第8章　线性规划与网络流

8.1 线性规划问题和单纯形算法

- **线性规划问题及其表示**
- 线性规划问题可表示为如下形式：

$$\max \quad \sum_{j=1}^{n} c_j x_j \tag{8.1}$$

s.t.

$$\sum_{t=1}^{n} a_{it} x_t \le b_i \quad i = 1,2,\cdots,m_1 \tag{8.2}$$

$$\sum_{t=1}^{n} a_{jt} x_t = b_j \quad j = m_1+1,\cdots,m_1+m_2 \tag{8.3}$$

$$\sum_{t=1}^{n} a_{kt} x_t \ge b_k \quad k = m_1+m_2+1,\cdots,m_1+m_2+m_3 \tag{8.4}$$

$$x_t \ge 0 \quad t = 1,2,\cdots,n \tag{8.5}$$

- 变量满足约束条件(8.2)-(8.5)式的一组值称为线性规划问题的一个**可行解**。
- 所有可行解构成的集合称为线性规划问题的**可行区域**。
- 使目标函数取得极值的可行解称为**最优解**。
- 在最优解处目标函数的值称为**最优值**。
- 有些情况下可能不存在最优解。
- 通常有两种情况：
- (1)根本没有可行解，即给定的约束条件之间是相互排斥的，可行区域为空集；
- (2)目标函数没有极值，也就是说在n 维空间中的某个方向上，目标函数值可以无限增大，而仍满足约束条件，此时目标函数值无界。

$$\begin{aligned} \max \quad & z = x_1 + x_2 + 3x_3 - x_4 \\ & x_1 + 2x_3 \le 18 \\ & 2x_2 - 7x_4 \le 0 \\ & x_1 + x_2 + x_3 + x_4 = 9 \\ & x_2 - x_3 + 2x_4 \ge 1 \\ & x_i \ge 0 \quad i = 1,2,3,4 \end{aligned}$$

- 这个问题的解为 (x1,x2,x3,x4) = (0,3.5,4.5,1)；最优值为16。

线性规划基本定理

- 约束条件(8.2)-(8.5)中n个约束以等号满足的可行解称为线性规划问题的**基本可行解**。
- 若n>m，则基本可行解中至少有n-m个分量为0，也就是说，基本可行解中最多有m个分量非零。
- **线性规划基本定理：**如果线性规划问题有最优解，则必有一基本可行最优解。
- 上述定理的重要意义在于，它把一个最优化问题转化为一个组合问题，即在(8.2) -(8.5)式的m+n个约束条件中，确定最优解应满足其中哪n个约束条件的问题。
- 由此可知，只要对各种不同的组合进行测试，并比较每种情况下的目标函数值，直到找到最优解。
- Dantzig于1948年提出了线性规划问题的单纯形算法。
- 单纯形算法的特点是：
- （1）只对约束条件的若干组合进行测试，测试的每一步都使目标函数的值增加；
- （2）一般经过不大于m或n次迭代就可求得最优解。

约束标准型线性规划问题的单纯形算法

- 当线性规划问题中没有不等式约束(8.2)和(8.4)式，而只有等式约束(8.3)和变量非负约束(8.5)时，称该线性规划问题具有**标准形式**。
- 为便于讨论，不妨先考察一类更特殊的标准形式线性规划问题。这一类线性规划问题中，每一个等式约束中，至少有一个变量的系数为正，且这个变量只在该约束中出现。
- 在每一约束方程中选择一个这样的变量，并以它作为变量求解该约束方程。这样选出来的变量称为左端变量或**基本变量**，其总数为m个。剩下的n-m个变量称为右端变量或**非基本变量**。
- 这一类特殊的标准形式线性规划问题称为**约束标准型线性规划问题**。
- 虽然约束标准型线性规划问题非常特殊，但是对于理解线性规划问题的单纯形算法是非常重要的。
- 稍后将看到，任意一个线性规划问题可以转换为约束标准型线性规划问题。

$$\max \quad z = -x_2 + 3x_3 - 2x_5$$

$$\begin{aligned} x_1 + 3x_2 - x_3 + 2x_5 &= 7 \\ x_4 - 2x_2 + 4x_3 &= 12 \\ x_6 - 4x_2 + 3x_3 + 8x_5 &= 10 \\ x_i \geq 0 \qquad i &= 1,2,3,4,5,6 \end{aligned}$$

		x_2	x_3	x_5
z	0	-1	3	-2
x_1	7	3	-1	2
x_4	12	-2	4	0
x_6	10	-4	3	8

- 任何约束标准型线性规划问题，只要将所有非基本变量都置为0，从约束方程式中解出满足约束的基本变量的值，可求得一个基本可行解。
- 单纯形算法的基本思想就是从一个基本可行解出发，进行一系列的基本可行解的变换。
- 每次变换将一个非基本变量与一个基本变量互调位置，且保持当前的线性规划问题是一个与原问题完全等价的标准线性规划问题。
- 基本可行解x=(7,0,0,12,0,10)。
- **单纯形算法的第1步：**选出使目标函数增加的非基本变量作为**入基变量**。
- 查看单纯形表的第1行（也称之为z行）中标有非基本变量的各列中的值。
- 选出使目标函数增加的非基本变量作为入基变量。
- z行中的正系数非基本变量都满足要求。
- 在上面单纯形表的z行中只有1列为正，即非基本变量相应的列，其值为3。
- 选取非基本变量x_3作为入基变量。
- **单纯形算法的第2步：**选取**离基变量**。
- 在单纯形表中考察由第1步选出的入基变量所相应的列。
- 在一个基本变量变为负值之前，入基变量可以增到多大。

