第 7 章　支配集、覆盖集、独立集与匹配

本章内容会涉及以下几个容易相互混淆的概念(设 G 为无向图)。

(1) 点支配集，极小点支配集，最小点支配集，点支配数—$\gamma_0(G)$。

点支配的概念—顶点支配顶点。

(2) 点覆盖集，极小点覆盖集，最小点覆盖集，点覆盖数—$\alpha_0(G)$。

点覆盖的概念—顶点集合的子集覆盖住所有边。

(3) 点独立集，极大点独立集，最大点独立集，点独立数—$\beta_0(G)$。

(4) 边覆盖集，极小边覆盖集，最小边覆盖集，边覆盖数—$\alpha_1(G)$。

边覆盖的概念—边集合的子集覆盖住所有顶点；

(5) 边独立集(匹配)，极大边独立集(极大匹配)，最大边独立集(最大匹配)，边独立数(或匹配数)—$\beta_1(G)$。

以上几个量存在以下关系(设无向图 G 有 n 个顶点，且没有孤立顶点)。

$$\alpha_0+\beta_0=n，即：点覆盖数+点独立数=n。 \tag{7-1}$$

$$\alpha_1+\beta_1=n，即：边覆盖数+边独立数=n。 \tag{7-2}$$

对二部图(设无向二部图 G 有 n 个顶点，且没有孤立顶点)，还有以下关系式。

(1) 二部图的点覆盖数 α_0 ＝匹配数 β_1。 (7-3)

(2) 二部图的点独立数 β_0 ＝顶点个数 n–匹配数 β_1。 (7-4)

说明：如果该二部图有 m 个孤立顶点，则 β_0 和 n 中分别扣除 m 个孤立顶点后满足该式，即：$\beta_0-m=n-m-\beta_1$。这样原式：$\beta_0=n-\beta_1$，仍然成立。

7.1　点支配集、点覆盖集、点独立集

7.1.1　点支配集

1. 支配、点支配集、支配数

支配与**支配集**：设无向图为 $G(V, E)$，顶点集合 $V^*\subseteq V$，若对于$\forall v\in(V-V^*)$，$\exists u\in V^*$，使得$(u, v)\in E$，则称 u **支配** v，并称 V^*为 G 的一个**点支配集**(Vertex Dominating Set，支配集)。

在图 7.1(a)中，取 $V^*=\{v_1, v_5\}$，则 V^*就是一个支配集。因为 $V-V^*=\{v_2, v_3, v_4, v_6, v_7\}$ 中的每个顶点都是 V^*中某个顶点的邻接顶点。

通俗地讲，所谓点支配集，就是 V^*中的顶点能“支配”$V-V^*$中的每个顶点，即 $V-V^*$ 中的每个顶点都是 V^*中某个顶点的邻接顶点，或者说 V 中的顶点要么是 V^*集合中的元素，要么与 V^*中的一个顶点相邻。

注意：在无向图中存在用尽可能少的顶点去支配其他顶点的问题，所以支配集有极小

和最小的概念。最大支配集的概念是没有意义的，因为对任何一个无向图 $G(V, E)$，取 $V^*=V$，总是满足支配集的定义。

极小支配集：若支配集 V^*的任何真子集都不是支配集，则称 V^*是极小支配集。

最小支配集：顶点数最少的支配集称为最小支配集。

点支配数(Vertex Dominating Number)：最小支配集中的顶点数称为点支配数，记作 $\gamma_0(G)$ 或简记为 γ_0。

在图 7.1(a)中，$\{v_1, v_5\}$、$\{v_3, v_5\}$和$\{v_2, v_4, v_7\}$都是极小支配集，$\{v_1, v_5\}$、$\{v_4, v_5\}$和$\{v_3, v_6\}$都是最小支配集，因此 $\gamma_0=2$。

在图 7.1(b)中，$\{v_1\}$和$\{v_2, v_3, v_4, v_5, v_6, v_7\}$都是极小支配集，$\{v_1\}$是最小支配集，因此 $\gamma_0=1$。

在图 7.1(c)中，$\{v_1\}$、$\{v_2, v_4\}$、$\{v_2, v_5\}$等都是极小支配集，显然 $\gamma_0=1$。

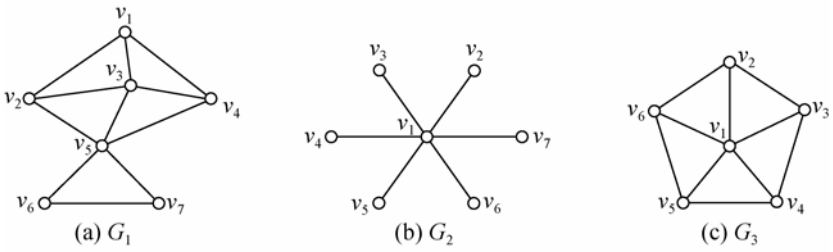


图 7.1　支配与点支配集

2. 点支配集的性质

性质 1　若 G 中无孤立顶点，则**存在**一个支配集 V^*，使得 G 中除 V^*外的所有顶点也组成一个支配集(即 $V-V^*$也是一个支配集)。(证明略。)

思考：在图 7.1(a)中，取 $V^*=\{v_3, v_5, v_6, v_7\}$，$V^*$是支配集，但 $V-V^*$是否是支配集？

性质 2　若 G 中无孤立顶点，V^*为**极小**支配集，则 G 中除 V^*外的所有顶点也组成一个支配集(即 $V-V^*$也是一个支配集)。(证明略。)

