }
    void Add_Edge( int a, int b, int c )
    {
        net[a][b].w=c;
    }
    void Dinic( )
    {
        int ans=0, c, k, i, a, b, cnt, temp;
        while( BFS( ) ) //求最大流
        {
            int flow;
            while( flow=DFS(s,INF) )
                ans+=flow;
        }
        printf( "%d\n", ans );
        //开始枚举
        if( ans==0 )  return;
        cnt=0;  temp=ans;
        for( i=1; i<=N && temp; i++ )
        {
            if( i==S || i==T )  continue;
            if( !net[i][i+N].f )  continue;
            net[i][i+N].w=0;
            for( a=1; a<=t; a++ )
            {
                for( b=1; b<=t; b++ )
                    net[a][b].f=0;
            }
            k=0;
            while( BFS( ) )
            {
                int flow;
                while( flow=DFS(S,INF) )
                k+=flow;
            }
            if( k!=temp )
            {
```

```
                set[cnt++]=i;  //将顶点存入最小割点集
                temp=k;
            }
            else net[i][i + N].w=1;
        }
        for( c=0; c<ans-1; c++ )  printf( "%d ", set[c] );  //输出
        printf( "%d\n", set[c] );
    }
    int main( )
    {
        int tail, i, j;
        scanf( "%d%d%d", &N, &S, &T );
        memset( net, 0, sizeof(net) ); //初始化
        s=0;  t=2*N+1;
        Add_Edge( s, S, INF );
        Add_Edge( T+N, t, INF );
        for( i=1; i<=N; i++ )
        {
            Add_Edge( i, i+N, 1 );
            for( j=1; j<=N; j++ )
            {
                scanf( "%d", &tail );
                if( tail )  Add_Edge( i+N, j, INF );
            }
        }
        Add_Edge( S, S+N, INF );   Add_Edge( T, T+N, INF );
        if( !net[S+N][T].w )  Dinic( );
        else  printf( "NO ANSWER!\n" );  //源点汇点相连时直接输出
        return 0;
    }
```

练　习

6.5　唯一的攻击(Unique Attack), ZOJ2587

题目描述：

N 台超级计算机连成一个网络。M 对计算机之间用光纤直接连在一起，光纤的连接是双向的。数据可以直接在有光纤直接连接的计算机之间传输，也可以通过一些计算机作为中转来传输。

有一群恐怖分子计划攻击网络。他们的目标是将网络中两台主计算机断开，这样这两台计算机之间就无法传输数据了。恐怖分子已经计算好了摧毁每条光纤所需要花费的钱。当然了，他们希望攻击的费用最少，因此就必须使得需要摧毁的光纤费用总和最少。

现在，恐怖分子的头头想知道要达到目标且费用最少，是否只有一种方案。

输入描述：

输入文件包含多个测试数据。每个测试数据的格式为：第 1 行为 4 个整数 N、M、A 和 B，$2 \leqslant N \leqslant 800$，$1 \leqslant M \leqslant 10\,000$，$1 \leqslant A$、$B \leqslant N$，$A \neq B$，$N$ 表示网络中计算机的数目，M

表示直接连接计算机的光纤数目，*A* 和 *B* 表示两台主计算机的序号；接下来有 *M* 行，描述了每条光纤的连接情况，分别为所连接的两台计算机的序号和摧毁它所需的费用，费用非负且不超过 105，任何两台计算机之间最多只有一根光纤连接，任何光纤都不会连着同一台计算机，初始时，两台主计算机是连通的。

输入文件最后一行为 4 个 0，表示输入结束。

输出描述：

对每个测试数据，如果只有一种方案来完成攻击，输出"UNIQUE"，否则输出"AMBIGUOUS"。

样例输入：

```
4 4 1 2
1 2 1
2 4 2
1 3 2
3 4 1
0 0 0 0
```

样例输出：

```
UNIQUE
```

6.6 让人恐慌的房间(Panic Room), ZOJ2788

题目描述：

你是电子防护系统 9042 的首席程序员，这套软件是 Jellern 公司最新、最好的家用安全软件。这套软件被设计用来保护一个房间。这套软件可以计算为了阻止从其他房间进入被保护的房间至少需要锁上几扇门。每扇门连接两个房间，只有一个控制面板，通过控制面板可以开启门。这个控制面板只能从门的某一面才能打开。例如，如果房子的布局如图 6.30 所示，房间编号为 0～6，标有"CP"的一面表示控制面板所在的一面(从这一面所在的房间可以打开这扇门)。在图中，为了阻止从房间 1 进入房间 2，至少需要锁上 2 扇门，即：房间 2 和房间 1 之间的门、房间 3 和房间 1 之间的门。注意，不能阻止从房间 3 进入房间 2，因为总是可以在房间 3 通过控制面板打开房间 3 和房间 2 之间的门。

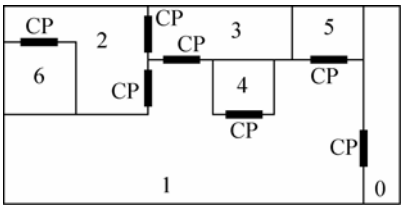


图 6.30 引起恐慌的房间

输入描述：

输入文件的第 1 行为一个整数 *T*，表示测试数据数目。每个测试数据包含以下两个部分。

(1) 开始行：占一行，为两个整数 *m* 和 *n*，1≤*m*≤20，0≤*n*≤19，*m* 表示房子里的房间数目，*n* 表示需要保护的房间号。

(2) 房间列表：共 *m* 行，第 *i* 行描述第 *i* 个房间是否有入侵者，"I"表示有入侵者，"NI"表示没有；然后是整数从，0≤*c*≤20，表示第 *i* 个房间有 *c* 扇门与某些房间相连，这些门的控制面板都是在第 *i* 个房间；最后是 *c* 个整数，为 *c* 扇门所连接的房间的编号(按升序列出)；*i* 为 0～*m*−1。某个房间可能有多扇门，这样，如果该房间有入侵者就将会有多个。

输出描述：

对每个测试数据，输出为了阻止所有的入侵者进入被保护的房间，至少需要锁上多少扇门。如果无法阻止所有的入侵者进入被保护的房间，则输出"PANIC ROOM BREACH"。假定初始时所有的门都是开着的，被保护的门没有入侵者。

样例输入：

```
1
7 2
NI 0
I 3 0 4 5
NI 2 1 6
NI 2 1 2
NI 0
NI 0
NI 0
```

样例输出：

```
2
```

6.7 项目发展规划(Develop)

题目描述：

某公司准备制定一份未来的发展规划。公司各部门提出的发展项目汇总成了一张规划表，该表包含了许多项目。对于每个项目，规划表中都给出了它所需的投资或预计的盈利。由于某些项目的实施必须依赖于其他项目的开发成果，所以如果要实施这个项目的话，它所依赖的项目也是必不可少的。现在请你担任该公司的总裁，从这些项目中挑选出一部分，使你的公司获得最大的净利润。

输入描述：

输入文件包括项目的数量 N，每个项目的预算 C_i 和它所依赖的项目集合 P_i。格式如下。

第 1 行是 N，$0 \leqslant N \leqslant 1\,000$；

接下来有 N 行，其中第 i 行每行表示第 i 个项目的信息。每行的第 1 个数是 C_i，$-1\,000\,000 \leqslant C_i \leqslant 1\,000\,000$，正数表示盈利，负数表示投资。剩下的数是项目 i 所依赖的项目的编号。

每行相邻的两个数之间用一个或多个空格隔开。

输出描述：

第 1 行是公司的最大净利润。接着是获得最大净利润的项目选择方案。若有多个方案，则输出挑选项目最少的一个方案。每行一个数，表示选择的项目的编号，所有项目按从小到大的顺序输出。

样例输入：