- 如果入基变量所在的列与基本变量所在行交叉处的表元素为负数，那么该元素将不受任何限制，相应的基本变量只会越变越大。
- 如果入基变量所在列的所有元素都是负值，则目标函数无界，已经得到了问题的无界解。
- 如果选出的列中有一个或多个元素为正数，要弄清是哪个数限制了入基变量值的增加。
- 受限的增加量可以用入基变量所在列的元素（称为主元素）来除主元素所在行的"常数列"（最左边的列）中元素而得到。所得到数值越小说明受到限制越多。
- 应该选取受到限制最多的基本变量作为离基变量，才能保证将入基变量与离基变量互调位置后，仍满足约束条件。
- 上例中，惟一的一个值为正的z行元素是3，它所在列中有2个正元素，即4和3。
- min{12/4,10/3}=4，应该选取x_4为离基变量；
- 入基变量x_3取值为3。
- **单纯形算法的第3步：转轴变换。**
- 转轴变换的目的是将入基变量与离基变量互调位置。
- 给入基变量一个增值，使之成为基本变量；
- 修改离基变量，让入基变量所在列中，离基变量所在行的元素值减为零，而使之成为非基本变量。

- 解离基变量所相应的方程，将入基变量x_3用离基变量x_4表示为 $x_3 - \frac{1}{2}x_2 + \frac{1}{4}x_4 = 3$
- 再将其代入其他基本变量和所在的行中消去x_3，

$$x_1 + \frac{5}{2}x_2 + \frac{1}{4}x_4 + 2x_5 = 10$$
$$x_6 - \frac{5}{2}x_2 - \frac{3}{4}x_4 + 8x_5 = 1$$

- 代入目标函数得到 $z = 9 + \frac{1}{2}x_2 - \frac{3}{4}x_4 - 2x_5$
- 形成新单纯形表

		x_2	x_4	x_5
z	9	1/2	-3/4	-2
x_1	10	5/2	1/2	2
x_3	3	-1/2	1/4	0
x_6	1	-5/2	-3/4	8

- **单纯形算法的第4步：**转回并重复第1步，进一步改进目标函数值。
- 不断重复上述过程，直到z行的所有非基本变量系数都变成负值为止。这表明目标函数不可能再增加了。
- 在上面的单纯形表中，惟一的值为正的z行元素是非基本变量x_2相应的列，其值为1/2。
- 因此，选取非基本变量x_2作为入基变量。
- 它所在列中有惟一的正元素5/2，即基本变量x_1相应行的元素。
- 因此，选取x_1为离基变量。
- 再经步骤3的转轴变换得到新单纯形表。
- 新单纯形表z行的所有非基本变量系数都变成负值，求解过程结束。
- 整个问题的解可以从最后一张单纯形表的常数列中读出。
- 目标函数的最大值为11；
- 最优解为：x*=(0,4,5,0,0,11)。

		x_1	x_4	x_5
z	11	-1/5	-4/5	-12/5
x_2	4	5/2	1/10	4/5
x_3	5	1/5	3/10	2/5
x_6	11	1	-1/2	10

单纯形算法计算步骤

- 单纯形算法的计算过程可以用单纯形表的形式归纳为一系列基本矩阵运算。
- 主要运算为转轴变换，该变换类似解线性方程组的高斯消去法中的消元变换。
- **单纯形表：**
- $x_1, x_2, \cdots, x_m$ 为基本变量，$x_{m+1}, x_{m+2}, \cdots, x_n$ 为非基本变量。
- 基本变量下标集为B={1,2,…,m}； 非基本变量下标集为N={m+1,m+2,…,n}；
- 当前基本可行解为（$b_1, b_2, \cdots, b_m, 0, \cdots, 0$）。

		x_{m+1}	x_{m+2}	…	x_n
z	c_0	c_{m+1}	c_{m+2}	…	c_n
x_1	b_1	a_{1m+1}	a_{1m+2}	…	a_{1n}
x_2	b_2	a_{2m+1}	a_{2m+2}	…	a_{2n}
⋮	⋮				
x_m	b_m	a_{mm+1}	a_{mm+2}	…	a_{mn}