3. 应用例子

例 7.1　应用点支配集设置通信基站

假设需要在 8 个城镇 A～H 之间选择若干个城镇建通信基站，使得通信信号覆盖这 8 个城镇。如果在 A 建设一个基站，能同时覆盖到 B、C、D 和 E；如果在 B 建设一个基站，能同时覆盖 A、C 和 G 等。问至少需要建几个基站？

用顶点表示每个城镇，如果在城镇 X 建设一个基站，能同时覆盖到 Y 城镇，那么在顶点 X 和 Y 之间连一条边。这样构造的图如图 7.2 所示。现在将问题转换成求最小支配集问题。在图 7.2 中，最小支配集是$\{A, H\}$，$\gamma_0=2$。因此至少需要建设两个通信基站。

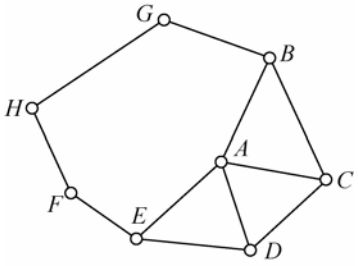


图 7.2　通信基站的设置

7.1.2 点覆盖集

1. 点覆盖边、点覆盖集、点覆盖数

点覆盖集(Vertex Covering Set)：设无向图为 $G(V, E)$，顶点集合 $V^* \subseteq V$，若对于 $\forall e \in E$，$\exists v \in V^*$，使得 v 与 e 相关联，则称 v **覆盖** e，并称 V^* 为 G 的一个**点覆盖集**或简称**点覆盖**。

在图 7.3(a)中，取 $V^* = \{ v_1, v_3, v_5, v_7 \}$，则 V^*就是一个点覆盖集。因为 G 中的每条边都被 V^*中某个顶点“覆盖”住了。

通俗地讲，所谓点覆盖集 V^*，就是 G 中所有的边至少有一个顶点属于 V^*。

注意：

(1) 点覆盖集里的“覆盖”，含义是顶点“覆盖”边；而 7.3.1 节中边覆盖集里的“覆盖”，含义是边“覆盖”顶点。

(2) 在无向图中存在用尽可能少的顶点去“覆盖”住所有边的问题，所以点覆盖集有极小和最小的概念。最大点覆盖集的概念是没有意义的，因为对任何一个无向图 $G(V, E)$，取 $V^*= V$，总是满足点覆盖集的定义。

极小点覆盖：若点覆盖 V^*的任何真子集都不是点覆盖, 则称 V^*是极小点覆盖。

最小点覆盖：顶点个数最少的点覆盖称为最小点覆盖。

点覆盖数(Vertex Covering Number)：最小点覆盖的顶点数称为点覆盖数，记作 $\alpha_0(G)$，简记为 α_0。

在图 7.3(a)中，$\{ v_2, v_3, v_4, v_6, v_7 \}$、$\{ v_1, v_3, v_5, v_7 \}$等都是极小点覆盖，$\{ v_1, v_3, v_5, v_7 \}$等是最小点覆盖，因此 $\alpha_0=4$。

在图 7.3(b)中，$\{ v_1 \}$和$\{ v_2, v_3, v_4, v_5, v_6, v_7 \}$是极小点覆盖，$\{ v_1 \}$是最小点覆盖，因此 $\alpha_0=1$。

在图 7.3(c)中，$\{ v_1, v_2, v_4, v_5 \}$、$\{ v_1, v_2, v_4, v_6 \}$是极小点覆盖，也都是最小点覆盖，因此 $\alpha_0= 4$。

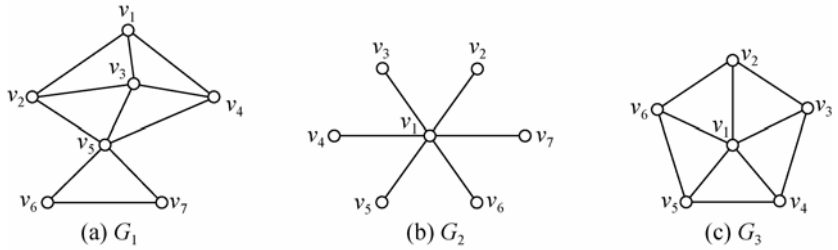


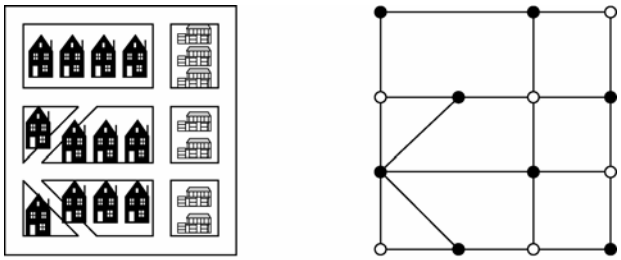
图 7.3 点覆盖集

2. 应用例子

例 7.2 应用点覆盖集配置小区消防设施

某小区计划在某些路口安装消防设施，约定只有与路口直接相连的道路才能使用该路口的消防设施(发生火灾时消防车开进小区道路上并使用路口的消防设施)，为了使所有道路在必要时都能使用消防设施，问至少要配置多少套消防设施。

小区平面图如图 7.4(a)所示，以路口为顶点、街道为边，构造如图 7.4(b)所示的无向图。本题要求用最少的顶点“控制”所有的边，即“覆盖”住所有的边。因此，本题要求的是最小顶点覆盖集。图 7.4(b)给出了一个解，实心圆圈顶点表示安装消防设施的路口。因此最少需要 8 套消防设施。