```
6
-4
1
2 2
-1 1 2
-3 3
5 3 4
```

样例输出：

```
3
1
2
3
4
6
```

6.3 流量有上下界的网络的最大流和最小流

6.3.1 流量有上下界的容量网络

1. 问题的引入

6.1 节讨论了容量网络的最大流问题。这种网络的每一条弧<u, v>都对应一个弧容量 $c(u, v)\geqslant 0$。本节讨论的网络，每条弧对应两个权值 $b(u, v)$和 $c(u, v)$，分别表示弧流量的下界和上界。如何求这一类网络的最大流(或最小流)?

很显然，6.1 节讨论的网络结构是流量有上下界网络的一种特例，即 $b(u, v) = 0$。

当 $b(u, v)>0$ 时，这种有上下界的网络不一定存在可行流。例如，图 6.31 所示的容量网络，弧上的第 1 个数字为 $b(u, v)$的值，第 2 个数字为 $c(u, v)$的值。由于 $b_{21} + b_{2t} > c_{s2}$，也就是说，弧<V_s, V_2>能提供的最大流量，小于弧<V_2, V_1>和<V_2, V_t>流量最小值之和。因此，在图 6.31 中，不存在可行流。

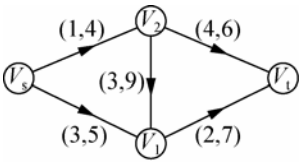


图 6.31 流量有上下界的网络不存在可行流的例子

所以流量有上下界的容量网络，首先要解决的问题是判断是否存在可行流。其数学模型为：在容量网络 $G(V, E)$中，每条弧<u, v>有两个权值 $b(u, v)$和 $c(u, v)$。满足以下条件的网络流 f 称为可行流。

(1) 弧流量限制条件：

$$b(u, v) \leqslant f(u, v) \leqslant c(u, v),\ \forall <u, v> \in E。 \tag{6-10}$$

(2) 平衡条件：

$$\sum_v f(u,v) - \sum_w f(w,u) = \begin{cases} |f| & 当 u = V_s \\ 0 & 当 u \neq V_s, V_t \\ -|f| & 当 u = V_t \end{cases}。 \tag{6-11}$$

式中：$\sum_v f(u,v)$ 表示从顶点 u 流出的流量总和；$\sum_w f(w,u)$ 表示流入顶点 u 的流量总和；$|f|$ 为该可行流的**流量**，即源点的净流出流量，或汇点的净流入流量。

2. 流量有上下界的容量网络的可行流的求解

为了利用标号法求流量有上下界容量网络的可行流，必须设法去掉每条弧上的流量下界。设顶点 u 为容量网络中除源点和汇点外的普通顶点，它满足平衡条件：

$$\sum_v f(u,v) = \sum_w f(w,u) \tag{6-12}$$

因为可行流上每条弧<u, v>上至少有 $b(u, v)$的流量，不妨设弧<u, v>上的流量 $f(u, v)$为：

$$f(u, v) = b(u, v) + f_1(u, v) \tag{6-13}$$

代入(6-12)式中，得：

$$\sum_{v}(b(u,v)+f_1(u,v))=\sum_{w}(b(w,u)+f_1(w,u)) \tag{6-14}$$

移项得：

$$\sum_{v}b(u,v)-\sum_{w}b(w,u)=\sum_{w}f_1(w,u)-\sum_{v}f_1(u,v) \tag{6-15}$$

注意(6-15)式的左边是只和下界 $b(u, v)$有关的常量，可以看成顶点 u 的属性，因此定义：

$$D(u)=\sum_{v}b(u,v)-\sum_{w}b(w,u) \tag{6-16}$$

式中：$D(u)$为顶点 u 发出的所有弧的流量下界和与进入顶点 u 的所有弧的流量下界和之差。

因为容量网络 G 上的可行流必须满足每条弧的流量至少达到下界，先考虑这样一个伪流 f_0(不满足流量平衡条件)：

$$f_0(u, v)=b(u, v)，\ \forall<u, v>\in E \tag{6-17}$$

虽然 f_0 不满足流量平衡条件，但 f_0 与(6-13)式中的伪流 f_1 叠加后的网络流 f 满足流量平衡条件，即式(6-12)。

由流量限制条件：

$$b(u, v)\leqslant f_0(u, v)+f_1(u, v)\leqslant c(u, v) \tag{6-18}$$

在式(6-18)中每项都减去 $b(u, v)$(即 $f_0(u, v)$)后得：

$$0\leqslant f_1(u, v)\leqslant c(u, v)-b(u, v) \tag{6-19}$$

这样在伪流 f_1 中成功地去掉了每条弧的流量下界。并且如果能求出伪流 f_1，那么原网络中的可行流 $f(u, v)=b(u, v)+f_1(u, v)$也能求出。

因为伪流 f_1 也不满足流量平衡条件，所以必须构造一个**伴随网络** $\overline{G}$ (Accompany Network)来求解伪流 f_1，该伴随网络必须满足以下条件。

(1) 伴随网络 $\overline{G}$ 与原网络 G 同构，可以增设顶点。

(2) 伴随网络中所有弧的流量只有上界(也称为弧的容量)，而没有下界，或者说弧流量下界为 0。

(3) 原网络的伪流 f_1 包含在伴随网络的最大流 $\overline{f}$ 中。

这样就可以采用标号法求伴随网络的最大流 $\overline{f}$，从而求出原网络的伪流 f_1 及可行流 f。

构造原网络的伴随网络 $\overline{G}$ 的方法如下。

(1) 新增两个顶点 $\overline{V}_s$ 和 $\overline{V}_t$，$\overline{V}_s$ 称为附加源点，$\overline{V}_t$ 称为附加汇点。

(2) 对原网络 G 中每个顶点 V_i，加一条新弧 $<V_i, \overline{V}_t>$，这条弧的容量为顶点 V_i 发出的所有弧的流量下界之和。

(3) 对原网络 G 中每个顶点 V_i，加一条新弧 $<\overline{V}_s, V_i>$，这条弧的容量为进入到顶点 V_i 的所有弧的流量下界之和。

(4) 原网络 G 中的每条弧<u, v>，在伴随网络 $\overline{D}$ 中仍保留，但弧的容量 $\overline{c}(u,v)$ 修正为：$c(u, v)-b(u, v)$。

(5) 再添两条新弧<V_s, V_t>和<V_t, V_s>，流量上界均为∞。

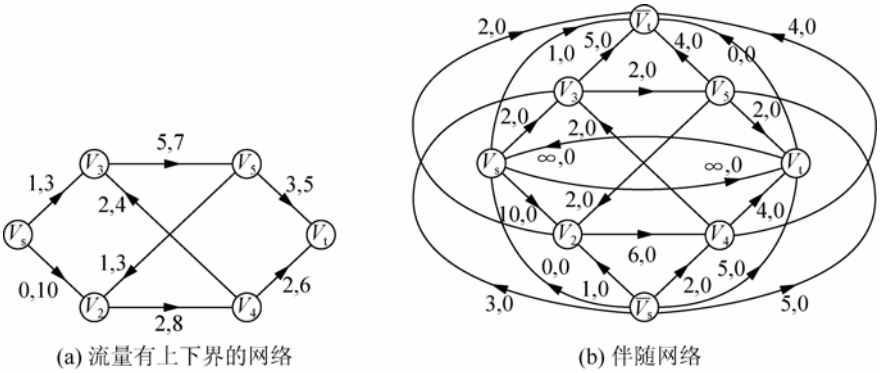
伴随网络 $\overline{G}$ 构造好以后，按照 6.1.4 节介绍的标号法求伴随网络中从附加源点 $\overline{V}_s$ 到附加汇点 $\overline{V}_t$ 的最大流。如果求得的最大流中，附加源点 $\overline{V}_s$ 流出的所有弧均满载，即 $\overline{V}_s$ 发出的

所有弧 e，均满足 $\overline{f}(e)=\overline{c}(e)$，则原网络 G 存在可行流：$f(e)=\overline{f}(e)+b(e)$，即在该可行流中，每条弧上的流量为伴随网络最大流对应弧上的流量加上该弧的流量下界。如果伴随网络的最大流中，附加源点 $\overline{V_s}$ 发出的某条弧未满载，则原网络 G 不存在可行流。

需要说明的是，如果附加源点 $\overline{V_s}$ 流出的所有弧均满载，则流入到附加汇点 $\overline{V_t}$ 的所有弧肯定也满载。

为什么要求解伴随网络的最大流并且要求附加 $\overline{V_s}$ 流出的所有弧均满载呢？因为附加源点发出的弧 $<\overline{V_s},V_i>$ 的容量是进入到顶点 V_i 的所有弧的流量下界之和，要使得原网络所有弧的流量都大于下界，则附加 $\overline{V_s}$ 流出的所有弧流量都取最大值。

例如，对图 6.32(a)所示的容量网络，构造好的伴随网络如图 6.32(b)所示。在图 6.32(b)中，每条弧上第 1 个数字代表这条弧的容量，第 2 个数字代表弧的流量。



(a) 流量有上下界的网络　　(b) 伴随网络

图 6.32　有上下界的流量网络及其伴随网络

在图 6.32(b)中，需要说明的是构造伴随网络中的第(2)和(3)步：在第(2)步中，因为在原网络 G 中不存在汇点 V_t 发出的弧，所以相应新增的弧容量为 0；同样，在第(3)步中，因为进入到源点 V_s 的弧不存在，所以相应新增的弧容量也为 0。

对 6.32(b)所示的伴随网络，求得的最大流如图 6.33 所示。从图 6.33 可以看出，附加源点 $\overline{V_s}$ 流出的所有弧均满载，所以原网络 G 存在可行流，可行流如图 6.34(a)所示，其流量为 5。在图 6.34(a)中，每条弧上的第 1、2、3 个数字分别表示弧流量的下界 $b(e)$、上界 $c(e)$ 及实际流量 $f(e)$。

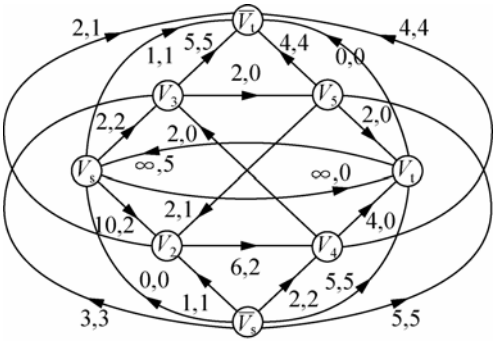


图 6.33　伴随网络的网络最大流

6.3.2　流量有上下界的网络的最大流

1. 数学模型

流量有上下界的网络最大流的数学模型为：在容量网络 $G(V, E)$中，每条弧$<u, v>$有两个权值 $b(u, v)$和 $c(u, v)$，分别表示通过该弧的流量的下界和上界；求满足以下条件的所有可行流 f 中流量最大的可行流。

(1) 弧流量限制条件：$b(u, v) \leqslant f(u, v) \leqslant c(u, v)$，$\forall <u, v> \in E$。

(2) 平衡条件：

$$\sum_{v} f(u,v) - \sum_{w} f(w,u) = \begin{cases} |f| & \text{当} u = V_s \\ 0 & \text{当} u \neq V_s, V_t \\ -|f| & \text{当} u = V_t \end{cases}$$

2. 流量有上下界的网络最大流算法

求有上下界网络最大流的算法是：先按照 6.3.1 节的方法求可行流，如果可行流不存在，则算法结束；否则从可行流出发，按照 6.1.4 节介绍的标号法(或其他算法)把可行流放大，从而求得最大流。但是在放大可行流的过程中，因为增广路上的反向弧的流量会减少，所以在选择可改进量 α 时要保证反向弧流量减少后不低于流量下限 $B(e)$。即可改进量 α 的取值为：

$$\alpha = \min\{\min_{P+}\{c(e) - f(e)\}, \min_{P-}\{f(e) - b(e)\}\} \text{。} \tag{6-20}$$

例如，在图 6.34(a)所示可行流的基础上，求得的最大流如图 6.34(b)所示，从图中可以看出，最大流的流量为 10。

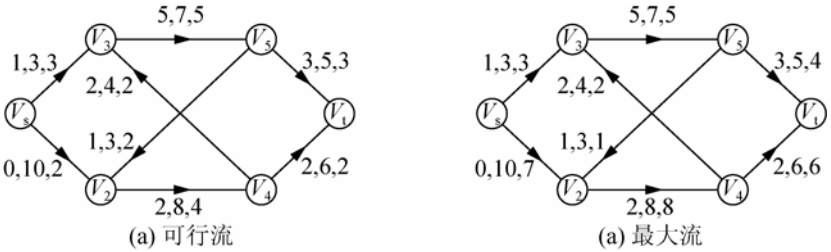


图 6.34　有上下界网络的可行流及最大流

3. 算法实现

把例 6.1 的程序稍作修改，即可实现有上下界网络的最大流：在构造好伴随网络后，利用标号法将可行流放大，在放大可行流时注意应按前面的公式选择可改进量 α，实现方法详见例 6.9。

6.3.3　流量有上下界的网络的最小流

1. 数学模型

流量有上下界的网络最小流的数学模型为：在容量网络 $G(V, E)$中，每条弧$<u, v>$有两个权值 $b(u, v)$和 $c(u, v)$，分别表示通过该弧的流量的下界和上界；求满足以下条件的所有

可行流 f 中流量最小的可行流。

(1) 弧流量限制条件：$b(u, v) \leqslant f(u, v) \leqslant c(u, v)$，$\forall <u, v> \in E$。

(2) 平衡条件：

$$\sum_{v} f(u,v) - \sum_{w} f(w,u) = \begin{cases} |f| & \text{当} u = V_s \\ 0 & \text{当} u \neq V_s, V_t \\ -|f| & \text{当} u = V_t \end{cases}$$

2. 流量有上下界的网络最小流算法

求有上下界网络最小流的算法是：先按照 6.3.1 节的方法求可行流，如果可行流不存在，则算法结束；否则从可行流出发，倒向求解，即保持网络中的弧方向不变，将 V_t 作为源点，将 V_s 作为汇点，按照 6.1.4 节介绍的标号法把可行流放大，最终求得的最大流即为从 V_s 到 V_t 的最小流。

图 6.34(a)所示的可行流倒向后，因为源点没有发出的弧，汇点没有流入的弧，所以该可行流已无法放大，求最小流没有意义(实际上图 6.34(a)所示的可行流就是原网络的最小流)。为了演示最小流求解过程，本节再举一个例子。

如图 6.35(a)所示的有上下界的网络，源点和汇点既有流量流入，也有流量流出，接下来以该网络为例演示流量有上下界的网络最小流的求解。

图 6.35(b)是图 6.35(a)所示容量网络对应的伴随网络。利用标号法求得伴随网络的最大流，如图 6.35(c)所示，该最大流满足：附加源点 $\overline{V_s}$ 流出的所有弧均满载，所以原网络的可行流存在。原网络的可行流为：$f(e)=\overline{f}(e)+b(e)$，如图 6.35(d)所示，其流量为 0。为了求原网络的最小流，将源点和汇点倒向后，得到如图 6.35(e)所示的网络。对该网络的可行流进行放大，同样在放大可行流的过程中，因为增广路上的反向弧的流量会减少，所以在选择可改进量 α 时要保证反向弧流量减少后不低于流量下限 $b(e)$。求得最大流后，再把源点和汇点交换回来，得到如图 6.35(f)所示的网络流，该网络流就是原网络的最小流，其流量为−2。

3. 算法实现

同样，把例 6.1 的程序稍作修改，即可实现有上下界网络的最小流：在构造好伴随网络后，将源点和汇点互换，然后利用标号法将可行流放大，在选择可改进量 α 时要保证反向弧流量减少后不低于流量下限 $b(e)$；求出最大流后再把源点和汇点交换回来，得到的网络流就是原网络的最小流。其实现方法详见例 6.9。

例 6.9 利用前面介绍的方法求流量有上下界的容量网络的可行流、最大流和最小流，输出各条弧及其流量，以及求得的网络流流量。

假设采用如下格式输入数据：首先输入顶点个数 n 和弧的数目 m，然后输入每条弧的数据。规定源点为第 1 个顶点，汇点为第 n 个顶点。每条弧的数据格式为：$u\ v\ b\ c$，分别表示这条弧的起点、终点、**流量下界**和**流量上界**。顶点序号从 1 开始计起。$n = m = 0$ 表示输入结束。

分析：

顶点序号从 1 开始计起是为了方便地添加附加源点和附加汇点，其顶点序号分别为 0 和 n+1。原网络存储在 Edge 数组中，构造好的伴随网络存储在 AccEdge 数组中。

为了求解流量有上下界的容量网络的可行流、最大流和最小流，将例 6.1 中的 Ford 函数做了修改，添加了 4 个参数。

(1) ArcType network[][MAXN+2]：取值为 Edge 或 AccEdge。

(2) int s：所求网络的源点。

(3) int t：所求网络的汇点。

(4) int max：表示求解伴随网络的最大流、原网络的最大流或最小流。

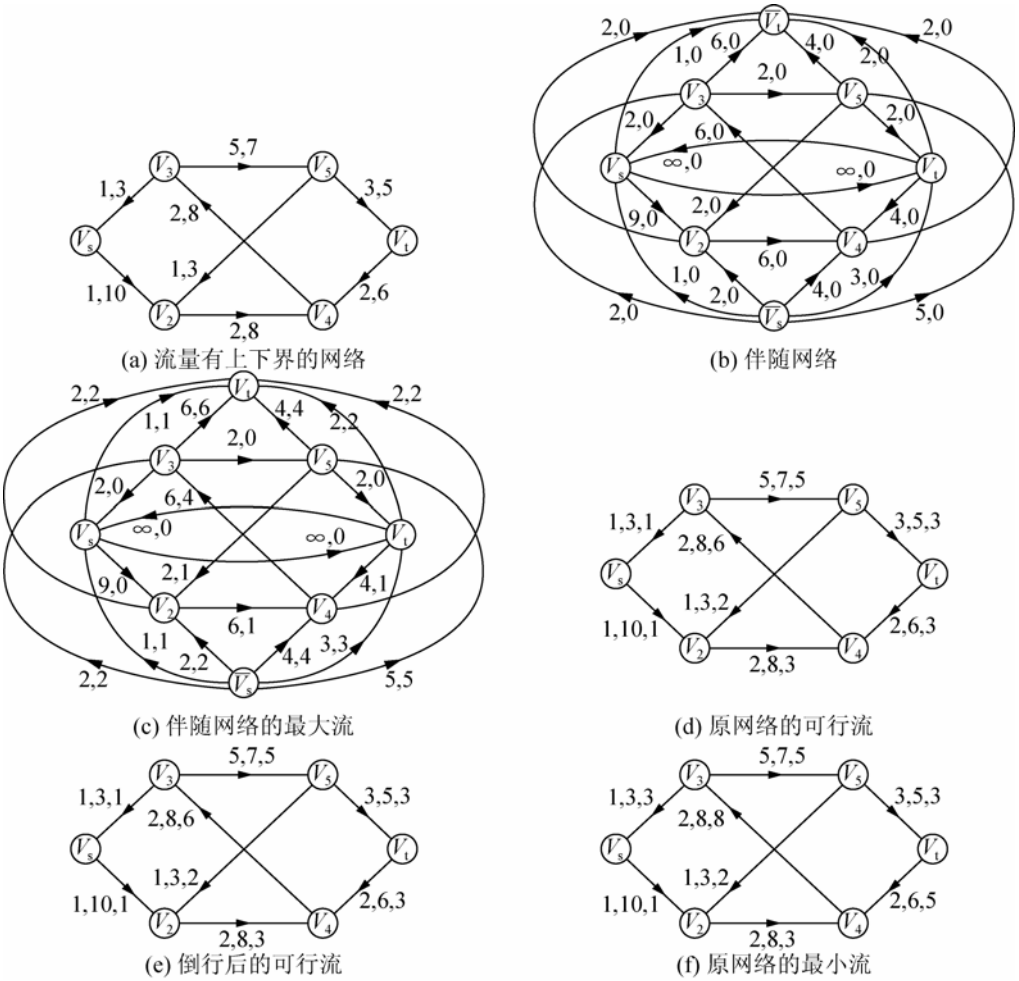


图 6.35　有上下界网络的最小流

如果要求伴随网络的最大流，调用 Ford 函数的形式为：Ford(AccEdge, 0, n+1, −1)，求原网络最大流是调用 Ford 函数的形式为：Ford(Edge, 1, n, 1)，求原网络的最小流时 Ford 函数的调用形式为：Ford(Edge, 1, n, 0)。

代码如下：