- **单纯形算法计算步骤**如下：
- **步骤1：选入基变量**。
- 如果所有$c_j \le 0$，则当前基本可行解为最优解，计算结束。
- 否则取$c_e > 0$相应的非基本变量x_e为入基变量。
- **步骤2：选离基变量**。
- 对于步骤1选出的入基变量x_e，如果所有$a_{ie} \le 0$，则最优解无界，计算结束。
- 否则计算

$$\theta = \min_{a_{ie}>0}\left\{\frac{b_i}{a_{ie}}\right\} = \frac{b_k}{a_{ke}}$$

- 选取基本变量x_k为离基变量。
- 新的基本变量下标集为 $\overline{B} = B + \{e\} - \{k\}$
- 新的非基本变量下标集为 $\overline{N} = N + \{k\} - \{e\}$
- **步骤3：作转轴变换**。
- 新单纯形表中各元素变换如下。

- $$\begin{cases} \overline{b}_i = b_i - a_{ie}\dfrac{b_k}{a_{ke}} & i \in \overline{B} \\ \overline{b}_e = \dfrac{b_k}{a_{ke}} \end{cases}$$ （8.10）
- $$\begin{cases} \overline{a}_{ij} = a_{ij} - a_{ie}\dfrac{a_{kj}}{a_{ke}} & i \in \overline{B},\quad j \in \overline{N} \\ \overline{a}_{ik} = -\dfrac{a_{ie}}{a_{ke}} \end{cases}$$ （8.11）
- $$\begin{cases} \overline{a}_{ej} = \dfrac{a_{kj}}{a_{ke}} & j \in \overline{N} \\ \overline{a}_{ek} = \dfrac{1}{a_{ke}} \end{cases}$$ （8.12）
- $$\begin{cases} \overline{c}_i = c_i - c_e\dfrac{a_{ki}}{a_{ke}} & i \in \overline{N} \\ \overline{c}_k = -\dfrac{c_e}{a_{ke}} \end{cases}$$ （8.13）
- **步骤4：**转步骤1。

将一般问题转化为约束标准型

- 有几种巧妙的办法可以将一般的线性规划问题转换为约束标准型线性规划问题。
- 首先，需要把(8.2)或(8.4)形式的不等式约束转换为等式约束。
- 具体做法是，引入**松弛变量**，利用松弛变量的非负性，将不等式转化为等式。
- 松驰变量记为y_i，共有m_1+m_3个。
- 在求解过程中，应当将松弛变量与原来变量同样对待。求解结束后，抛弃松弛变量。
- 注意松弛变量前的符号由相应的原不等式的方向所确定。

$$\begin{array}{rcl} x_1 + 2x_3 & \le & 18 \\ 2x_2 - 7x_4 & \le & 0 \\ x_1 + x_2 + x_3 + x_4 & = & 9 \\ x_2 - x_3 + 2x_4 & \ge & 1 \end{array} \qquad \begin{array}{rcl} x_1 + 2x_3 + y_1 & = & 18 \\ 2x_2 - 7x_4 + y_2 & = & 0 \\ x_1 + x_2 + x_3 + x_4 & = & 9 \\ x_2 - x_3 + 2x_4 - y_3 & = & 1 \end{array}$$

- 为了进一步构造标准型约束，还需要引入m个**人工变量**，记为z_i。
- 至此，原问题已经变换为等价的约束标准型线性规划问题。
- 对极小化线性规划问题，只要将目标函数乘以-1即可化为等价的极大化线性规划问题。

$$\begin{array}{rcl} z_1 + x_1 + 2x_3 + y_1 & = & 18 \\ z_2 + 2x_2 - 7x_4 + y_2 & = & 0 \\ z_3 + x_1 + x_2 + x_3 + x_4 & = & 9 \\ z_4 + x_2 - x_3 + 2x_4 - y_3 & = & 1 \end{array}$$

一般线性规划问题的2阶段单纯形算法

- 引入人工变量后的线性规划问题与原问题并不等价，除非所有z_i都是0 。
- 为了解决这个问题，在求解时必须分2个阶段进行。
- 第一阶段用一个辅助目标函数 $z' = -\sum_{i=1}^{m} z_i$ 替代原来的目标函数。
- 这个线性规划问题称为原线性规划问题所相应的辅助线性规划问题。
- 对辅助线性规划问题用单纯形算法求解。
- 如果原线性规划问题有可行解，则辅助线性规划问题就有最优解，且其最优值为0，即所有z_i都为0。
- 在辅助线性规划问题最后的单纯形表中，所有z_i均为非基本变量。
- 划掉所有z_i相应的列，剩下的就是只含x_i和y_i的约束标准型线性规划问题了。
- 单纯形算法第一阶段的任务就是构造一个初始基本可行解。
- 单纯形算法第二阶段的目标是求解由第一阶段导出的问题。
- 此时要用原来的目标函数进行求解。
- 如果在辅助线性规划问题最后的单纯形表中，z_i不全为0，则原线性规划问题没有可行解，从而原线性规划问题无解。