(a) 小区平面图　　(b) 配置消防设施的路口

图 7.4　小区消防设施的配置

7.1.3　点独立集

1. 点独立集、点独立数

点独立集(Vertex Independent Set)：设无向图为 $G(V, E)$，顶点集合 $V^* \subseteq V$，若 V^*中任何两个顶点均不相邻，则称 V^*为 G 的**点独立集**，或简称**独立集**。

在图 7.5(a)中，取 $V^* = \{ v_1, v_5 \}$，则 V^*就是一个独立集。因为 v_1 和 v_5 是不相邻的。

注意：在无向图中存在将尽可能多的、相互独立的顶点包含到顶点集合的子集 V^*中的问题，所以独立集有极大和最大的概念。最小独立集的概念是没有意义的，因为对任何一个无向图 $G(V, E)$，取 $V^* = \phi$(空集)，总是满足独立集的定义。

极大点独立集：若在 V^*中加入任何顶点都不再是独立集，则称 V^*为极大点独立集。

最大点独立集：顶点数最多的点独立集称为最大点独立集。

点独立数(Vertex Independent Number)：最大点独立集的顶点数称为点独立数，记作 β_0(G)，简记为 β_0。

在图 7.5(a)中，$\{ v_1, v_5 \}$、$\{ v_3, v_6 \}$、$\{ v_2, v_4, v_7 \}$都是极大点独立集，$\{ v_2, v_4, v_7 \}$是最大点独立集，因此 $\beta_0 = 3$。

在图 7.5(b)中，$\{ v_1 \}$和$\{ v_2, v_3, v_4, v_5, v_6, v_7 \}$都是极大点独立集，$\{ v_2, v_3, v_4, v_5, v_6, v_7 \}$是最大点独立集，因此 $\beta_0 = 6$。

在图 7.5(c)中，$\{ v_2, v_4 \}$、$\{ v_2, v_5 \}$都是极大点独立集，也都是最大点独立集，显然 $\beta_0 = 2$。

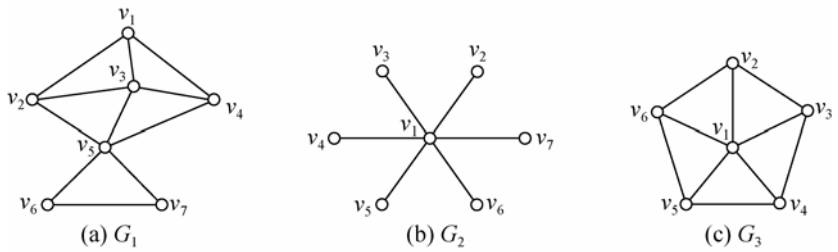


图 7.5　点独立集

2. 应用例子

例 7.3 应用点独立集存放化学药品

某仓库要存放 n 种化学药品，其中有些药品彼此不能存放在一起，因为相互之间可能引起化学或药物反应导致危险，所以必须把仓库分成若干区，各区之间相互隔离。至少把仓库分成多少隔离区，才能确保安全？

考虑 7 种药品的情况。用 $v_1, v_2, \cdots, v_7$ 分别表示这 7 种药品，已知不能存放在一起的药品为：(v_1, v_2)，(v_1, v_4)，(v_2, v_3)，(v_2, v_5)，(v_2, v_7)，(v_3, v_4)，(v_3, v_6)，(v_4, v_5)，(v_4, v_7)，(v_5, v_6)，(v_5, v_7)，(v_6, v_7)。在本题中，把各种药品作为顶点，即顶点集为：$V(G) = \{ v_1, v_2, v_3, v_4, v_5, v_6, v_7 \}$。然后把不能存放在一起的药品用边相连，就构成一个图，如图 7.6 所示。

由点独立集的定义可知，能存放到同一个仓库中的药品应属于同一个点独立集，为了使得仓库数尽可能少，在不导致危险的前提下应该在同一个仓库中存放尽可能多的药品，这个点独立集还应该尽量是极大点独立集。另外，存放在不同仓库中的药品集合分别对应不同的点独立集，而且这些点独立集没有公共元素。(由第 9 章的内容可知，这实际上是图的点着色问题)。

设想把仓库划分为若干个隔离区，分别用Ⅰ、Ⅱ、Ⅲ、…来代表，每个隔离区相当于一个顶点子集。根据题意，图中各边的两个顶点不能存入在同一个隔离区。现在按照如下的方法将各个顶点划分到不同的隔离区中：任取一顶点，如 v_1，存放在Ⅰ区；因 v_2 与 v_1 有边相连，所以把 v_2 存放在Ⅱ区；v_3 与 v_2 有边相连，但与 v_1 无边相连，故可存放在Ⅰ区；……。以此类推，最后一个顶点 v_7，既与 v_5 相连，也与 v_2、v_4、v_6 相连，所以既不能存放在Ⅰ区，也不能存放在Ⅱ区，只好存放在Ⅲ区。从而这 7 种药品可用 3 个隔离区存放。每个隔离区存放的药品分别为，Ⅰ区：v_1，v_3，v_5；Ⅱ区：v_2，v_4，v_6；Ⅲ区：v_7。图 7.6 标明了各顶点(代表对应的药品)所属的隔离区。