```
#define MAXN 10         //顶点个数最大值
#define INF1  1000000   //无穷大1(顶点之间不存在之间弧连接时设置的流量上界)
#define INF2  10000     //无穷大2(伴随网络中新增的两条弧的流量上界)
#define MIN(a,b) ((a)<(b)?(a):(b))
```

```
struct ArcType          //弧结构
{
    int b, c, f;        //弧流量下界、上界、实际流量
};
ArcType Edge[MAXN+2][MAXN+2];       //原网络邻接矩阵(每个元素为 ArcType 类型)
ArcType AccEdge[MAXN+2][MAXN+2];    //伴随网络的邻接矩阵
int n, m;           //顶点个数、弧的数目
int flag[MAXN+2];   //顶点状态：-1——未标号,0——已标号未检查,1——已标号已检查
int prev[MAXN+2];   //标号的第 1 个分量：指明标号从哪个顶点得到，以便找出 α
int alpha[MAXN+2];  //标号的第 2 个分量：可改进量 α
int queue[MAXN+2];  //相当于 BFS 算法中的队列
int v;              //从队列里取出来的队列头元素
int qs, qe;         //当前检查弧的起点(队列头位置)、终点(队列尾位置)
//求容量网络 network 的最大流，网络中的顶点序号为 s～t，其中 s 为源点，t 为汇点
//max 为-1 表示求伴随网络最大流,max 为 1 表示求原网络最大流，max 为 0 表示求最小流
void Ford( ArcType network[][MAXN+2], int s, int t, int max )
{
    int i, j;   //循环变量
    while( 1 )  //标号直至不存在可改进路
    {
        memset( flag, -1, sizeof(flag) );  //标号前对顶点状态数组初始化
        memset( prev,-1,sizeof(prev) ); memset( alpha, -1, sizeof(alpha) );
        flag[s]=0;  prev[s]=0;  alpha[s]=INF1; //源点为已标号未检查顶点
        qs=qe=s;  queue[qe]=s;  qe++;   //源点(顶点 s)入队列

        //qs<qe 表示队列非空, flag[t]==-1 表示汇点未标号
        while( qs<qe && flag[t]==-1 )
        {
            v=queue[qs]; qs++;  //取出队列头顶点
            for( i=s; i<=t; i++ )  //检查顶点 v 的正向和反向"邻接"顶点
            {
                if( flag[i]==-1 )  //顶点 i 未标号
                {
                    //"正向"且流量还可以增加
                    if( network[v][i].c<INF1 && network[v][i].f<network[v]
                        [i].c )
                    {
                        flag[i]=0; prev[i]=v;   //给顶点 i 标号(已标号未检查)
                        alpha[i]=MIN(alpha[v],network[v][i].c-network[v]
                        [i].f );
                        queue[qe]=i;  qe++;     //顶点 i 入队列
                    }
                    //"反向"且有流量还可以减少
                    else if( network[i][v].c<INF1 && network[i][v].f>
                             network[i][v].b )
                    {
                        flag[i]=0;prev[i]=-v; //给顶点 i 标号(已标号未检查)
                        alpha[i]=MIN( alpha[v],network[i][v].f-network[i]
                                      [v].b );
                        queue[qe]=i;  qe++; //顶点 i 入队列
                    }
```

```
                }
            }
            flag[v]=1;  //顶点v已标号已检查
        }
        //当汇点没有获得标号，或者汇点的调整量为0，应该退出while循环
        if( flag[t]==-1 || alpha[t]==0 )  break;
        //当汇点有标号时,应该进行调整了
        int k1=t, k2=abs( prev[k1] );
        int a=alpha[t]; //可改进量
        while( 1 )
        {
            if( network[k2][k1].f<INF1 )                    //正向
                network[k2][k1].f=network[k2][k1].f+a;
            else  network[k1][k2].f=network[k1][k2].f-a;    //反向
            if( k2==s )  break; //调整一直到源点vs
            k1=k2;  k2=abs( prev[k2] );
        }
    }//end of while
    //输出各条弧及其流量，以及求得的最大流(或最小流)流量
    int maxFlow=0;
    for( i=s; i<=t; i++ )
    {
        for( j=s; j<=t; j++ )
        {
            if( i==s && network[i][j].f<INF1 ) //源点流出量
                maxFlow += network[i][j].f;
            if( i==s && network[j][i].f<INF1 ) //源点流入量
                maxFlow-=network[j][i].f;
            if( network[i][j].c<INF1 && network[i][j].f<INF1 )
                printf( "%d->%d : %d\n", i, j, network[i][j].f );
        }
    }
    if( max )  printf( "maxFlow : %d\n", maxFlow );    //输出求得的最大流
    else  printf( "minFlow : %d\n", -maxFlow );        //输出求得的最小流
}
int readcase( ) //读入测试数据
{
    int i, u, v, b, c;          //弧的起点、终点、流量下界、流量上界
    scanf( "%d%d", &n, &m );    //读入顶点个数n
    if( n==0 && m==0 )  return 0;
    for( i=0; i<MAXN+2; i++ )  //初始化邻接矩阵中各元素
    {
        for( int j=0; j<MAXN+2; j++ )
            //INF1表示没有直接边连接
            Edge[i][j].b=Edge[i][j].c=Edge[i][j].f=INF1;
    }
    for( i=1; i<=m; i++ )
    {
        scanf("%d%d%d%d", &u, &v, &b, &c); //读入边的起点和终点、下界、上界
        Edge[u][v].b=b; Edge[u][v].c=c; Edge[u][v].f=0; //构造邻接矩阵
    }
```

```
    return 1;
}
int accompany( )    //构造原网络的伴随网络并求可行流、最大流
{
    memcpy( AccEdge, Edge, sizeof(Edge) );
    int i, j;       //循环变量
    int sum1, sum2;
    //附加源点为顶点 0，附加汇点为顶点 n+1
    for( i=1; i<=n; i++ )
    {
        sum1=sum2=0;
        for( j=0; j<=n; j++ )
        {
            //统计第 i 行(顶点 i 发出的弧)、统计第 i 列(进入到顶点 i 的弧)
            if( AccEdge[i][j].b!=INF1 ) sum1+=AccEdge[i][j].b;
            if( AccEdge[j][i].b!=INF1 ) sum2+=AccEdge[j][i].b;
        }
        //增加一条新弧<i, n+1>和<0, i>
        AccEdge[i][n+1].c=sum1; AccEdge[i][n+1].b=AccEdge[i][n+1].f=0;
        AccEdge[0][i].c=sum2; AccEdge[0][i].b=AccEdge[0][i].f=0;
    }
    for( i=1; i<=n; i++ )
    {
        for( j=1; j<=n; j++ )
        {
            if( AccEdge[i][j].c!=INF1 ) //修改原网络中的弧
            {
                AccEdge[i][j].c=AccEdge[i][j].c-AccEdge[i][j].b;
                AccEdge[i][j].b=0;
            }
        }
    }
    //再增加两条弧：<1, n>和<n, 1>，其流量上界为 INF2
    AccEdge[1][n].c=AccEdge[n][1].c=INF2;
    AccEdge[1][n].b=AccEdge[n][1].b=0;AccEdge[1][n].f=AccEdge[n][1].f=0;
    Ford( AccEdge, 0, n+1, -1 );   //求伴随网络的最大流
    bool feasible=1;   //附加源点发出的所有弧是否均满载，以此判断是否存在可行流
    for( i=0; i<=n+1; i++ )//检查伴随网络中附加源点发出的所有弧是否满载
    {
        if( AccEdge[0][i].c!=INF1 && AccEdge[0][i].f != AccEdge[0][i].c )
            feasible=0;
    }
    if( feasible==0 )  //没有可行流
    {
        printf( "No feasible network flow.\n" );  return 0;
    }
    //求原网络的可行流
    for( i=1; i<=n; i++ )
    {
        for( j=1; j<=n; j++ )
            if( Edge[i][j].c!=INF1 )   //修改原网络中的弧
```

```
                Edge[i][j].f=AccEdge[i][j].f+Edge[i][j].b;
        }
        Ford( Edge, 1, n, 1 );              //求原网络的最大流
        int b, c, f;
        //求原网络的最小流：先还原到原网络的可行流
        for( i=1; i<=n; i++ )
        {
            for( j=1; j<=n; j++ )
                if( Edge[i][j].c!=INF1 )    //修改原网络中的弧
                    Edge[i][j].f=AccEdge[i][j].f+Edge[i][j].b;
        }
        //将原网络的源点和汇点互换(第1行与第n行互换，第1列与第n列互换)
        for( i=1; i<=n; i++ )
        {
            b=Edge[1][i].b; c=Edge[1][i].c; f=Edge[1][i].f;
            Edge[1][i].b=Edge[n][i].b; Edge[1][i].c=Edge[n][i].c;
            Edge[1][i].f=Edge[n][i].f; Edge[n][i].b=b;
            Edge[n][i].c=c; Edge[n][i].f=f;
            b=Edge[i][1].b; c=Edge[i][1].c; f=Edge[i][1].f;
            Edge[i][1].b=Edge[i][n].b; Edge[i][1].c=Edge[i][n].c;
            Edge[i][1].f=Edge[i][n].f; Edge[i][n].b=b;
            Edge[i][n].c=c; Edge[i][n].f=f;
        }
        Ford( Edge, 1, n, 0 ); //求原网络的最小流
        return 1;
    }
    int main( )
    {
        while( readcase( ) )
            accompany( );
        return 0;
    }
```

6.3.4 例题解析

以下通过两道例题的分析，详细介绍流量有上下界网络的最大流和最小流的求解方法。

例 6.10 核反应堆的冷却系统(Reactor Cooling)

题目来源：

Andrew Stankevich's Contest #1, ZOJ2314

题目描述：

本拉登恐怖组织为了制造原子弹，打算建一个核反应堆来生产钚。试设计反应堆的冷却系统。反应堆的冷却系统包含了许多管子，管子里流动的是用来冷却用的特殊液体。管子通过结点相连，每根管子有一个起点、一个终点，冷却液只能从起点流向终点，不能逆向流动。

结点的编号从 1 到 N。冷却系统必须设计得让冷却液体能循环流动，对每个结点，流入结点的流量等于流出结点的流量。即，如果用 $f_{i,j}$ 来标明从结点 i 流向结点 j 的流量(如果

从结点 i 到结点 j 没有管子，则 $f_{i,j}$=0)，对每个结点，都满足以下条件：

$$f_{i,1} + f_{i,2} + \cdots + f_{i,N} = f_{1,i} + f_{2,i} + \cdots + f_{N,i}$$

每根管子都有有限的容量，因此对连接结点 i 和结点 j 的管子，必须满足：$f_{i,j} \leqslant C_{i,j}$，其中 $C_{i,j}$ 是管子的容量。为了提供足够的冷却液，$f_{i,j}$ 还有一个最低流量的限制，即 $f_{i,j} \geqslant L_{i,j}$。

给定所有管子的 $C_{i,j}$ 和 $L_{i,j}$，求满足以上条件的流量 $f_{i,j}$。

输入描述：

输入文件包含多个测试数据。输入文件的第 1 行为一个整数 T，接下来是一个空行，然后是 T 个测试数据。每两个测试数据之间有一个空行。每个测试数据的格式如下。

每个测试数据的第 1 行为两个整数：N 和 M，$1 \leqslant N \leqslant 200$，其中 N 表示结点的数目，M 表示管子的数目。接下来有 M 行，每行描述了一根管子，每行为 4 个整数：$i, j, L_{i,j}$ 和 $C_{i,j}$，$0 \leqslant L_{i,j} \leqslant C_{i,j} \leqslant 10^5$。任意两个结点之间最多有一根管子，没有管子连自结点本身。如果存在一根从结点 i 流向结点 j 的管子，那么就没有从结点 j 流向结点 i 的管子。

输出描述：

对输入文件中的每个测试数据，其输出内容为：如果存在满足条件的流量 $f_{i,j}$，则输出 YES，否则输出 NO；前一种情形还要输出 M 个整数，第 k 个整数为第 k 根管子的流量，管子的序号为输入时的序号。

每个测试数据的输出之后输出一个空行。

样例输入：