退化情形的处理

- 用单纯形算法解一般的线性规划问题时，可能会遇到退化的情形，即在迭代计算的某一步中，常数列中的某个元素的值变成0，使得相应的基本变量取值为0。
- 如果选取退化的基本变量为离基变量，则作转轴变换前后的目标函数值不变。在这种情况下，算法不能保证目标函数值严格递增，因此，可能出现无限循环。
- 考察下面的由Beale在1955年提出的退化问题的例子。
- 按照2阶段单纯形算法求解该问题将出现无限循环。

$$\begin{array}{lrcl} \max & z = \dfrac{3}{4}x_1 - 20x_2 + \dfrac{1}{2}x_3 - 6x_4 & & \\ & \dfrac{1}{4}x_1 - 8x_2 - x_3 + 9x_4 & \le & 0 \\ & \dfrac{1}{2}x_1 - 12x_2 - \dfrac{1}{2}x_3 + 3x_4 & \le & 0 \\ & x_3 & \le & 1 \\ & x_i \ge 0 \qquad i = 1,2,3,4 & & \end{array}$$

- Bland提出避免循环的一个简单易行的方法。
- Bland提出在单纯形算法迭代中，按照下面的2个简单规则就可以避免循环。
- 规则1：设 $e=\min\{j \mid c_j>0\}$ ，取x_e为入基变量。
- 规则2：设 $k=\min\left\{l \mid \frac{b_l}{a_{le}}=\min_{a_{ie}>0}\left\{\frac{b_i}{a_{ie}}\right\}\right\}$
- 取x_k为离基变量。
- 算法leave(col)已经按照规则2选取离基变量。
- 选取入基变量的算法enter(objrow) 中只要加一个break语句即可。

仓库租赁问题

- 某企业计划为流通的货物租赁一批仓库。必须保证在时间段i=1,2,…,n，有b_i的仓库容量可用。现有若干仓库源可供选择。设c_{ij}是从时间段i到时间段j租用1个单位仓库容量的价格，1≤i≤j≤n。应如何安排仓库租赁计划才能满足各时间段的仓库需求，且使租赁费用最少。
- 设租用时间段i到时间段j的仓库容量为y_{ij}，1≤i≤j≤n。则租用仓库的总费用为：$\sum_{i=1}^{n}\sum_{j=i}^{n}c_{ij}y_{ij}$
- 在时间段k可用的仓库容量为：$\sum_{i=1}^{k}\sum_{j=k}^{n}y_{ij}$
- 仓库租赁问题可表述为下面的线性规划问题：

$$\min \quad \sum_{i=1}^{n}\sum_{j=i}^{n}c_{ij}y_{ij}$$

$$\sum_{i=1}^{k}\sum_{j=k}^{n}y_{ij}\geq b_k$$

$$y_{ij}\geq 0$$

- 设 m=n(n+1)/2；
- （ $y_{11},y_{12},\cdots,y_{1n},y_{22},y_{23},\cdots,y_{2n},\cdots,y_{nn}$ ）=（ $x_1,x_2,\cdots,x_m$ ）；
- （ $c_{11},c_{12},\cdots,c_{1n},c_{22},c_{23},\cdots,c_{2n},\cdots,c_{nn}$ ）=（ $d_1,d_2,\cdots,d_m$ ）；
- 上述线性规划问题可表述为n个约束和m个变量的标准线性规划问题如下。

$$\min \quad d^T x$$

$$Ax\geq b$$

$$x\geq 0$$

$$A=\begin{bmatrix}1&1&\cdots&1&1&0&0&\cdots&0&0&0&\cdots&0\\0&1&\cdots&1&1&1&1&\cdots&1&0&&&\\0&0&1&\cdots&1&0&1&\cdots&1&\vdots&\vdots&\vdots&\vdots\\\vdots&\vdots&\vdots&\ddots&\vdots&\vdots&\vdots&\ddots&\vdots&0&0&0&0\\0&0&\cdots&0&1&0&\cdots&0&1&0&\cdots&0&1\end{bmatrix}$$

8.2 最大网络流问题

- **1 基本概念和术语**
-
- **（1） 网络**
- G是一个简单有向图，G=(V,E)，V={1，2，…，n}。
- 在V中指定一个顶点s，称为**源**和另一个顶点t，称为**汇**。
- 有向图G的每一条边(v,w)∈E，对应有一个值cap(v,w)≥0，称为边的**容量**。
- 这样的有向图G称作一个**网络**。
- **（2） 网络流**
- 网络上的**流**是定义在网络的边集合E上的一个非负函数flow={flow(v,w)}，并称flow(v,w)为边(v,w)上的**流量**。

- **（3） 可行流**
- 满足下述条件的流flow称为**可行流**：
- **(3.1)容量约束：**对每一条边(v,w)∈E，0≤flow(v,w)≤cap(v,w)。
- **(3.2)平衡约束：**
- 对于中间顶点：流出量=流入量。
- 即对每个v∈V(v≠s,t)有：顶点v的流出量－顶点v的流入量=0，即

$$\sum_{(v,w)\in E}flow(v,w)-\sum_{(w,v)\in E}flow(w,v)=0$$

- 对于源s：s的流出量－s的流入量=源的净输出量f，即

$$\sum_{(s,v)\in E}flow(s,v)-\sum_{(v,s)\in E}flow(v,s)=f$$

- 对于汇t：t的流入量－t的流出量的=汇的净输入量f，即

$$\sum_{(v,t)\in E}flow(v,t)-\sum_{(t,v)\in E}flow(t,v)=f$$

- 式中f 称为这个可行流的流量，即源的净输出量(或汇的净输入量)。
- 可行流总是存在的。
- 例如，让所有边的流量flow(v,w)=0，就得到一个其流量f=0的可行流(称为0流)。