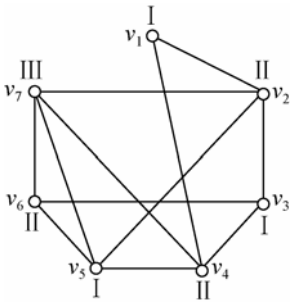


图 7.6 化学药品的存放

例 7.4 应用点独立集设计交通信号灯

图 7.7(a)所示的是两个繁忙街道交叉路口的交通车道。当一辆车到达这个路口时，它会在 9 个车道中的一个车道上出现，这 9 个车道分别记为 $L1 \sim L9$。在这个路口处有一个交通灯，它告知不同车道上的司机何时可以通过这个路口，这是为了确保某些处于不同车道上的车辆不会在同一时间进入路口，比如 $L1$ 和 $L7$。然而 $L1$ 和 $L5$ 上的车辆同时穿过路口是没有问题的。现在的问题是，为了让所有的车辆都能安全通过路口，对于交通灯来说，所需要的相位最少是多少？

用顶点 $L1$～$L9$ 表示每个车道，当两个车道上的车辆不能同时进入路口时(否则可能引发交通事故)，则在这两个车道对应的顶点间连一条边。构造好的无向图如图 7.7(b)所示。

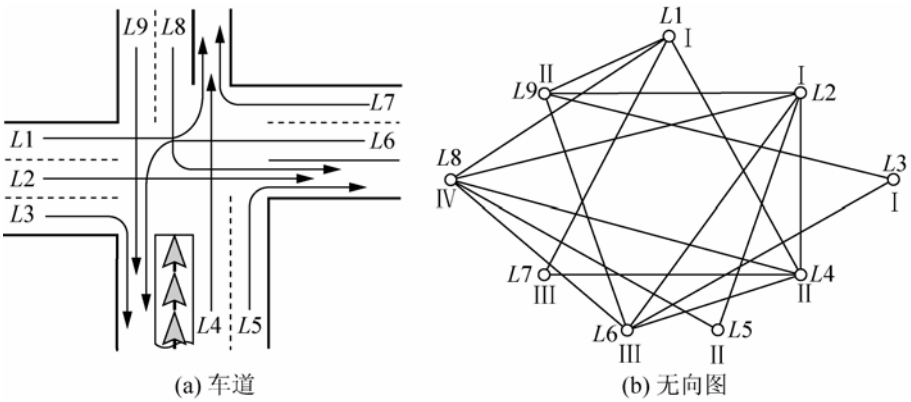


图 7.7　街道交叉路口的交通车道

由点独立集的定义可知，能同时进入路口的车道应该属于同一个点独立集，在不引发交通事故的前提下为了让尽可能多的车道同时进入路口，这个点独立集还应该尽量是极大点独立集。另外，由信号灯不同相位控制的车道集合分别对应不同的点独立集，而且这些点独立集没有公共元素(由第 9 章的内容可知，这实际上是图的点着色问题)。

如图 7.7(b)所示的无向图中最少可以分成 4 个没有公共元素的点独立集，例如：{ $L1, L2, L3$ }、{ $L4, L5, L9$ }、{ $L6, L7$ }、{ $L8$ }，而且各顶点所属的是点独立集。这样每个点独立集中的车道由信号灯一个相位控制，所以至少需要 4 个相位的信号灯。

7.1.4　点支配集、点覆盖集、点独立集之间的联系

点支配集、点覆盖集、点独立集都是顶点的集合，这些集合之间存在以下联系。

定理 7.1　设无向图 $G(V, E)$中无孤立顶点，则 G 的极大点独立集都是 G 的极小支配集。逆命题不成立(即极小支配集未必是极大独立集)。

定理 7.2　一个独立集是极大独立集，当且仅当它是一个支配集。

定理 7.3　设无向图 $G(V, E)$中无孤立顶点，顶点集合 $V^* \subseteq V$，则 V^*是 G 的点覆盖，当且仅当 $V - V^*$是 G 的点独立集。

推论：设 G 是 n 阶无孤立点的图，则 V^*是 G 的极小(最小)点覆盖集，当且仅当 $V - V^*$是 G 的极大(最大)点独立集，从而有：$\alpha_0 + \beta_0 = n$ (n 为顶点个数)。

7.2　点支配集、点覆盖集、点独立集的求解

7.2.1　逻辑运算

因为点支配集、点覆盖集、点独立集都是顶点集合，求解时要用到集合的逻辑运算。设 G 是一个图，用 v_i 表示事件“包含顶点 v_i”。定义以下两种逻辑运算。

(1) 逻辑或运算：用 $v_i + v_j$ 或 $v_i \vee v_j$ 表示事件“要么包含顶点 v_i，要么包含顶点 v_j”。

(2) 逻辑与运算：用 v_iv_j 或 $v_i \wedge v_j$ 表示事件“既包含顶点 v_i，又包含顶点 v_j”。

则上述逻辑运算满足以下运算定律。

(1) 交换律：$v_i + v_j = v_j + v_i$；$v_iv_j = v_jv_i$。

(2) 结合律：$(v_i + v_j) + v_k = v_i + (v_j + v_k)$；$(v_iv_j)v_k = v_i(v_jv_k)$。

(3) 分配律：$v_i(v_j + v_k) = v_iv_j + v_jv_k$；$(v_j + v_k)v_i = v_iv_j + v_iv_k$。

(4) 吸收律：$v_i + v_i = v_i$；$v_iv_i = v_i$；$v_i + v_iv_j = v_i$。

上述定律尤其是吸收律，在求解点支配集、点覆盖集、点独立集时用处很大。

7.2.2 极小点支配集的求解

设无向连通图为 $G(V, E)$，顶点集合 $V = \{v_1, v_2, \cdots, v_n\}$，则求所有极小支配集的公式为：

$$\gamma(v_1, v_2, \cdots, v_n) = \prod_{i=1}^{n}\left(v_i + \sum_{u \in N(v_i)} u\right) \tag{7-5}$$