```
2

4 6
1 2 1 2
2 3 1 2
3 4 1 2
4 1 1 2
1 3 1 2
4 2 1 2

4 6
1 2 1 3
2 3 1 3
3 4 1 3
4 1 1 3
1 3 1 3
4 2 1 3
```

样例输出：

```
NO

YES
1
2
3
2
1
1
```

分析：

本题要求容量网络中的所有顶点都满足流量平衡条件，可以认为这样的容量网络中没有源点和汇点。本题要求解的是容量网络中的可行流。

题目中给出的两个测试数据所描述的两个容量网络如图 6.36 所示，其中测试数据 1 没有可行流，测试数据 2 有可行流。

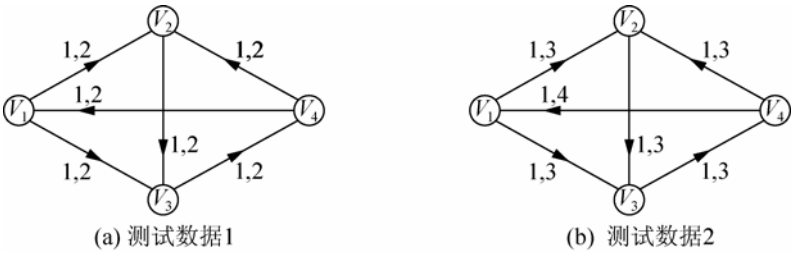


图 6.36　核反应堆的冷却系统：两个测试数据

本题采用另外一种方法构造容量网络的伴随网络 $\overline{D}$。

(1) 新增两个顶点 $\overline{V}_s$ 和 $\overline{V}_t$，$\overline{V}_s$ 称为附加源点，$\overline{V}_t$ 称为附加汇点。

(2) 对原网络 D 中每个顶点 u，按照式(6-16)计算 $D(u)$，如果 $D(u)>0$，则增加一条新弧 $<u,\overline{V}_t>$，这条弧的容量为 $D(u)$；如果 $D(u)<0$，则增加一条新弧 $<\overline{V}_s,u>$，这条弧的容量为 $-D(u)$；如果 $D(u)=0$，则不增加弧。

(3) 原网络 D 中的每条弧，在伴随网络 $\overline{D}$ 中仍保留，但弧的容量 $c(e)$ 修正为：$c(e)-b(e)$。

例如，对测试数据 2 构造的伴随网络 $\overline{D}$ 如图 6.37(a)所示。伴随网络构造好以后，求伴随网络的最大流 $\overline{f}(e)$，如果最大流中从附加源点 $\overline{V}_s$ 流出的所有弧均满载，则原网络 D 存在可行流：$f(e)=\overline{f}(e)+b(e)$；否则原网络 D 不存在可行流。

如果原网络存在可行流，题目还要求输出一个可行流中各弧的流量，顺序为输入时的顺序，所以表示弧结构的结构体 ArcType 中，增加了一个分量 no，为每条弧输入时的序号。

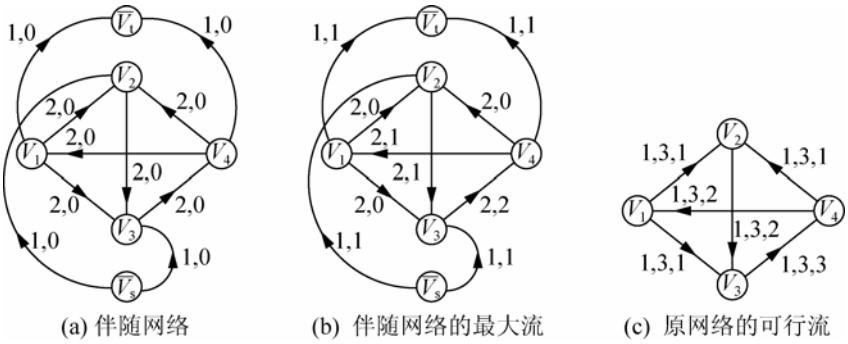


图 6.37　核反应堆的冷却系统：伴随网络的构造及最大流、可行流的求解

代码如下：

```
#define MAXN 201        //顶点个数最大值
#define INF  1000000    //无穷大(顶点之间不存在之间弧连接)
#define MIN(a,b) ((a)<(b)?(a):(b))
struct ArcType          //弧结构
{
    int b, c, f;        //弧流量下界、上界、实际流量
    int no;             //弧的序号
};
ArcType Edge[MAXN+2][MAXN+2];       //原网络邻接矩阵(每个元素为 ArcType 类型)
ArcType AccEdge[MAXN+2][MAXN+2];    //伴随网络的邻接矩阵
int N, M;                           //顶点个数,弧的数目
```

```
int flag[MAXN+2];   //顶点状态：-1——未标号,0——已标号未检查,1——已标号已检查
int prev[MAXN+2];   //标号的第 1 个分量：指明标号从哪个顶点得到，以便找出 α
int alpha[MAXN+2];  //标号的第 2 个分量：可改进量 α
int queue[MAXN+2];  //相当于 BFS 算法中的队列
int v, qs, qe;      //从队列里取出来的队列头元素，队列头位置，队列尾位置
int compare( const void*elem1, const void *elem2 ) //qsort 函数用的比较函数
{
    return ((ArcType *)elem1)->no-((ArcType *)elem2)->no;
}
//求容量网络 network 的最大流，网络中的顶点序号为 s～t，其中 s 为源点，t 为汇点
void Ford( ArcType network[][MAXN+2], int s, int t )
{
    int i;  //循环变量
    while( 1 )  //标号直至不存在可改进路
    {
        memset( flag, -1, sizeof(flag) );  //标号前对顶点状态数组初始化
        memset( prev,-1,sizeof(prev)); memset( alpha, -1, sizeof(alpha) );
        flag[s]=0;  prev[s]=0;  alpha[s]=INF; //源点为已标号未检查顶点
        qs=qe=0;  queue[qe]=s;  qe++;   //源点(顶点 s)入队列
        while( qs<qe && flag[t]==-1 )  //队列非空并且汇点未标号
        {
            v=queue[qs];  qs++; //取出队列头顶点
            for( i=s; i<=t; i++ )  //检查顶点 v 的正向和反向"邻接"顶点
            {
                if( flag[i]==-1 )  //顶点 i 未标号
                {
                    //"正向"且流量还可以增加
                    if( network[v][i].c<INF && network[v][i].f<network[v]
                        [i].c )
                    {
                        flag[i]=0; prev[i]=v; //给顶点 i 标号(已标号未检查)
                        alpha[i]=MIN( alpha[v], network[v][i].c-network[v]
                        [i].f );
                        queue[qe]=i;  qe++; //顶点 i 入队列
                    }
                    //"反向"且有流量还可以减少
                    else if( network[i][v].c<INF&&network[i][v].f>
                             network[i][v].b )
                    {
                        flag[i]=0; prev[i]=-v; //给顶点 i 标号(已标号未检查)
                        alpha[i]=MIN( alpha[v], network[i][v].f-network[i]
                        [v].b );
                        queue[qe]=i;  qe++; //顶点 i 入队列
                    }
                }
            }
            flag[v]=1;  //顶点 v 已标号已检查
        }
        //当汇点无法获得标号，或者汇点的调整量为 0，应该退出 while 循环
        if( flag[t]==-1 || alpha[t]==0 )  break;
        //当汇点有标号时,应该进行调整了
```

```
            int k1=t, k2=abs( prev[k1] ), a=alpha[t];  //a 为可改进量
            while( 1 )
            {
                if( network[k2][k1].f<INF ) //正向
                    network[k2][k1].f=network[k2][k1].f+a;
                else  network[k1][k2].f=network[k1][k2].f-a;   //反向
                if( k2==s )  break; //调整一直到源点 vs
                k1=k2;  k2=abs( prev[k2] );
            }
        }//end of while
    }
    void readcase( )    //读入测试数据
    {
        int i, j, u, v, b, c;       //弧的起点、终点、流量下界、流量上界
        scanf( "%d%d", &N, &M );    //读入顶点个数 N
        for( i=0; i<MAXN+2; i++ )  //初始化邻接矩阵中各元素
        {
            for( j=0; j<MAXN+2; j++ )  //INF 表示没有直接边连接
                Edge[i][j].b=Edge[i][j].c=Edge[i][j].f=Edge[i][j].no=INF;
        }
        for( i=1; i<=M; i++ )
        {
            scanf( "%d%d%d%d", &u, &v, &b, &c ); //读入边的起点和终点、下界、上界
            Edge[u][v].b=b; Edge[u][v].c=c; Edge[u][v].f=0; Edge[u][v].no=i;
        }
    }
    void accompany( )  //构造原网络的伴随网络并求最大流及原网络的可行流
    {
        memcpy( AccEdge, Edge, sizeof(Edge) );
        int i, j;   //循环变量
        //附加源点为顶点 0，附加汇点为顶点 N+1
        for( i=1; i<=N; i++ )
        {
            int sum1=0,  sum2=0;
            for( j=1; j<=N; j++ )
            {
                //统计第 i 行(顶点 i 发出的弧)、统计第 i 列(进入到顶点 i 的弧)
                if( AccEdge[i][j].b!=INF )  sum1+=AccEdge[i][j].b;
                if( AccEdge[j][i].b!=INF )  sum2+=AccEdge[j][i].b;
            }
            if( sum2>sum1 ) //增加一条新弧<0, i>
                AccEdge[0][i].c=sum2-sum1, AccEdge[0][i].b=AccEdge[0][i].f=0;
            else    //增加一条新弧<i, N+1>
                AccEdge[i][N+1].c=sum1-sum2,AccEdge[i][N+1].b=AccEdge[i][N+1].f=0;
        }
        for( i=1; i<=N; i++ )
        {
            for( j=1; j<=N; j++ )
            {
                if( AccEdge[i][j].c!=INF ) //修改原网络中的弧
                {
```

```
                    AccEdge[i][j].c=AccEdge[i][j].c-AccEdge[i][j].b;
                    AccEdge[i][j].b=0;
                }
            }
        }
        Ford( AccEdge, 0, N+1 );    //求伴随网络的最大流
        bool feasible=1;    //附加源点发出的所有弧是否均满载，以此判断是否存在可行流
        for( i=0; i<=N+1; i++ )//检查伴随网络中附加源点发出的所有弧是否满载
        {
            if( AccEdge[0][i].c!=INF && AccEdge[0][i].f != AccEdge[0][i].c )
                feasible=0;
        }
        if( feasible==0 )   //没有可行流
        {
            printf( "NO\n" );  return;
        }
        //求原网络的可行流
        for( i=1; i<=N; i++ )
        {
            for( j=1; j<=N; j++ )
            {
                if( Edge[i][j].c!=INF )//修改原网络中的弧
                    Edge[i][j].f=AccEdge[i][j].f + Edge[i][j].b;
            }
        }
        printf( "YES\n" );
        //按弧的序号从小到大排序
        qsort( Edge, (MAXN+2)*(MAXN+2), sizeof(Edge[0][0]), compare );
        for( i=0; i<M; i++ )  printf( "%d\n", Edge[i/M][i%M].f );
    }
    int main( )
    {
        int T;  scanf( "%d", &T );  //测试数据数目
        for( int i=1; i<=T; i++ )
        {
            readcase( );  accompany( );  printf( "\n" );
        }
        return 0;
    }
```

例 6.11　预算(Budget)

题目来源：

Asia 2003, Tehran (Iran), Preliminary, ZOJ1994, POJ2396

题目描述：

现在要针对多赛区竞赛制定一个预算，该预算是一个行代表不同种类支出、列代表不同赛区支出的矩阵。组委会曾经开会讨论过各类支出的总和，以及各赛区所需支出的总和。另外，组委会还讨论了一些特殊的约束条件：例如，有人提出计算机中心至少需要 1 000k 里亚尔(伊朗货币)，用于购买食物；也有人提出 Sharif 赛区用于购买 T 恤衫的费用不能超

过 30 000k 里亚尔。组委会的任务是制定一个满足所有约束条件且行列和满足要求的预算。

输入描述：

输入文件包含多个测试数据。输入文件的第一行是一个正整数 T，表示测试数据的个数。接下来是一个空行，然后是 T 个测试数据。

每个测试数据的格式如下。

第 1 行为两个整数 m 和 n，分别表示矩阵的行数和列数，$m \leqslant 200$，$n \leqslant 20$。

第 2 行包括 m 个整数，代表矩阵每一行的和。

第 3 行包括 n 个整数，代表矩阵每一列的和。

第 4 行是一个整数 c，表示约束条件的个数。

接下来 c 行，每行给出一个约束条件，格式为：r q 字符 v，其中整数 r 和 q 分别代表行号和列号(如果 r 和 q 都为 0，代表整个矩阵；如果 r 或 q 一个为 0，则代表 r 整行或 q 整列；否则，代表第 r 行第 q 列)；字符取值为{<, =, >}；v 为整数；例如：1 2 > 5 表示第 1 行第 5 列的元素必须严格大于 5；4 0 = 3 表示第 4 行的元素必须都等于 3。

输出描述：

对每组输入数据，输出一个符合要求的所有元素均非负的矩阵，如果找不到这样的矩阵，则输出"IMPOSSIBLE"，每两个矩阵之间输出一个空行。

样例输入：