- **（4） 边流**
- 对于网络G的一个给定的可行流flow，将网络中满足flow(v,w)=cap(v,w)的边称为**饱和边**；flow(v,w)<cap(v,w)的边称为**非饱和边**；flow(v,w)=0的边称为**零流边**；flow(v,w)>0的边称为**非零流边**。当边(v,w)既不是一条零流边也不是一条饱和边时，称为**弱流边**。
- **（5） 最大流**
- 最大流问题即求网络G的一个可行流flow，使其流量f达到最大。即flow满足：
- 0≤flow(v,w)≤cap(v,w)，(v,w)∈E；且

$$\sum flow(v,w)-\sum flow(w,v)=\begin{cases}f & v=s\\0 & v\neq s,t\\-f & v=t\end{cases}$$

- **（6） 流的费用**
- 在实际应用中，与网络流有关的问题，不仅涉及流量，而且还有费用的因素。此时网络的每一条边(v,w)除了给定容量cap(v,w)外，还定义了一个单位流量费用cost(v,w)。对于网络中一个给定的流flow，其费用定义为：

$$\cos t(flow)=\sum_{(v,w)\in E}\cos t(v,w)\times flow(v,w)$$

- **（7）残流网络**
- 对于给定的一个流网络G及其上的一个流flow，网络G关于流flow的残流网络G*与G有相同的顶点集V，而网络G中的每一条边对应于G*中的1条边或2条边。
- 设(v,w)是G的一条边。
- 当flow(v,w)>0时，（w,v）是G*中的一条边，该边的容量为cap*(w,v)=flow(v,w)；
- 当flow(v,w)<cap(v,w)时，(v,w)是G*中的一条边，该边的容量为
- cap*(v,w)=cap(v,w)-flow(v,w)。
- 按照残流网络的定义，当原网络G中的边(v,w)是一条零流边时，残流网络G*中有唯一的一条边(v,w)与之对应，且该边的容量为cap(v,w)。
- 当原网络G中的边(v,w)是一条饱和边时，残流网络G*中有唯一的一条边(w,v)与之对应，且该边的容量为cap(v,w)。
- 当原网络G中的边(v,w)是一条弱流边时，残流网络G*中有2条边(v,w)和(w,v)与之对应，这2条边的容量分别为cap(v,w) -flow(v,w)和flow(v,w)。
- 残流网络是设计与网络流有关算法的重要工具。

增广路算法

- **1 算法基本思想**
- 设P是网络G中联结源s和汇t的一条路。定义路的方向是从s到t。
- 将路P上的边分成2类：
- 一类边的方向与路的方向一致，称为**向前边**。向前边的全体记为P+。
- 另一类边的方向与路的方向相反，称为**向后边**。向后边的全体记为P-。
- 设flow是一个可行流，P是从s到t的一条路，若P满足下列条件：
- （1）在P的所有向前边(v,w)上，flow(v,w)<cap(v,w)，即P+中的每一条边都是非饱和边；
- （2）在P的所有向后边(v,w)上，flow(v,w)>0，即P-中的每一条边都是非零流边。
- 则称P为关于可行流flow的一条可增广路。
- 可增广路是残流网络中一条容量大于0的路。
- 将具有上述特征的路P称为可增广路是因为可以通过修正路P上所有边流量flow(v,w)将当前可行流改进成一个流值更大的可行流。

- 增流的具体做法是：
- （1）不属于可增广路P的边(v,w)上的流量保持不变；
- （2）可增广路P上的所有边(v,w)上的流量按下述规则变化：
- 在向前边(v,w)上，flow(v,w)+d；
- 在向后边(v,w)上，flow(v,w)-d。
- 按下面的公式修改当前的流。

$$flow(v,w)=\begin{cases} flow(v,w)+d & (v,w)\in P^{+} \\ flow(v,w)-d & (v,w)\in P^{-} \\ flow(v,w) & (v,w)\notin P \end{cases}$$

- 其中d称为可增广量，可按下述原则确定：d取得尽量大，又要使变化后的流仍为可行流。
- 按照这个原则，d既不能超过每条向前边(v,w)的cap(v,w)-flow(v,w)，也不能超过每条向后边(v,w)的flow(v,w)。
- 因此d应该等于向前边上的cap(v,w)-flow(v,w)与向后边上的flow(v,w)的最小值。也就是残流网络中P的最大容量。
- **增广路定理：**设flow是网络G的一个可行流，如果不存在从s到t关于flow的可增广路P，则flow是G的一个最大流。