式中：$N(v_i)$为顶点 v_i 的邻接顶点集合，也称 v_i 的为**邻域**(Neighborhood)；$\sum$ 表示求和；$\prod$ 表示连乘。在上式中，每个顶点与它的所有邻接顶点进行加法运算组成一个因子项，所有因子项再连乘。连乘过程中根据上述运算规律展开成积之和的形式。在运算完毕得到的结果中，每个乘积项代表一个极小支配集，其中最小者为最小支配集。

例如，对图 7.1(a)所示的无向图，求所有极小支配集的公式为：

$$\begin{aligned}
&\gamma(v_1, v_2, v_3, v_4, v_5, v_6, v_7) \\
&= (v_1 + v_2 + v_3 + v_4)(v_2 + v_1 + v_3 + v_5)(v_3 + v_1 + v_2 + v_4 + v_5)(v_4 + v_1 + v_3 + v_5) \\
&\quad (v_5 + v_2 + v_3 + v_4 + v_6 + v_7)(v_6 + v_5 + v_7)(v_7 + v_5 + v_6) \\
&= (1 + 2 + 3 + 4)(2 + 1 + 3 + 5)(3 + 1 + 2 + 4 + 5)(4 + 1 + 3 + 5) \\
&\quad (5 + 2 + 3 + 4 + 6 + 7)(6 + 5 + 7)(7 + 5 + 6) \\
&= 15 + 16 + 17 + 246 + 247 + 25 + 35 + 36 + 37 + 45
\end{aligned}$$

因此，该图的所有极小支配集为：$\{v_1, v_5\}$、$\{v_1, v_6\}$、$\{v_1, v_7\}$、$\{v_2, v_4, v_6\}$、$\{v_2, v_4, v_7\}$、$\{v_2, v_5\}$、$\{v_3, v_5\}$、$\{v_3, v_6\}$、$\{v_3, v_7\}$、$\{v_4, v_5\}$。点支配数 $\gamma_0(G) = 2$。

请注意理解上述乘法的执行过程。以$(1 + 2 + 3 + 4)(2 + 1 + 3 + 5)$为例解释：因为根据吸收律有 $11 = 1$，以及 $12 = 1, \cdots, 15 = 1$，所以所有包含 1 的乘积项都被 1“吸收”了；同理所有包含 2 的乘积项被 2 吸收了，所有包含 3 的乘积项都被 3 吸收了；这样该乘积运算的结果为：$1 + 2 + 3 + 45$。

7.2.3 极小点覆盖集、极大点独立集的求解

设无向连通图为 $G(V, E)$，顶点集合 $V = \{v_1, v_2, \cdots, v_n\}$，则求所有极小覆盖集的公式为：

$$\alpha(v_1, v_2, \cdots, v_n) = \prod_{i=1}^{n}\left(v_i + \prod_{u \in N(v_i)} u\right)。 \tag{7-6}$$

在式(7-6)中，每个顶点的所有邻接顶点进行积运算后再与该顶点进行和运算，组成一个因子项；所有因子项再连乘，并根据逻辑运算规律展开成积之和的形式。在运算完毕得到的结果中，每个乘积项代表一个极小覆盖集，其中最小者为最小覆盖集。

例如，对如图 7.1(a)所示的无向图，求所有极小覆盖集的公式为：

$\alpha(v_1, v_2, v_3, v_4, v_5, v_6, v_7)$

$= (v_1 + v_2v_3v_4)(v_2 + v_1v_3v_5)(v_3 + v_1v_2v_4v_5)(v_4 + v_1v_3v_5)(v_5 + v_2v_3v_4v_6v_7)(v_6 + v_5v_7)(v_7 + v_5v_6)$

$= (1 + 234)(2 + 135)(3 + 1245)(4 + 135)(5 + 23467)(6 + 57)(7 + 56)$

$= 12456 + 12457 + 1356 + 1357 + 23456 + 23457 + 23467$

因此，该图的所有极小覆盖集为：$\{v_1, v_2, v_4, v_5, v_6\}$、$\{v_1, v_2, v_4, v_5, v_7\}$、$\{v_1, v_3, v_5, v_6\}$、$\{v_1, v_3, v_5, v_7\}$、$\{v_2, v_3, v_4, v_5, v_6\}$、$\{v_2, v_3, v_4, v_5, v_7\}$、$\{v_2, v_3, v_4, v_6, v_7\}$。点覆盖数 $\alpha_0(G) = 4$。

由 7.1.4 节定理 3 的推论可知：无向连通图 G 的极小点覆盖集与极大点独立集存在互补性，求出极小点覆盖集和点覆盖数后，就可以求出极大点独立集和点独立数。由此得图 7.1(a)的极大点独立集为：

$$V - \{v_1, v_2, v_4, v_5, v_6\} = \{v_3, v_7\}$$
$$V - \{v_1, v_2, v_4, v_5, v_7\} = \{v_3, v_6\}$$
$$V - \{v_1, v_3, v_5, v_6\} = \{v_2, v_4, v_7\}$$
$$V - \{v_1, v_3, v_5, v_7\} = \{v_2, v_4, v_6\}$$
$$V - \{v_2, v_3, v_4, v_5, v_6\} = \{v_1, v_7\}$$
$$V - \{v_2, v_3, v_4, v_5, v_7\} = \{v_1, v_6\}$$
$$V - \{v_2, v_3, v_4, v_6, v_7\} = \{v_1, v_5\}$$