```
1

2 3
8 10
5 6 7
4
0 2 > 2
2 1 = 3
2 3 > 2
2 3 < 5
```

样例输出：

```
2 3 3
3 3 4
```

分析：

本题可以转化为没有源点、汇点的流量有上下界的可行流模型，关键在于容量网络的构造。接下来以样例输入中第 1 个测试数据为例解释容量网络的构造。从该测试数据可以得到如表 6-2 所示的信息。

表 6-2　预算

行号 / 列号	1	3	3	行和
1	(0, ∞)	(3, ∞)	(0, ∞)	8
2	(3, 3)	(3, ∞)	(2, 4)	10
列和	5	6	7	

其中括号内第 1 个数值表示该项的下界，第 2 个数值表示上界。这样就可以以行和列为顶点构造容量网络。

第 1 步，根据条件构造容量网络。对于上述测试数据构造的容量网络如图 6.38 所示。在图中，每条弧上第 1 个数值表示该弧的下界，第 2 个数值表示该弧的上界。通过这样的转化，把题目转成了上下界可行流模型。只要求出这个网络的一个可行流即可。

第 2 步，将无源汇的上下界可行流模型转化为普通的最大流模型。

首先，对于无源汇的上下界可行流，常见做法是拆边，然后转换成无下界的模型去做：即添加超级源汇 S 和 T，然后将任意一条边(x, y, u, c)(即 x 到 y,下界 u,上界 c 的边)拆成 3 条，分别为$(x, y, 0, c—u)$、$(S, x, 0, u)$和$(y, T, 0, u)$。其思想实际就是让所有边的下界流量的分离出来，作为一条“必要边”(即如果有可行流，这些容量为下界的边一定是满的)，让其统一流入汇，然后让源点来提供这样的流量。然后在这个网络上求最大流。看最大流是否等于所有边的下界之和。当然，在建图时可以先统计每个顶点的入流与出流，然后再加边。

然而此题已经有了源汇，所以就先连一条$(T, S, 0, \infty)$的边(显然这不影响流量平衡条件)。这样就转换成了前面所说的无源汇的情况，然后求之。

第 3 步，求最大流，判断是否有解，有解则输出方案。

输出的结果就让相应原始边的流量加上它们的下界就可以了(即边$(x, y, 0, c—t)$的流量 + u)。

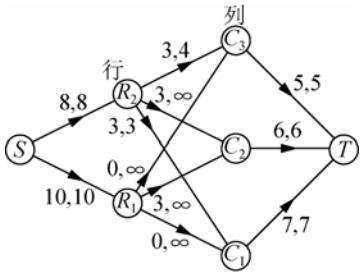


图 6.38　预算：容量网络的构造

代码如下：

```
#define INF 10000000
#define N 300   //顶点上限数
#define KN 205
#define KM 25
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<(b)?(a):(b))
struct edge
{
    int c, f, low, x, y;    //x,y 为边的两点
    edge *nex, *bak;        //同一结点的下一条边,同一条边的另外一个结点
    edge( ){ }
    //nex 为同一结点的上一条边,本边是该结点第 1 条边的时候,nex 应为 NULL
    edge( int x, int y, int c, int f, int low, edge* nex )
        :x(x), y(y), c(c), f(f), low(low), nex(nex), bak(0) { }
    void* operator new( size_t, void *p )  { return p; }
}*E[N]; //保存每个结点的最后一条边
struct NODE
```

```
{
    int low, high;
};
int S, T, m;            //点数,源点,汇点
int Q[100000], D[N];    //队列,标号
edge *cur[N], *path[N];//保存当前弧,路径
edge *base, *data, *it;
NODE limit[KN][KM];     //保存每个格子的约束条件
int sumn[KN], summ[KM], in[N], out[N];
int kn, km;             //kn：方阵的行数，km：方阵的列数
void DFS( )             //深搜建立层次图
{
    memset( D, -1, sizeof(D) );
    int i, j, p1=0, p2=0;
    Q[p2++]=S; D[S]=0;
    while( 1 )
    {
        i=Q[p1++];
        for( edge* e=E[i]; e; e=e->nex )
        {
            if( e->c==0 )  continue;
            j=e->y;
            if( -1==D[j] )
            {
                Q[p2++]=j;  D[j]=D[i]+1;
                if( j==T )  return;
            }
        }
        if( p1==p2 )  break;
    }
}
int maxflow( )
{
//flow用于最大流;path_n为增广路上的边的数目
    int i, k, mink, d, flow=0, path_n;
    while( 1 )
    {
        DFS( );                     //建层次图
        if( D[T]==-1 )  break;      //如果源点不再分层图上,则算法结束
        memcpy( cur, E, sizeof(E) );  path_n=0;
        i=S;
        while( 1 )
        {
            if( i==T )      //找到增广路
            {
                mink=0, d=INF;
//在路径上寻找最小剩余流量, 并记录边的起点
```

```
                for( k=0; k<path_n; ++k )
                {
                    if( (path[k]->c)<d )
                    {
                        d=path[k]->c;  mink=k;
                    }
                }
                for( k=0; k<path_n; ++k )  //修改残余网
                {
                    (path[k]->c)-=d;
                    ((path[k]->bak)->c)+=d;
                }
                path_n=mink;  i=path[path_n]->x;  flow+=d;
            }
            edge* e;
            for( e=cur[i]; e; e=e->nex )   //找一条可以扩展的边
            {
                if( !e->c )  continue;
                int j=e->y;
                if( D[i]+1==D[j] )  break;
            }
            cur[i]=e;
            if( e ) //在路径上保存新边
            {
                path[path_n++]=e;  i=e->y;
            }
            else    //没找到增广路
            {
                D[i]=-1;
                if( !path_n )  break;
                path_n--;  i=path[path_n]->x;   //退一条边,重新搜索
            }
        }
    }
    return flow;
}
//检查是否有满足要求的方案
bool isok( )
{
    for( edge*e=E[S]; e; e=e->nex )
        if( e->c )  return 0;
    return 1;
}
void print( int ok=1 )
{
    int i, j;
    if( ok==-1 )  printf("IMPOSSIBLE\n");
```

```
        else
        {
            for( i=0, it=base; i<kn; i++ )
            {
                for( j=0; j<km; j++, it+=2 )
                    printf( "%d ", it->f-it->c+it->low );
                printf("\n");
            }
        }
        printf("\n");
    }
    int setlimit( int x, int y, char op, int v )
    {
        if( op=='=' )
        {
            if( v>limit[x][y].high )  return 0;
            if( v<limit[x][y].low )  return 0;
            limit[x][y].high=limit[x][y].low=v;
        }
        else if( op=='>' ) //注意加一，因为是大于号
            limit[x][y].low=max( limit[x][y].low, v+1 );
        else if( op=='<')      //注意减一，因为是小于号
            limit[x][y].high=min( limit[x][y].high, v-1 );
        if( limit[x][y].low>limit[x][y].high ) return 0;
        return 1;
    }
    int build( )    //先把所有的关系预处理出来
    {
        int i, j, T, ok=1;
        scanf( "%d%d", &kn, &km );
        for( i=1; i<=kn; i++ )  scanf( "%d", &sumn[i] );
        for( i=1; i<=km; i++ )  scanf( "%d", &summ[i] );
        for( i=1; i<=kn; i++ )
        {
            for( j=1; j<=km; j++ )
            { limit[i][j].low=0;  limit[i][j].high=INF; }
        }
        scanf( "%d", &T );
        while( T-- )
        {
            int x, y, v;
            char op[2];
            scanf( "%d%d%s%d\n", &x, &y, op, &v );
            if( !x && !y )
            {
                for( i=1; i<=kn; i++ )
                    for( j=1; j<=km; j++ )
```

```
                    if( !setlimit(i, j, op[0], v) )  ok=0;
            }
            else if( !x && y )
            {
                for( i=1; i<=kn; i++ )
                    if( !setlimit(i, y, op[0], v) )  ok=0;
            }
            else if( x&&!y )
            {
                for( i=1; i<=km; i++ )
                    if( !setlimit(x, i, op[0], v) )  ok=0;
            }
            else if( !setlimit(x, y, op[0], v) )  ok=0;
        }
        return ok;
    }
    void addedge( int x, int y, int w, int u )
    {
        E[x]=new ((void*) data++) edge(x, y, w, w, u, E[x]);
        E[y]=new ((void*) data++) edge(y, x, 0, 0, u, E[y]);
        E[x]->bak=E[y], E[y]->bak=E[x];
    }
    //S,T：改造后新建图的超级源汇；s,t：原图的源汇
    void solve( )
    {
        int i, j, n, x, y, w, s, t, u, c;
        //注意：预处理中可能出现矛盾，要剔除矛盾的情况
        if( !build( ) )
        {
            print(-1);  return;
        }
        memset( E, 0, sizeof(E) );
        //在新建的图中，行的代表点编号为 2 至 kn+1，列的代表点为 kn+2 至 kn+km+1
        S=0;  T=kn+km+3;  n=kn+km+2;
        memset( in, 0, sizeof(in) );  memset( out, 0, sizeof(out) );
        data=new edge[5 * n * n];  base=data;
        //先建立代表方阵中各个格子数字的边，把这些边建在前面，能够方便最后输出答案
        for( i=1; i<=kn; i++ )
        {
            for( j=1; j<=km; j++ )
            {
                x=i+1;  y=j+kn+1;
                c=limit[i][j].high; u=limit[i][j].low;
                w=c-u;
                addedge(x, y, w, u);
                in[y]+=u; out[x]+=u;
            }
```

```
        }
        s=1;  t=kn+km+2;
        for( i=1; i<=kn; i++ ) //行和的约束
        {
            x=s;  y=i+1;  u=sumn[i];
            in[y]+=u;  out[x]+=u;
        }
        for( i=1; i<=km; i++ ) //列和的约束
        {
            x=i+kn+1;  y=t;  u=summ[i];
            in[y]+=u;  out[x]+=u;
        }
        for( i=1; i<=n; i++ )  //新建图的补边
        {
            if( in[i]>out[i] ){ x=S;  y=i;  w=in[i]-out[i]; }
            else{ x=i;  y=T;  w=out[i]-in[i]; }
            addedge( x, y, w, 0 );
        }
        //将图变成无源汇图
        addedge( t, s, INF, 0 );
        maxflow( );
        if( !isok( ) )  print(-1);
        else  print( );
        delete[] base;
    }
    int main( )
    {
        int T;
        scanf( "%d", &T );
        while( T-- )  solve( );
        return 0;
    }
```

练　　习

6.8　能源(Energy), Asia 2007, ChangChun(Mainland China)

题目描述：

ACM 在 *X* 星球的基地需要能源。幸运的是，在某个广阔的地区里探测到了很多资源，但这个地区离基地很远。基地必须派机器人去采集资源。

基地有两种类型的机器人：*G*–型机器人，负责采集资源；*R*–型机器人，负责修补桥梁。为了节省能源，每个机器人都被限定了转向。机器人每步只能向南或东移动。*G*–型机器人能采集能源，当它到达有资源的地方(称为 *E*–地)时，它将采集 *E*–地的资源，采集到一个单位的资源后，它立刻离开这个地方，然后向南或东继续移动，或者被运回基地。*R*–型机器人能修补桥梁，使得其他机器人能通过。

为了简化题目，本题将这个地区用 $N \times M$ 的矩阵来表示。矩阵中的每个单元的状态如下。

(1) .：机器人能通过这个地方。

(2) #：山或河流，机器人不能通过。

(3) *B*：一座被损坏的桥梁，在修补好之前 *G*-型机器人不能到达这个地方，而当一个 *R*-型机器人到达这个地方时，它可以修复这座桥梁；

(4) *E*：表示一个 *E*-地。在这个地方已经探测到有一定数量的资源。当一个 *G*-型机器人到达这个地方，它采集一个单位的资源。当这个 *E*-地的资源全部被采集完，这个地方将会塌陷，两种类型的机器人都不能通过这个地方。

G-型机器人能采集尽可能多的资源，没有容量限制。两种类型的机器人都能被运到矩阵中任何一个地方，也可以从任何一个地方运回基地。但运输过程会损坏机器人，如果一个机器人有 *L* 的生命期，那么它最多只能被运输 *L* 次。

基地的工作人员计划用这两种机器人采集所有探测到的资源。但是由于每个机器人都有一定的生命期，可能完成不了这个任务。所以必须分析整个地区的地图，判断是否能采集到所有的资源。

输入描述：

输入文件的第 1 行为一个正整数 *T*，1≤*T*≤21，表示测试数据的数目。接下来有 *T* 个测试数据。每个测试数据的第 1 行为 4 个整数：*N*, *M*, *Lr*, *Lc*，*N* 和 *M* 表示地区的大小，*Lr* 表示 *R*-型机器人的生命期，*Lc* 表示 *G*-型机器人的生命期；接下来有 *N* 行，每行有 *M* 个字符，描述了地区的地图；接下来一行有若干个整数 t_i，1≤t_i≤50, 1≤*i*≤50，整数的个数与地图中字母"*E*"的数目一样，每个整数表示对应的 *E*-地探测到的资源数目。*E*-地按从北到南、同一行从西到东的顺序进行编号。

每个测试数据中至少有一个、至多有 50 个 *E*-地，被损坏的桥梁数目少于 11 个。

输出描述：

对输入文件中的每个测试数据，如果可以完成任务，输出"Yes"；否则输出"No"。

样例输入：