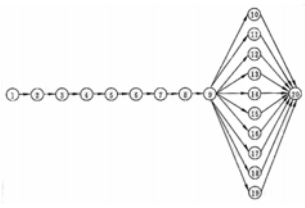
- **2 算法描述**
- 最大流的增广路算法如下。该算法也常称作Ford Fulkerson算法。

```
template <class Graph, class Edge> class MAXFLOW
{   const Graph &G;
    int s, t,maxf;
    vector<int> wt;
    vector<Edge *> st;
    int ST(int v) const { return st[v]->other(v); }
    void augment(int s, int t)
    {   int d = st[t]->capRto(t);
        for (int v = ST(t); v != s; v = ST(v))
          if (st[v]->capRto(v) < d)  d = st[v]->capRto(v);
        st[t]->addflowRto(t, d);
        maxf+=d;
        for ( v = ST(t); v != s; v = ST(v))  st[v]->addflowRto(v, d);
    }
    bool pfs();
public:
     MAXFLOW(const Graph &G, int s, int t,int &maxflow) :
        G(G), s(s), t(t), st(G.V()), wt(G.V()),maxf(0)
     {  while (pfs()) augment(s, t); maxflow+=maxf;}
};
```


- **3 算法的计算复杂性**
- 增广路算法的效率由下面2个因素所确定。
- （1）整个算法找增广路的次数；
- （2）每次找增广路所需的时间。
- 给定的网络中有n个顶点和m条边，且每条边的容量不超过M。
- 可以证明，在一般情况下，增广路算法中找增广路的次数不超过nM次。
- 最短增广路算法在最坏情况下找增广路的次数不超过nm/2次。
- 找1次增广路最多需要O(m)计算时间。
- 因此，在最坏情况下最短增广路算法所需的计算时间为$O(nm^2)$。
- 当给定的网络是稀疏网络，即m=O(n)时，最短增广路算法所需的计算时间为$O(n^3)$。
- 最大容量增广路算法在最坏情况下找增广路的次数不超过2mlogM次。
- 由于使用堆来存储优先队列，找1次增广路最多需要O(nlogn)计算时间。
- 因此，在最坏情况下最大容量增广路算法所需的计算时间为 $O(mn\log n\log M)$
- 当给定的网络是稀疏网络时，最大容量增广路算法所需的计算时间为 $O(n^2\log n\log M)$

预流推进算法

- **1 算法基本思想**
- 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条边进行操作。
- 最坏情况下，每一次增广需要O(n)计算时间。
- 有些情况下，这个代价是很高的。下面是一个极端的例子。



- 无论用哪种增广路算法，都会找到10条增广路，每条路长为10，容量为1。
- 共需要10次增广，每次增广需要对10条边进行操作，每条边增广1个单位流量。
- 10条增广路中的前9个顶点（前8条边）是完全一样的。
- 如果直接将前8条边的流量增广10个单位，而只对后面长为2的不同的有向路单独操作，就可以节省许多计算时间。
- 这就是预流推进（preflow push)算法的基本思想。
- 预流推进算法注重对每一条边的增流，而不必每次一定对一条增广路增流。
- 通常将沿一条边增流的运算称为一次推进（push）。
- 在算法的推进过程中，网络流满足容量约束，但一般不满足流量平衡约束。
- 从每个顶点（除s和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。
- 这种流称为预流（preflow）。这也是这类算法被称为预流推进算法的原因。
- 下面先给出预流的严格定义。
- 给定网络G=(V,E)一个**预流**是定义在G的边集E上的一个正边流函数。
- 该函数满足容量约束，即对G的每一条边(v,w)∈E，满足0≤flow(v,w)≤cap(v,w)。

- G的每一中间顶点满足流出量小于或等于流入量。
- 即对每个v∈V(v≠s,t)有 $\sum_{(v,w)\in E} flow(v,w) \le \sum_{(w,v)\in E} flow(w,v)$
- 满足条件 $\sum_{(v,w)\in E} flow(v,w) < \sum_{(w,v)\in E} flow(w,v)$ 的中间顶点v称为**活顶点**。
- 量 $\sum_{(w,v)\in E} flow(w,v) - \sum_{(v,w)\in E} flow(v,w)$ 称为顶点v的存流。
- 按此定义，源s和汇t不可能成为活顶点。
- 对网络G上的一个预流，如果存在活顶点，则说明该预流不是可行流。
- 预流推进算法就是要选择活顶点，并通过把一定的流量推进到它的邻点，尽可能地将当前活顶点处正的存流减少为0，直至网络中不再有活顶点，从而使预流成为可行流。
- 如果当前活顶点有多个邻点，那么首先推进到哪个邻点呢?
- 由于算法最后的目的是尽可能将流推进到汇点t，因此算法应寻求把流量推进到它的邻点中距顶点t最近的顶点。
- 预流推进算法中用到一个高度函数h来确定推流边。
- 对于给定网络G=(V,E)的一个流，其高度函数h是定义在G的顶点集V上的一个非负函数。该函数满足：
- （1）对于G的残流网络中的每一条边(u,v)有，h(u) ≤ h(v)+1；
- （2）h(t)=0。
- G的残流网络中满足h(u) = h(v)+1的边(u,v)称为G的**可推流边**。