点独立数 $\beta_0(G) = n - \alpha_0(G) = 3$。

说明：上述求极小点支配集、极小点覆盖集和极大点独立集算法的复杂度是指数阶的。例如，对于求极小点覆盖集的算法，需要处理式(7-6)右边式子展开式中的 2^n 个乘积项，因此时间复杂度至少是 $O(2^n)$。事实上，极小点支配集、极小点覆盖集和极大点独立集问题都是 NP 问题，目前尚没有有效的精确算法。因此上述算法只能用来计算比较简单(顶点数 n 较小)的图。

7.3　边覆盖集与边独立集

7.3.1　边覆盖集

1. 边覆盖点、边覆盖集、边覆盖数

覆盖与**边覆盖集**：设无向图为 $G(V, E)$，边的集合 $E^* \subseteq E$，若对于 $\forall v \in V$，$\exists e \in E^*$，使得：v 与 e 相关联，则称 e **覆盖** v，并称 E^*为**边覆盖集**(Edge Covering Set)，或简称**边覆盖**。

在图 7.8(a)中，取 $E^* = \{e_1, e_4, e_7\}$，则 E^*就是图 G 的一个边覆盖集，因为图 G 中每个顶点都被 E^*中某条边“覆盖”住了。

通俗地讲，所谓边覆盖集 E^*，就是 G 中所有的顶点都是 E^*中某条边的邻接顶点(边覆盖顶点)。

注意：在无向图中存在用尽可能少的边去“覆盖”住所有顶点的问题，所以边覆盖集有极小和最小的概念。最大边覆盖集的概念是没有意义的，因为对任何一个无向图 $G(V, E)$，取 $E^* = E$，总是满足边覆盖集的定义。

极小边覆盖：若边覆盖 E^*的任何真子集都不是边覆盖，则称 E^*是极小边覆盖。

最小边覆盖：边数最少的边覆盖集称为最小边覆盖。

边覆盖数(Edge Covering Number)：最小的边覆盖所含的边数称为边覆盖数，记作 $\alpha_1(G)$，或简记为 α_1。

在图 7.8(a)中，$\{e_1, e_4, e_7\}$和$\{e_2, e_5, e_6, e_7\}$都是极小边覆盖，$\{e_1, e_4, e_7\}$是最小边覆盖，因此 $\alpha_1 = 3$。

在图 7.8(b)中，$\{e_1, e_3, e_6\}$和$\{e_2, e_4, e_8\}$都是极小边覆盖，也都是最小边覆盖，因此 $\alpha_1=3$。

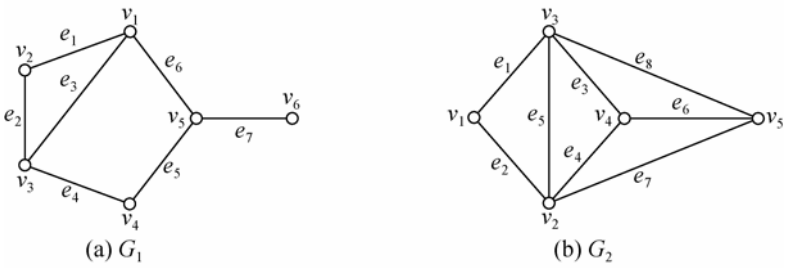


图 7.8　边覆盖集

2. 应用例子

例 7.5　应用边覆盖集安排 ACM 竞赛题目的讲解

某高校举办了一次全校 ACM 程序设计个人赛，共出了 8 道题目(P_1～P_8)。赛后组委会要求比赛前 10 名学生(s_1～s_{10})来讲解题目，并要求他们每人选择两道题目来讲解，比如 s_1 学生准备讲解 P_2 和 P_8；s_2 学生准备讲解 P_6 和 P_8 等。要讲解这 8 道题目至少需要多少名学生。

以题目 P_1～P_8 为顶点，如果某学生 s_k 准备讲解题目 P_i 和 P_j，则在顶点 P_i 和 P_j 之间连一条边 s_k，这样构造的无向图如图 7.9 所示。本题要求解的是讲解这 8 道题目至少需要多少名学生，转换成求图 7.9 的最小边覆盖集。在本题中，边 s_k“覆盖住”顶点 P_i 含义是学生 s_k 讲解题目 P_i。图 7.9 中粗线所示的边构成了一个最小边覆盖集，共 5 条边，因此至少需要 5 名学生来讲解，很明显某些学生只能讲一道题目。

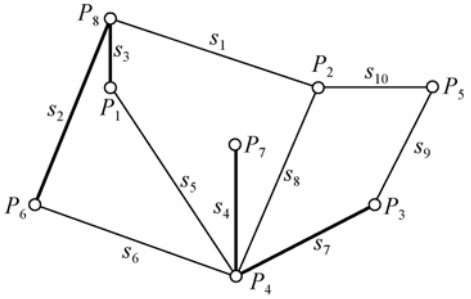


图 7.9　ACM 题目讲解

7.3.2　边独立集(匹配)

1. 边独立集、边独立数

边独立集(匹配)：设无向图为 $G(V, E)$，边的集合 $E^* \subseteq E$，若 E^*中任何两条边均不相邻，则称 E^*为 G 的**边独立集**(Edge Independent Set)，也称 E^*为 G 的**匹配**(Matching)。所谓任何

两条边均不相邻，通俗地讲，就是任何两条边都没有公共顶点。

例如，在图 7.10(a)中，取 $E^* = \{e_1, e_4, e_7\}$，则 E^*就是图 G_1 的一个边独立集，因为 E^*中每两条边都没有公共顶点。

注意：在无向图中存在将尽可能多的、相互独立的边包含到边的集合 E^*中的问题，所以边独立集有极大和最大的概念。最小边独立集的概念是没有意义的，因为对任何一个无向图 $G(V, E)$，取 $E^* = \phi$(空集)，总是满足边独立集的定义。