```
3

2 2 1 4
.#
#E
2

3 3 2 3
E#E
BEB
##E
1 1 1 1

2 2 0 0
EE
EE
4 3 2 1
```

样例输出：

```
Yes
No
No
```

注解：

第 1 个测试数据中的任务可以完成。一个可行的方案(机器人被运输了 4 次)如下。

G-型机器人：运到(2,2)位置→采集 1 个单位的资源→运回基地。

G-型机器人：运到(2,2)位置→采集 1 个单位的资源→运回基地。

第 2 个测试数据中的任务不可能完成，因为机器人至少必须运输 4 次。

在样例输入中放置第 3 个测试数据的目的是为了解释 *E*-地的编号：(1,1)位置，4 单位资源；(1,2)位置，3 单位资源；(2,1)位置，2 单位资源；(2,2)位置，1 单位资源。

6.4　最小费用最大流

6.4.1　基本概念

1. 问题的引入

6.1 节介绍了网络最大流，毫无疑问，任何容量网络的最大流流量是唯一的、是确定的，但最大流 *f* 是唯一的吗？

例如，在图 6.9 和图 6.10 所示的标号法两个实例中，初始流不同，求得的网络最大流相同，流量也相等。但如图 6.39 所示的容量网络中，最大流流量为 11，图 6.39(a)和 6.39(b)的网络流都取得最大流，因此在该流量网路中，最大流不唯一，即在最大流中，可以选择某些边上走不同的流量，比如图 6.39(a)中边<V_s, V_2>的流量为 8，而在图 6.39(b)中，该边的流量为 4。

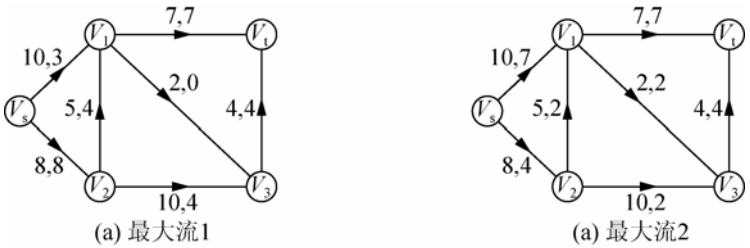


图 6.39　最大流不唯一的例子

既然最大流 *f* 不是唯一的，因此，如果每条弧上不仅有容量限制，还有**费用**，即每条弧上有一个单位费用的参数，那么在保证最大流的前提下，还存在一个选择费用最小的最大流的问题。

2. 最小费用最大流数学模型

先看一个最优运输方案设计的例子。图 6.40 所示是连接产品产地 V_s 和销售地 V_t 的交通网，每一条弧<*u*, *v*>代表从 *u* 到 *v* 的运输线，产品经这条弧由 *u* 输送到 *v*。每条弧旁边有两个数字，第 1 个数字 *c*(*u*, *v*)表示这条运输线的最大通过能力(简称容量，单位：吨)，第 2 个数字 *r*(*u*, *v*)表示每吨产品通过该公路的费用。产品经过交通网从 V_s 输送到 V_t。现在要求制定一个运输方案，使得从 V_s 运到 V_t 的产品数量最多，并且总的费用最少。

图 6.40 所示的网络中，每一条弧<*u*, *v*>除了给定的容量限制 *c*(*u*, *v*)外，还给定了单位流量费用 *r*(*u*, *v*)≥0。上述问题(即最小费用最大流问题)的数学模型为：求一个最大流 *f*，使得

该最大流的总运输费用：

$$r(f)=\sum_{<u,v>\in E} f(u,v)\times r(u,v) \tag{6-21}$$

最少。

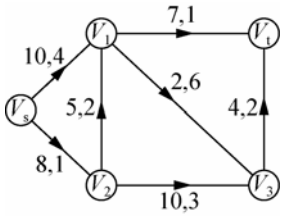


图 6.40　最小费用最大流：最优运输方案的设计

6.4.2　最小费用最大流算法

1. 算法思想

从 6.1.4 节可知，寻找最大流的方法是从某个可行流 f 出发，找到关于这个流的一条增广路 P，沿着 P 调整 f；对新的可行流又试图寻找关于它的增广路；如此反复直至不存在增广路为止。

现在要寻求最小费用最大流，首先考察一下，当沿着一条关于 f 的增广路 P，以改进量 $\alpha=1$ 调整 f，得到新的可行流 f'(显然$|f|=|f'|+1$)，$r(f')$比 $r(f)$增加多少？

不难看出：

$$\begin{aligned} r(f')-r(f)&=\left[\sum_{P+} r(e)*\big(f'(e)-f(e)\big)-\sum_{P-} r(e)*\big(f'(e)-f(e)\big)\right]\\ &=\sum_{P+} r(e)-\sum_{P-} r(e)\end{aligned} \tag{6-22}$$

把

$$\sum_{P+} r(e)-\sum_{P-} r(e) \tag{6-23}$$

称为这条增广路 P 的“费用”。

显然，若 f 是流量为$|f|$的所有可行流中费用最小者，而 P 是关于 f 的所有增广路中费用最小的增广路，那么沿着 P 去调整 f，得到的可行流为 f'，就是流量为$|f'|$的所有可行流中的最小费用流。这样，当 f 是最大流时，它也就是所要求的最小费用最大流了。

注意，由于 $r(u,v)\geqslant 0$，所有 $f=\{0\}$必是流量为 0 的最小费用流。这样，总可以从 $f=\{0\}$开始进行增广。一般，设已知 f 是流量为$|f|$的最小费用流，余下的问题是如何寻求关于 f 的最小费用增广路。为此构造容量网络关于 f 的伴随网络 $W(f)$：它的顶点是原网络中的顶点，而把原网络中的每条弧$<u,v>$变成两个方向相反的弧$<u,v>$和$<v,u>$；定义 $W(f)$中弧的权值为：

$$\begin{aligned} W(u,v)&=\begin{cases} r(u,v) & 若 f(u,v)<c(u,v)\\ +\infty & 若 f(u,v)=c(u,v)\end{cases}\\ W(v,u)&=\begin{cases} -r(u,v) & 若 f(u,v)>0\\ +\infty & 若 f(u,v)=0\end{cases}\end{aligned} \tag{6-24}$$

(长度为∞的可以从$W(f)$中略去)

为什么要做这样的处理？因为同一条弧可能在某条链中是前向弧，在另外一条链中是后向弧，而同一条弧上的流量在调整过程中可能增加或减少，因此，每条弧要变成两个方向相反的弧。

于是在网络中寻求关于 f 的最小费用增广路，就等价于在伴随网络 $W(f)$中，寻求从 V_s 到 V_t 的最短路。

2. 最小费用最大流算法

根据上述分析，得到求解最小费用最大流的算法如下。

(1) 开始取 $f(0)=\{0\}$。

(2) 一般若在第 $k-1$ 步得到的最小费用流为 $f(k-1)$，则构造伴随网络 $W(f(k-1))$。

(3) 在 $W(f(k-1))$中寻求从 V_s 到 V_t 的最短路。若不存在最短路(即最短路的权为$+\infty$)，转 (5)；若存在最短路，则转(4)。

(4) 在原网络 G 中得到相应的增广路 P，在增广路 P 上对 $f(k-1)$进行调整。

$$\alpha=\min\{\min_{P+}(c_{uv}-f_{uv}(k-1)),\ \min_{P-}f_{uv}(k-1)\}$$

$$f_{uv}(k)=\begin{cases}f_{uv}(k-1)+\alpha & (u,v)\in P+\\ f_{uv}(k-1)-\alpha & (u,v)\in P-\\ f_{uv}(k-1) & (u,v)\notin P\end{cases}\tag{6-25}$$

调整后新的可行流为 $f(k)$；转(2)。

(5) $f(k-1)$为最小费用最大流，执行完毕。

接下来以图 6.41(a)所示的容量网络演示最小费用最大流算法的求解过程。

图 6.41(a)：从零流 $f(0)=\{0\}$开始改进最小费用最大流 f。

图 6.41(b)：按照前面介绍的方法构造伴随网络 $W(f(0))$，并求从源点 V_s 到汇点 V_t 的最短路径，在图中用粗线标明最短路径上的边。这条最短路径作为 $f(0)$的费用最小的增广路。

图 6.41(c)：沿着求得的费用最小的增广路，可改进量 $\alpha=5$，改进 $f(0)$，得到 $f(1)$，其流量$|f(1)|=5$。网络中有改进的弧，其流量用粗体、斜体标明了。

图 6.41(d)～图 6.41(e)：构造伴随网络 $W(f(1))$，并求从源点 V_s 到汇点 V_t 的最短路径。这条最短路径作为 $f(1)$的费用最小的增广路，可改进量 $\alpha=2$，改进 $f(1)$，得到 $f(2)$，其流量$|f(2)|=7$。

图 6.41(f)～图 6.41(g)：构造伴随网络 $W(f(2))$，并求从源点 V_s 到汇点 V_t 的最短路径。这条最短路径作为 $f(2)$的费用最小的增广路，可改进量 $\alpha=3$，改进 $f(2)$，得到 $f(3)$，其流量$|f(3)|=10$。

图 6.41(h)～图 6.41(i)：构造伴随网络 $W(f(3))$，并求从源点 V_s 到汇点 V_t 的最短路径。这条最短路径作为 $f(3)$的费用最小的增广路，可改进量 $\alpha=1$，改进 $f(3)$，得到 $f(4)$，其流量$|f(4)|=11$。

在图 6.41(j)中，构造伴随网络 $W(f(4))$后，从源点 V_s 到汇点 V_t 的最短路径不存在，算法结束。因此，最小费用最大流为 $f(4)$，其流量为：$|f(4)|=11$，其费用为：$\sum f(u,v)\times r(u,v)=52$。

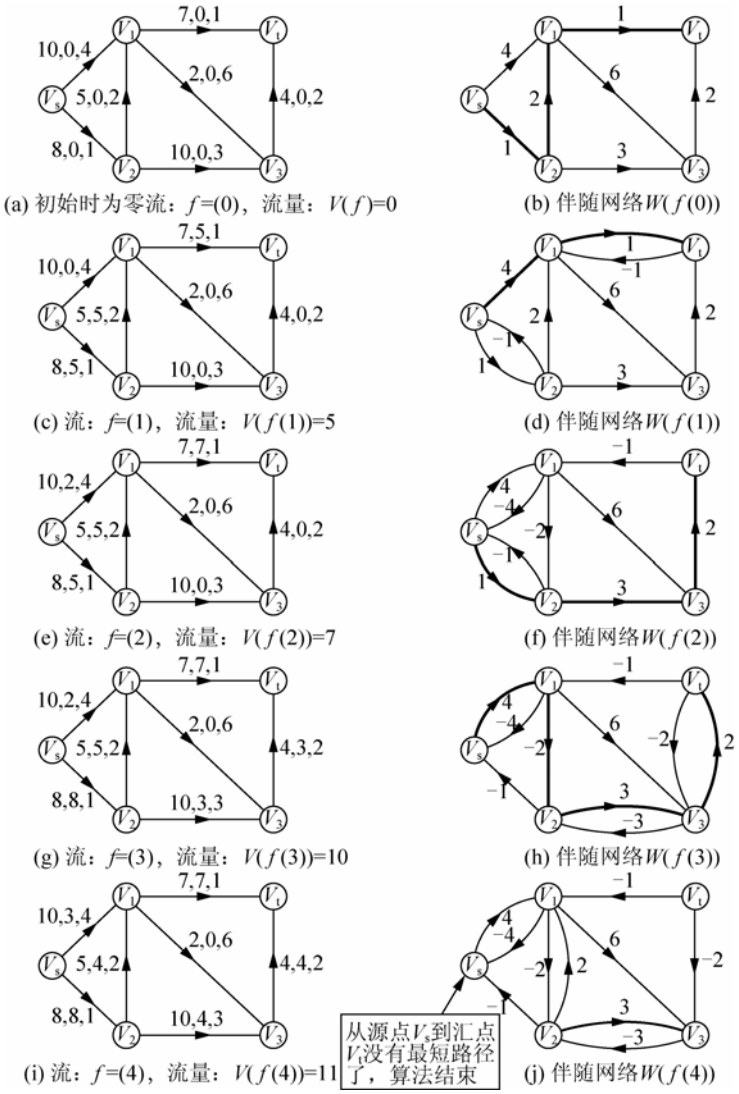


图 6.41　最小费用最大流算法实例

6.4.3　例题解析

以下通过两道例题的分析，详细介绍最小费用最大流的求解方法。

例 6.12　志愿者招募

题目来源：

NOI2008

题目描述：

申奥成功后，布布经过不懈努力，终于成为奥组委下属公司人力资源部门的主管。布布刚上任就遇到了一个难题：为即将启动的奥运新项目招募一批短期志愿者。经过估算，这个项目需要 N 天才能完成，其中第 i 天至少需要 A_i 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 i 类可以从第 S_i 天工作到第

T_i天，招募费用是每人C_i元。新官上任三把火，为了出色地完成自己的工作，布布希望用尽量少的费用招募足够的志愿者，但这并不是他的特长！试帮他设计一种最优的招募方案。

输入描述：

输入文件的第1行包含两个整数N、M，表示完成项目的天数和可以招募的志愿者的种类。

接下来的一行中包含N个非负整数，表示每天至少需要的志愿者人数。

接下来的M行中每行包含3个整数S_i、T_i、C_i，含义如上文所述。为了方便起见，可以认为每类志愿者的数量都是无限多的。

输出描述：

输出一个整数，表示你所设计的最优方案的总费用。

样例输入：