一般的预流推进算法

步骤0： 构造初始预流flow：
对源顶点s的每条出边(s,v)令flow(s,v)=cap(s,v)；
对其余边(u,v)令flow(u,v)=0。构造一有效的高度函数h。
步骤1： 如果残量网络中不存在活顶点，则计算结束，已经得到最大流；
否则转步骤2。
步骤2： 在网络中选取活顶点v。
如果存在顶点v的出边为可推流边，则选取一条这样的可推流边，并沿此边推流。
否则令h(v) = min{h(w)+1 | (v,w)是当前残流网络中的边}，并转步骤1。

- 一般的预流推进算法的每次迭代是一次推进运算或者一次高度重新标号运算。
- 如果推进的流量等于推流边上的残留容量，则称为饱和推进，否则称为非饱和推进。
- 算法终止时，网络中不含有活顶点。此时只有顶点s和t的存流非零。此时的预流实际上已经是一个可行流。
- 算法预处理阶段已经令h(s)=n，而高度函数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不存在增广路。
- 根据增广路定理，算法终止时的可行流是一个最大流。

- 一般的预流推进算法并未给出如何选择活顶点和可推流边。
- 不同的选择策略导致不同的预流推进算法。
- 在基于顶点的预流推进算法中，选定一个活顶点后，算法沿该活顶点的所有推流边进行推流运算，直至无可推流边或该顶点的存变成0时为止。
- **3 算法的计算复杂性**
- 基于顶点的预流推进算法用一个广义队列gQ存储当前活顶点集合。
- 广义队列可以是通常的FIFO队列，LIFO栈，随机化队列，随机化栈，或按各种优先级定义的优先队列。
- 算法的效率与广义优先队列的选择密切相关。
- 如果选用通常的FIFO队列，则在最坏情况下，预流推进算法求最大流所需的计算时间为$O(mn^2)$，其中m和n分别为图G的边数和顶点数。
- 如果以顶点高度值为优先级，选用优先队列实现预流推进算法，则在最坏情况下，求最大流所需的计算时间为 $O(\sqrt{m}n^2)$
- 这个算法也称为最高顶点标号预流推进算法。
- 近来已提出许多其它预流推进算法的实现策略，在最坏情况下算法所需的计算时间已接近$O(mn)$。

8.3 最小费用流问题

- **1 网络流的费用**
- 在实际应用中，与网络流有关的问题，不仅涉及流量，而且还有费用的因素。
- 网络的每一条边(v,w)除了给定容量cap(v,w)外，还定义了一个单位流量费用cost(v,w)。对于网络中一个给定的流flow，其费用定义为：

$$cost(flow) = \sum_{(v,w)\in E} cost(v,w) \times flow(v,w)$$

- **2 最小费用流问题**
- 给定网络G，要求G的一个最大用流flow，使流的总费用最小。

- **3 最小费用可行流问题**
- 给定多源多汇网络G，要求G的一个可行流flow，使可行流的总费用最小。
- 可行流问题等价于最大流问题。最小费用可行流问题也等价于最小费用流问题。

消圈算法

- **1 算法基本思想**
- 最小费用流问题有关的算法中，仍然沿用残流网络的概念。
- 此时，残流网络中边的费用定义为：
- **int costRto(int v) { return from(v) ? -pcost : pcost; }**
- 当残流网络中的边是向前边时，其费用不变。
- 当残流网络中的边是向后边时，其费用为原费用的负值。
- 由于残流网络中存在负费用边，因此残流网络中就不可避免地会产生负费用圈。
- 在与最小费用流问题有关的算法中，负费用圈是一个重要概念。

- **最小费用流问题的最优性条件**
- 网络G的最大流flow是G的一个最小费用流的充分且必要条件是flow所相应的残流网络中没有负费用圈。

- **最小费用流的消圈算法**

步骤0：用最大流算法构造最大流flow。

步骤1：如果残量网络中不存在负费用圈，则计算结束，已经找到最小费用流；否则转步骤2。

步骤2：沿找到的负费用圈增流，并转步骤1。

- **3 算法的计算复杂性**
- 给定网络中有n个顶点和m条边，且每条边的容量不超过M，每条边的费用不超过C。
- 最大流的费用不超过mCM，而每次消去负费用圈至少使得费用下降1个单位，因此最多执行mCM次找负费用圈和增流运算。
- 用Bellman-Ford算法找1次负费用圈需要O(mn)计算时间。
- 最小费用流的消圈算法在最坏情况下需要计算时间$O(m^2nCM)$