极大匹配：若在 E^*中加入任意一条边所得到的集合都不匹配，则称 E^*为极大匹配。

最大匹配：边数最多的匹配称为最大匹配。

边独立数(Edge Independent Number)：最大匹配的边数称为边独立数或匹配数，记作 β_1(G)，简记为 β_1。

在图 7.10(a)中，$\{e_2, e_6\}$、$\{e_3, e_5\}$和$\{e_1, e_4, e_7\}$都是极大匹配，$\{e_1, e_4, e_7\}$是最大匹配，因此 $\beta_1 = 3$。

在图 7.10(b)中，$\{e_1, e_3\}$、$\{e_2, e_4\}$和$\{e_4, e_7\}$都是极大匹配，也都是最大匹配，因此 $\beta_1 = 2$。

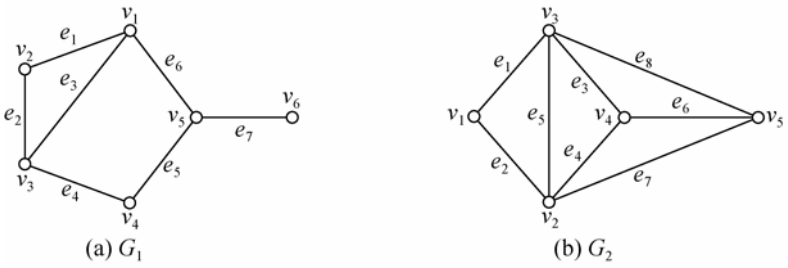


图 7.10　边独立集

以下几个概念都是针对无向图 $G(V, E)$中一个给定的匹配 M 而言的。

在无向图 G 中，若边$(u, v) \in M$，则称**顶点 u 与 v 被 M 所匹配**。

盖点与未盖点：设 v 是图 G 的一个顶点，如果 v 与 M 中的某条边关联，则称 v 为 M 的**盖点**(有的文献上也称为 ***M* 饱和点**)。如果 v 不与任意一条属于匹配 M 的边相关联，则称 v 是匹配 M 的**未盖点**(相应地，有的文献上也称为**非 *M* 饱和点**)。所谓盖点，就是被匹配中的边盖住了，而未盖点就是没有被匹配 M 中的边“盖住”的顶点。

例如，在图 7.11(a)所示的无向图中，取定 $M = \{e_1, e_4\}$，M 中的边用粗线标明，则顶点 v_1 与 v_2 被 M 所匹配；v_1、v_2、v_3 和 v_4 是 M 的盖点，v_5 和 v_6 是 M 的未盖点。

而在图 7.11(b)中，取定 $M = \{e_1, e_4, e_7\}$，则 G 中不存在未盖点。

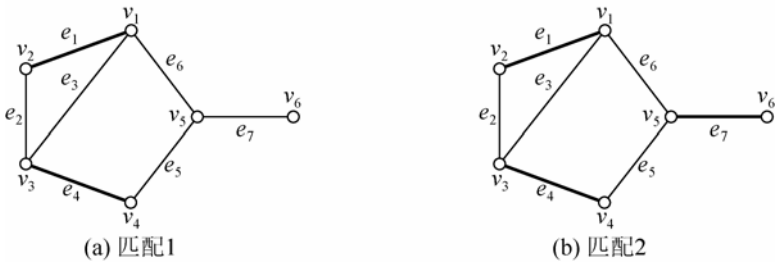


图 7.11　盖点与未盖点

2. 应用例子

例 7.6 飞行员搭配问题 1——**最大匹配**问题

飞行大队有若干个来自各地的飞行员，专门驾驶一种型号的飞机，每架飞机有两个飞行员。由于种种原因，例如互相配合的问题，有些飞行员不能在同一架飞机上飞行，问如何搭配飞行员，才能使出航的飞机最多。

为简单起见，设有 10 个飞行员，图 7.12 中的 $v_1, v_2, \cdots, v_{10}$ 就代表这 10 个飞行员。如果两个人可以同机飞行，就在他们之间连一条线，否则就不连。

图 7.12 中的 3 条粗线代表一种搭配方案。由于一个飞行员不能同时派往两架飞机，因此任何两条粗线不能有公共端点。因此该问题就转化为：如何找一个包含最多边的匹配，这个问题就是图的最大匹配问题。(思考，图 7.12 中粗线所示的匹配是最大匹配吗？)

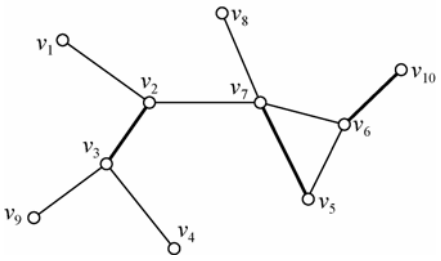


图 7.12 飞行员搭配问题 1

7.3.3 最大边独立集(最大匹配)与最小边覆盖集之间的联系

从最大匹配出发可以构造最小边覆盖，从最小边覆盖出发也可以构造最大匹配。通常，可以按如下方法进行。

(1) 从最大匹配出发，通过增加关联未盖点的边获得最小边覆盖。

(2) 从最小边覆盖出发，通过移去相邻的一条边获得最大匹配。

(详见定理 7.4。)

任取一个最大匹配，例如在图 7.13(a)中，取匹配 $M = \{ e_2, e_4 \}$，M 中的边用粗线标明，则 $M \cup \{ e_6 \}$、$M \cup \{ e_8 \}$、$M \cup \{ e_7 \}$都是图的最小边覆盖，其中顶点 v_5 是 M 的未盖点，而边 e_6、e_8、e_7 都与 v_5 关联。