```
4 5
4 2 5 3
1 2 3
1 1 4
2 3 3
3 3 5
3 4 6
```

样例输出：

```
36
```

分析：

本题可以转化成求解容量网络的最小费用最大流问题，本题的关键在于构造容量网络。接下来以样例输入中的第2个测试数据为例解释容量网络的构造方法。

设雇用第i类志愿者的人数为$X[i]$，每个志愿者的费用为$V[i]$，第j天雇用的人数为$P[j]$，则每天的雇用人数应满足一个不等式。对该测试数据而言，可以列出以下4个不等式。

(1) $P[1] = X[1] + X[2] \geqslant 4$

(2) $P[2] = X[1] + X[3] \geqslant 2$

(3) $P[3] = X[3] + X[4] + X[5] \geqslant 5$

(4) $P[4] = X[5] \geqslant 3$

对于每个不等式，可以添加辅助变量$Y[i](Y[i] \geqslant 0)$，使其变为如下等式。

(1) $P[1] = X[1] + X[2] - Y[1] = 4$

(2) $P[2] = X[1] + X[3] - Y[2] = 2$

(3) $P[3] = X[3] + X[4] + X[5] - Y[3] = 5$

(4) $P[4] = X[5] - Y[4] = 3$

在上述等式最前面和最后面添加$P[0] = 0$、$P[5] = 0$，每次用下边的式子减去上边的式子，可以得到如下等式。

(1) $P[1] - P[0] = X[1] + X[2] - Y[1] = 4$

(2) $P[2] - P[1] = X[3] - X[2] - Y[2] + Y[1] = -2$

(3) $P[3] - P[2] = X[4] + X[5] - X[1] - Y[3] + Y[2] = 3$

(4) $P[4] - P[3] = -X[3] - X[4] + Y[3] - Y[4] = -2$

(5) $P[5] - P[4] = -X[5] + Y[4] = -3$

观察发现，每个变量$P[I]$都在两个式子中出现了，而且一次为正，一次为负。所有等

式右边之和为 0。接下来，根据上面 5 个等式构图。

(1) 每个等式为图中一个顶点，添加源点 S 和汇点 T。

(2) 如果一个等式右边为非负整数 c，从源点 S 向该等式对应的顶点连接一条容量为 c、权值为 0 的有向边；如果一个等式右边为负整数 c，从该等式对应的顶点向汇点 T 连接一条容量为$-c$、权值为 0 的有向边。

(3) 如果一个变量 $X[i]$在第 j 个等式中出现为 $X[i]$，在第 k 个等式中出现为$-X[i]$，从顶点 j 向顶点 k 连接一条容量为∞、权值为 $V[i]$的有向边。

(4) 如果一个变量 $Y[i]$在第 j 个等式中出现为 $Y[i]$，在第 k 个等式中出现为$-Y[i]$，从顶点 j 向顶点 k 连接一条容量为∞、权值为 0 的有向边。

构图以后，求从源点 S 到汇点 T 的最小费用最大流，费用值就是结果。

根据上面的例子可以构造出容量网络，如图 6.42(a)所示，其中粗线边为每个变量 X 代表的边，虚线边为每个变量 Y 代表的边，边的容量和权值标已经标出。在该容量网络中求最小费用最大流，网络流如图 6.42(b)所示，每条粗线边的流量就是对应的变量 X 的值。因此，所求的最小费用为：4×3 + 2×3 + 3×6 = 36。

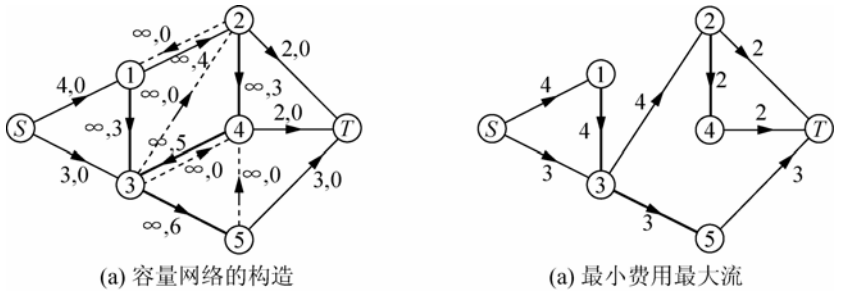


图 6.42　志愿者招募：容量网络的构造

上面的方法很神奇地求出了结果，下面解释为什么要这样构造容量网络。将最后的 5 个等式进一步变形，得出以下结果。

(1) $-X[1]-X[2]+Y[1]+4=0$。

(2) $-X[3]+X[2]+Y[2]-Y[1]-2=0$。

(3) $-X[4]-X[5]+X[1]+Y[3]-Y[2]+3=0$。

(4) $X[3]+X[4]-Y[3]+Y[4]-2=0$。

(5) $X[5]-Y[4]-3=0$。

可以发现，每个等式左边都是几个变量和一个常数相加减，右边都为 0，恰好就像网络流中除了源点和汇点的顶点都满足流量平衡。每个正的变量相当于流入该顶点的流量，负的变量相当于流出该顶点的流量，而正常数可以看作来自附加源点的流量，负的常数是流向附加汇点的流量。因此可以据此构造容量网络，求出从附加源点到附加汇点的网络最大流，即可满足所有等式。而还要求最小，所以要在 X 变量相对应的边上加上权值，然后求最小费用最大流。

代码如下：

```
#define MAXN 1003
#define MAXM 10002*4
```

```
#define INF 1000000
struct edge             //邻接表结构
{
    edge *next, *op;
    int t, c, v;
}ES[MAXM], *V[MAXN];
struct Queue            //队列结构
{
    int Q[MAXN], QH, QL, Size;
    bool inq[MAXN];
    void ins( int v )   //入队列
    {
        if( ++QL>=MAXN ) QL=0;
        Q[QL]=v;
        inq[v]=true;
        Size++;
    }
    int pop( )          //出队列
    {
        int r=Q[QH];
        inq[r]=false;
        Size--;
        if( ++QH>=MAXN ) QH=0;
        return r;
    }
    void reset( )       //清空队列
    {
        memset( Q, 0, sizeof(Q) );
        QH=Size=0;
        QL=-1;
    }
}Q;
int N, M, S, T, EC=-1;
int demond[MAXN], sp[MAXN], prev[MAXN];
edge *path[MAXN];
void addedge( int a, int b, int v, int c=INF ) //插入邻接表
{
    edge e1={ V[a], 0, b, c, v }, e2={ V[b], 0, a, 0, -v };
    ES[++EC]=e1; V[a]=&ES[EC];
    ES[++EC]=e2; V[b]=&ES[EC];
    V[a]->op=V[b]; V[b]->op=V[a];
}
void init( )        //初始化
{
    int i, a, b, c;
    scanf( "%d%d", &N, &M );
    for( i=1; i<=N; i++ ) scanf( "%d", &demond[i] );
    //构造容量网络
    for( i=1; i<=M; i++ )
    {
        scanf( "%d%d%d", &a, &b, &c );
```

```
            addedge( a, b+1, c );
        }
        S=0, T=N+2;
        for( i=1; i<=N+1; i++ )
        {
            c=demond[i]-demond[i-1];
            if( c>=0 ) addedge( S, i, 0, c );
            else addedge( i, T, 0, -c );
            if( i>1 ) addedge( i, i-1, 0 );
        }
    }
    bool SPFA( )    //SPFA 算法求最短路
    {
        int u, v;
        for( u=S; u<=T; u++ ) sp[u]=INF;
        Q.reset( ); Q.ins( S );
        sp[S]=0;    prev[S]=-1;
        while( Q.Size )
        {
            u=Q.pop( );
            for( edge *k=V[u]; k; k=k->next )
            {
                v=k->t;
                if( k->c>0 && sp[u]+k->v<sp[v] )   //松弛操作
                {
                    sp[v]=sp[u] + k->v;
                    prev[v]=u;
                    path[v]=k;
                    if( !Q.inq[v] ) Q.ins(v);  //顶点不在队列中则入队列
                }
            }
        }
        return sp[T] != INF;
    }
    int argument( ) //增广路算法,寻找增广路并调整流量
    {
        int i, cost=INF, flow=0;
        edge *e;
        for( i=T; prev[i]!=-1; i=prev[i] )
        {
            e=path[i];
            if( e->c< cost) cost=e->c;
        }
        for( i=T; prev[i]!=-1; i=prev[i] ) //调整流量
        {
            e=path[i];
            e->c-=cost; e->op->c+=cost;
            flow+=e->v*cost;
        }
        return flow;    //返回调整后的流量
    }
```

```
int maxcostflow( ) //求最小费用最大流
{
    int Flow=0;
    while( SPFA( ) )
        Flow+=argument( );
    return Flow;
}

int main( )
{
    init( );
    printf( "%d\n", maxcostflow( ) );
    return 0;
}
```

例 6.13 卡卡的矩阵之旅(Kaka's Matrix Travels)

题目来源：

POJ Monthly—2007.10.06, POJ3422

题目描述：

有一个 $N \times N$ 大小的矩阵，每个位置上都有一个非负整数。卡卡从 *SUM*=0 开始他的矩阵之旅。每次矩阵之旅，总是从矩阵最左上角位置走到右下角位置，每次移动只能向右移动或向下移动。每次移动到某个方格，卡卡将方格中的数字加到 *SUM*，并将该位置上的数字替换为 0。要求卡卡第一次旅行能获得 *SUM* 的最大值并不难，现在卡卡想知道他旅行完 *K* 次之后，*SUM* 的最大值是多少。注意，*SUM* 的值在这 *K* 次旅行中是累加的。

输入描述：

测试数据的第 1 行为两个整数 *N* 和 *K*，$1 \leqslant N \leqslant 50$，$0 \leqslant K \leqslant 10$。接下来有 *N* 行，描述了一个矩阵，矩阵中的元素都不超过 1 000。

输出描述：

输出卡卡 *K* 次旅行后能获得的最大 *SUM* 值。

样例输入：

```
3 2
1 2 3
0 2 1
1 4 2
```

样例输出：

```
15
```

分析：

本题可以转化为最小费用最大流问题。构建容量网络的方法如下：将每个位置拆成两个——出点和入点，出点和入点之间连接一条容量为 1、费用为矩阵中该位置上的数值；若点 *p* 与 *q* 能连通，则连接 *p* 与 *q*，$n \times n+p$ 与 $n \times n+q$，*p* 与 $n \times n+q$，$n \times n+p$ 与 *q* 四条边，容量为无穷大，费用为 0；假设源点与汇点，源点与 1、汇点与 $2 \times n \times n$ 之间连接边，容量为 *K*，费用为 0。容量网络构建完成后，求最小费用最大流即可。

代码如下：