最小费用路算法

- **1 算法基本思想**
- 消圈算法首先找到网络中的一个最大流，然后通过消去负费用圈使费用降低。
- 最小费用路算法不用先找最大流，而是用类似于求最大流的增广路算法的思想，不断在残流网络中寻找从源s到汇t的最小费用路，然后沿最小费用路增流，直至找到最小费用流。
- 残流网络中从源s到汇t的最小费用路是残流网络中从s到t的以费用为权的最短路。
- 残流网络中边的费用定义为：

$$wt(v,w)=\begin{cases}\cos t(v,w) & (v,w)\in P^{+}\\ -\cos t(w,v) & (v,w)\in P^{-}\end{cases}$$

- 当残流网络中边(v,w)是向前边时，其费用为cost(v,w)；
- 当(v,w)是向后边时，其费用为-cost(w,v)。

最小费用流的最小费用路算法

步骤0：初始可行0流。

步骤1：如果不存在最小费用路，则计算结束，已经找到最小费用流；

否则用最短路算法在残流网络中找从s到t的最小费用可增广路，转步骤2。

步骤2：沿找到的最小费用可增广路增流，并转步骤1。

- **3 算法的计算复杂性**
- 算法的主要计算量在于连续寻找最小费用路并增流。
- 给定网络中有n个顶点和m条边，且每条边的容量不超过M，每条边的费用不超过C。
- 每次增流至少使得流值增加1个单位，因此最多执行M次找最小费用路算法。
- 如果找1次最小费用路需要s(m,n,C)计算时间，则求最小费用流的最小费用路算法需要O(Ms(m,n,C))计算时间。

网络单纯形算法

- **1 算法基本思想**
- 网络单纯形算法是从解线性规划问题的单纯形算法演变而来，但从算法的运行机制来看，可以将网络单纯形算法看作另一类消圈算法。
- 其基本思想是用一个可行支撑树结构来加速找负费用圈的过程。
- 对于给定的网络G和一个可行流，相应的**可行支撑树**定义为G的一棵包含所有弱流边的支撑树。
- 网络单纯形算法的第一步是构造可行支撑树。
- 从一个可行流出发，不断找由弱流边组成的圈，然后沿找到的弱流圈增流，消除所有弱流圈。
- 在剩下的所有弱流边中加入零流边或饱和边构成一棵可行支撑树。
- 在可行支撑树结构的基础上，网络单纯形算法通过顶点的势函数，巧妙地选择非树边，使它与可行支撑树中的边构成负费用圈。然后，沿找到的负费用圈增流。

- 定义了顶点的势函数Φ后，残流网络中各边(v,w)的势费用定义为：
- c*(v,w)=c(v,w)-(Φ(v)- Φ(w))。
- 其中，c(v,w)是(v,w)在残流网络中的费用。
- 如果对可行支撑树中所有边（v,w）有c*(v,w)=0，则相应的势函数Φ是一个有效势函数。
- 对于一棵可行支撑树，如果将一条非树边加入可行支撑树，产生残流网络中的一个负费用圈，则称该非树边为一条可用边。
- **可用边定理：**给定一棵可行支撑树及其上的一个有效势函数，非树边e是一条可用边的充分必要条件是，e是一条有正势费用的饱和边，或e是一条有负势费用的零流边。
- 事实上，设e=(v,w)。
- 边e与树边t1,t2,…,td构成一个圈cycle: t1,t2,…,t_d，t1, 其中v=t1，w=t_d 。
- 按照边的势费用的定义有：
- c(w,v)=c*(w,v)+ Φ(t_d)- Φ(t_1)
- c(t1,t2)= Φ(t1)- Φ(t2)
- c(t2,t3)= Φ(t2)- Φ(t3)
- …
- c(t_{d-1}, t_d)= Φ(t_{d-1})- Φ(t_d)

- 各式相加得：cost(cycle)=c*(w,v)。
- 由此可见，e是一条可用边当且仅当cost(cycle)<0；
- 当且仅当c*(w,v)<0；
- 当且仅当e是一条有正势费用的饱和边或e是一条有负势费用的零流边。
- **最优性条件：**给定网络G的可行流flow及相应的可行支撑树T，如果不存在T的可用边，则flow是一个最小费用流。
- 事实上，如果不存在T的可用边，则由可用边的定义知残流网络中没有负费用圈。又由最小费用流问题的最优性条件知flow是一个最小费用流。

最小费用流的网络单纯形算法

步骤0：构造flow为初始可行0流。

构造相应的可行支撑树T和有效的顶点势函数。

步骤1：如果不存在T的可用边，则计算结束，已经找到最小费用流；否则转步骤2。

步骤2：选取T的一条可用边与T的树边构成负费用圈，沿找到的负费用圈增流，从T中删去一条饱和边或零流边，重构可行支撑树，并转步骤1。

- **3 算法的计算复杂性**
- 给定网络中有n个顶点和m条边，且每条边的容量不超过M，每条边的费用不超过C。
- 最大流的费用不超过mCM，而每次消去负费用圈至少使得费用下降1个单位，因此最多执行mCM次找负费用圈和增流运算。
- 用网络单纯形算法找1次负费用圈需要O(m)计算时间。
- 因此，求最小费用流的网络单纯形算法在最坏情况下需要计算时间 $O(m^2CM)$