```
#define INF 10000000
#define MAXN 5100
struct edge
```

```
{
    int next;
    int f, c, w;
}N, P;
vector<edge> map[MAXN];
int s, t, n, k;
int queue[MAXN*1000];   //数组模拟队列
int cost[MAXN];
int pre[MAXN];          //pre[i]为增广路径顶点 i 前一个顶点的序号
int m[51][51];
bool SPFA( )            //SPFA 算法求最短路
{
    int i, H=0, T=0, cur;
    pre[s]=0;
    for( i=0; i<=t; i++ ) cost[i]=INF;
    cost[s]=0; queue[T++]=s;
    while( H<T )
    {
        cur=queue[H++];
        for( i=0; i<map[cur].size(); i++ )
        {
            N=map[cur][i];
            if( N.c-N.f>0 && cost[N.next]>cost[cur]+N.w )  //松弛操作
            {
                cost[N.next]=cost[cur]+N.w;
                pre[N.next]=cur;
                queue[T++]=N.next;
            }
        }
    }
    if( cost[t]!=INF ) return 1;
    else return 0;
}
int argument( ) //增广路算法,得到增广路并调整流量
{
    int i, j, min=INF;
    for( i=t; i!=s; i=pre[i] )
    {
        for( j=0; j<map[pre[i]].size(); j++ )
        {
            if( map[pre[i]][j].next==i && map[pre[i]][j].c-map[pre[i]]
                [j].f<min )
                min=map[pre[i]][j].c-map[pre[i]][j].f;
        }
    }
    for( i=t; i!=s; i=pre[i] ) //调整流量
    {
        for( j=0; j<map[pre[i]].size(); j++ )
            if( map[pre[i]][j].next==i )
                map[pre[i]][j].f+=min;
    }
```

```
    for( i=t; i!=s; i=pre[i] )
    {
        for( j=0; j<map[i].size(); j++ )
            if( map[i][j].next==pre[i] )
                map[i][j].f-=min;
    }
    return min*cost[t]; //返回调整后的流量
}
int maxcostflow( )      //求最小费用最大流
{
    int Flow=0;
    while( SPFA() )
        Flow+=argument();
    return Flow;
}
void build( )   //构建网络
{
    int i;
    N.c=k; N.f=0; N.next=1;N.w=0;
    map[s].push_back(N);
    N.c=0; N.f=0; N.next=s; N.w=0;
    map[1].push_back(N);
    N.c=k; N.f=0; N.next=t; N.w=0;
    map[2*n*n].push_back(N);
    N.c=0; N.f=0; N.next=2*n*n; N.w=0;
    map[t].push_back(N);
    for( i=1; i<=n*n; i++ )
    {
        N.c=1;N.f=0; N.next=n*n+i;N.w=-m[(i-1)/n+1][(i-1)%n+1];
        map[i].push_back(N);
        N.c=0; N.f=0; N.next=i; N.w=m[(i-1)/n+1][(i-1)%n+1];
        map[i+n*n].push_back(N);
    }
    for( i=1; i<=n*n; i++ )
    {
        if( i%n!=0 )
        {
            N.c=INF;N.f=0; N.next=i+1;N.w=0;
            map[i].push_back(N);
            N.c=0; N.f=0; N.next=i; N.w=0;
            map[i+1].push_back(N);
            N.c=INF;N.f=0; N.next=n*n+i+1;N.w=0;
            map[n*n+i].push_back(N);
            N.c=0; N.f=0; N.next=n*n + i; N.w=0;
            map[n*n+i+1].push_back(N);
            N.c=INF;N.f=0; N.next=i+1+ n*n;N.w=0;
            map[i].push_back(N);
            N.c=0; N.f=0; N.next=i; N.w=0;
            map[i+1+n*n].push_back(N);
            N.c=INF;N.f=0; N.next=i+1; N.w=0;
            map[i+n*n].push_back(N);
```

```
            N.c=0; N.f=0; N.next=i+n*n ; N.w=0;
            map[i+1].push_back(N);
        }
        if( i<=n*(n-1) )
        {
            N.c=INF; N.f=0; N.next=i+n;N.w=0;
            map[i].push_back(N);
            N.c=0; N.f=0; N.next=i; N.w=0;
            map[i+n].push_back(N);
            N.c=INF; N.f=0; N.next=i+n+n*n;N.w=0;
            map[n*n+i].push_back(N);
            N.c=0; N.f=0; N.next=i+n*n; N.w=0;
            map[n*n+i+n].push_back(N);
            N.c=INF; N.f=0; N.next=i+n+n*n;N.w=0;
            map[i].push_back(N);
            N.c=0; N.f=0; N.next=i; N.w=0;
            map[i+n+n*n].push_back(N);
            N.c=INF; N.f=0; N.next=i+n; N.w=0;
            map[i+n*n].push_back(N);
            N.c=0; N.f=0; N.next=i+n*n;N.w=0;
            map[i+n].push_back(N);
        }
    }
}
int main( )
{
    int i, j;
    while( scanf("%d %d",&n,&k) != EOF )
    {
        //初始化
        s=0, t=2*n*n+1;
        for( i=1; i<=n; i++ )
        {
            for(j=1; j<=n ; j++)
                scanf("%d",&m[i][j]);
        }
            for( i=0; i<=2*n*n+1; i++ )
            map[i].clear();     //清空
        build( );               //构建网络
        printf( "%d\n", -maxcostflow() );  //求最小费用最大流
    }
    return 0;
}
```

练　习

6.9　回家(Going Home), ZOJ2404, POJ2195

题目描述：

在一个网格地图上，有 *n* 个小人和 *n* 栋房子。在每个单位时间内，每个小人可以往水

平方向或垂直方向上移动一步，走到相邻的方格中。对每个小人，每走一步需要支付 1 美元，直到他走入到一栋房子里。每栋房子只能容纳一个小人。

任务是：要让 n 个小人移动到 n 个不同的房子，需要支付的最小费用。输入的地图中，字符'.'表示空方格，字符'H'代表在该位置上有一栋房子，字符'm'代表该位置上有一个小人。

输入描述：

输入文件包含多个测试数据。每个测试数据的第 1 行为两个整数：N 和 M，$2 \leqslant N$、$M \leqslant 100$，分别代表地图的行和列；接下来有 N 行，每行有 M 个字符，描述了地图，地图中字符'H'和字符'm'的数目一样；每个测试数据中至多有 100 栋房子。输入文件最后一行为两个 0，代表输入结束。

输出描述：

对输入文件中的每个测试数据，输出一行，为一个整数，表示需要支付美元的最少数目。

样例输入：

```
5 5
HH..m
.....
.....
.....
mm..H
0 0
```

样例输出：

```
10
```

6.10　最小费用(Minimum Cost), POJ2516

题目描述：

Dearboy 是一个优秀的食品供应商，他现在面临一个大问题，需要帮忙。在他的销售地区，有 N 个店主(编号从 1～N)帮他销售食品。Dearboy 有 M 仓库(编号从 1～M)，每个仓库可以提供 K 种不同的食品(编号从 1～K)。一旦有店主向他订食品，Dearboy 应该安排哪个仓库、向该店主提供多少食品，以减少总的运输费用？

现已知道，从不同的仓库向不同的店主运输不同种类的单位重量食品所需的费用是不同的。给定每个仓库 K 种食品格子的储藏量，N 个店主对 K 种食品的订量，以及从不同的仓库运输不同种食品到不同的店主的所需费用，试安排每个仓库的各种食品供应量，以减少总的运输费用。

输入描述：

输入文件包含多个测试数据。每个测试数据的第 1 行为 3 个整数：N、M 和 K，$0 < N$、M、$K < 50$，含义如前所述。接下来 N 行描述了每个店主的订量，每行为 K 个整数，范围为[0, 3]，代表每个店主对每种食品的订量。接下来 M 行描述了每个仓库的各种食品的储藏量，每行也是 K 个整数，范围也是[0, 3]，代表每个仓库的每种食品的储藏量。

接下来有 K 个整数矩阵，每个矩阵的大小都是 $N \times M$，矩阵中所有整数的范围都是(0, 100)，第 K 个居中的第 i 行、第 j 列的整数代表将单位重量的第 K 种食品从第 j 个仓库运往第 i 个店主所需的运输费用。

输入文件的最后一行为 3 个 0，表示输入结束。

输出描述：

对输入文件中的每个测试数据，如果 Dearboy 能满足所有店主的所有订购需求，则输出一个整数，代表最小费用。否则输出-1。

样例输入：

```
1 3 3
1 1 1
0 1 1
1 2 2
1 0 1
1 2 3
1 1 1
2 1 1
0 0 0
```

样例输出：

```
4
```

6.11　疏散计划(Evacuation Plan), ZOJ1553, POJ2175

题目描述：

某个城市有许多市政大楼，在城市中修建了一些防辐射的庇护所，这些庇护所是用来在核战争情况下保护市政工作人员。每个庇护所有一定的容量，能容下一定数量的人，而且该城市的所有庇护所中，几乎没有多余的容量。在理想的情况下，所有工作人员从大楼里跑出来，跑向最近的庇护所。然后，这有可能导致某些庇护所拥挤，而其他一些庇护所则是半空状态。

为了解决这个问题，市政委员会设计了一个疏散计划。在计划中，如果将每个工作人员安排到指定的庇护所，这将导致需要维护大量的信息，因此疏散计划是把庇护所分配给市政大楼。委员会列出了每栋大楼需要共用某个庇护所的人数，然后将个人安排的任务交给每栋大楼的管理者。疏散计划考虑到了每栋大楼的人数——所有人都安排到庇护所；同时也充分考虑了每个庇护所的容量是有限的——每个庇护所分配到的人数都不超过它的容量，尽管这将导致某些庇护所没有充分利用。

委员会宣称他们的疏散计划是最优的，也就是说所有工作人员到达指定的庇护所所需的时间是最少的。该市市长，并不相信委员会的疏散计划，因此他想请你验证该疏散计划。要么是验证该疏散计划是最优的，要么通过找到另外一种花费时间更少的方案，从而证明委员会是无能的。

该城市用长方形的网格来描述，市政大楼和庇护所的位置用两个整数来表示，这样从(X_i, Y_i)位置处的市政大楼跑到(P_j, Q_j)位置处的庇护所，所需时间为 $D_{i,j} = |X_i - P_j| + |Y_i - Q_j| + 1$ 分钟。

图 6.43 给出了城市地图的一个例子。在图中，$B_1(5)$、$B_2(6)$等表示大楼，括号里的数字表示大楼里工作人员的数目；$S_1(3)$、$S_2(4)$等表示庇护所，括号里的数字表示庇护所的容量。

输入描述：

输入文件包含多个测试数据，每个测试数据给出了城市的描述和一个疏散计划的描述。每个测试数据的第 1 行为两个整数：N 和 M，用空格隔开，其中 $N(1 \leqslant N \leqslant 100)$表示市政大楼的数目，这些市政大楼编号为 1～$N$；$M(1 \leqslant M \leqslant 100)$表示庇护所的数目，这些庇护所的编号为 1～$M$。

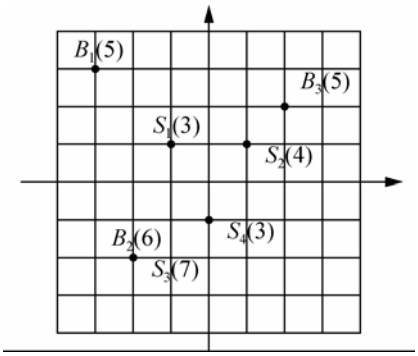


图 6.43　疏散计划：城市地图

接下来有 N 行数据，描述了这 N 个市政大楼。每行为 3 个整数：X_i, Y_i 和 B_i，用空格隔开，其中 X_i、Y_i (−1 000≤X_i、Y_i≤1 000)为市政大楼的坐标位置，B_i(1≤B_i≤1 000)为该市政大楼中的工作人员人数。

接下来有 M 行数目，描述了 M 个庇护所。每行也是 3 个整数：P_j、Q_j 和 C_j，用空格隔开，其中 P_i、Q_i (−1 000≤P_j, Q_j≤1 000)为庇护所的坐标位置，C_j(1≤C_j≤1 000)为庇护所的容量。

接下来 N 行描述了市政委员会的疏散计划。每一行代表一栋市政大楼的疏散计划，按大楼在输入数据中的顺序给出。第 i 栋大楼的疏散计划包含了 M 个整数 $E_{i,j}$，用空格隔开。$E_{i,j}$ (0≤$E_{i,j}$≤1 000)为从第 i 栋大楼疏散到第 j 个庇护所的人数。

输入文件中的疏散计划保证是有效的，也就是说，在计划中，所有人都能疏散出来，每个庇护所也都不会超出它的容量。测试数据一直到文件尾。

输出描述：

对输入文件中的每个测试数据，如果市政委员会的计划是最后的，输出"OPTIMAL"；否则先在第 1 行输出"SUBOPTIMAL"，然后是 N 行，描述疏散计划，格式如输入文件中给出的疏散计划一样。完成计划不一定是最优的，但必须保证是有效的，且比市政委员会的计划更好。

样例输入：

```
3 4
-3 3 5
-2 -2 6
2 2 5
-1 1 3
1 1 4
-2 -2 7
0 -1 3
3 1 1 0
0 0 6 0
0 3 0 2
```

样例输出：

```
SUBOPTIMAL
3 0 1 1
0 0 6 0
0 4 0 1
```