任取一个最小边覆盖，例如在图 7.13(b)中，取最小边覆盖 $W = \{ e_1, e_3, e_6 \}$，W 中的边用粗线标明，从中移去一条相邻的边，如去掉 e_6，则$\{ e_1, e_3 \}$是最大匹配；去掉 e_3，则$\{ e_1, e_6 \}$是最大匹配。

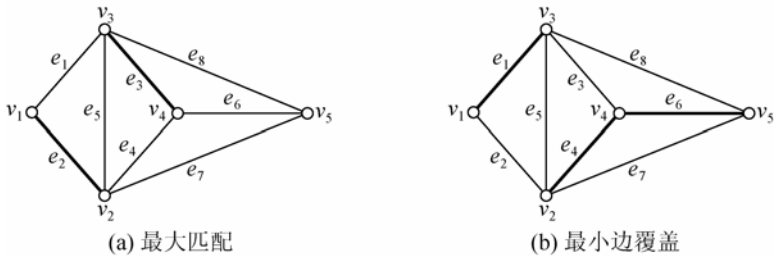


图 7.13 最大匹配与最小边覆盖之间的联系

定理 7.4　设无向图 G 的顶点个数为 n，且 G 中无孤立点。

(1) 设 M 为 G 的一个最大匹配，对于 G 中 M 的每个未盖点 v，选取一条与 v 关联的边所组成的边的集合为 N，则 $W = M \cup N$ 为 G 中的最小边覆盖。

(2) 设 W_1 为 G 的最小边覆盖，若 G 中存在相邻的边就移去其中的一条，设移去的边集为 N_1，则 $M_1 = W_1 - N_1$ 为 G 中一个最大匹配。

(3) G 中边覆盖数 α_1 与匹配数 β_1，满足：$\alpha_1 + \beta_1 = n$，即：边覆盖数+边独立数= n。

(证明略。)

7.4　匹 配 问 题

匹配问题是图论中一类常见的问题。7.3.2 节介绍了匹配问题的一个例子，接下来再看一个实例。

例 7.7　飞行员搭配问题 2——**二部图的最大匹配**问题

在例 7.6 中，如果飞行员分成两部分，一部分是正驾驶员，一部分是副驾驶员。如何搭配正副驾驶员才能使得出航飞机最多的问题可以归结为一个二部图上的最大匹配问题。

例如，假设有 4 个正驾驶员，有 5 个副驾驶员，飞机必须要有一名正驾驶员和一名副驾驶员才能起飞。正驾驶员和副驾驶员之间存在搭配的问题。

在图 7.14(a)中，x_1, x_2, x_3, x_4 表示 4 个正驾驶员，y_1, y_2, y_3, y_4, y_5 表示 5 个副驾驶员，如图 7.14(a)所示。正驾驶员之间不能搭配，副驾驶员之间也不能搭配，所以这是一个二部图。图 7.14(b)中的 4 条粗线代表一种搭配方案。这个问题实际上是求一个二部图的最大匹配。

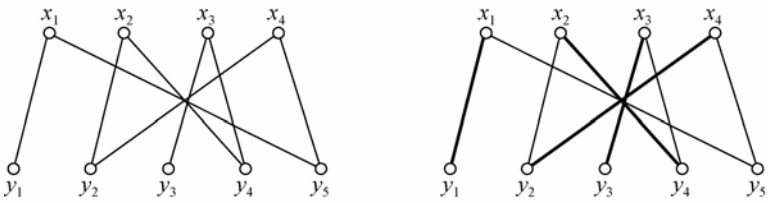


图 7.14　飞行员搭配问题 2

下面对匹配问题作进一步的探讨，7.5 节将讨论二部图最大匹配的求解算法。

7.4.1　完美匹配

完美匹配：对于一个图 G 与给定的一个匹配 M，如果图 G 中不存在 M 的未盖点，则称匹配 M 为图 G 的完美匹配。

例如，图 7.15(a)所示的无向图，取 $M = \{ e_1, e_4, e_7 \}$，则 M 是 G 的一个完美匹配，同时 M 也是图 G 的最大匹配及最小边覆盖。

而在图 7.15(b)中，不可能有完美匹配，因为对任何匹配都存在未盖点。

定理 7.4 的推论　设 G 中顶点个数为 n，且 G 中无孤立顶点，M 为 G 中的匹配，W 是 G 中的边覆盖，则|M|≤|W|，|M|表示 M 中边的数目。当等号成立时，M 为 G 中完美匹配，W 为 G 中最小边覆盖。

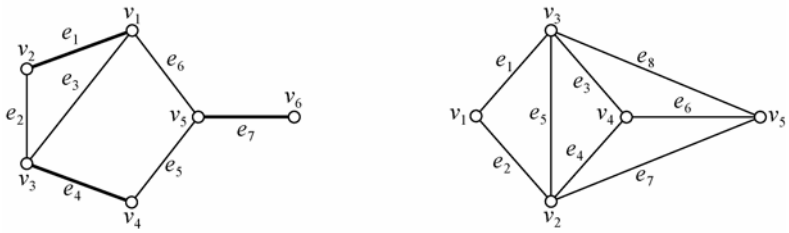


图 7.15　完美匹配

7.4.2　二部图的完备匹配与完美匹配

二部图的完备匹配：设无向图 $G(V, E)$为二部图，它的两个顶点集合为 X 和 Y，且$|X|\leqslant|Y|$，M 为 G 中的一个最大匹配，且$|M| = |X|$，则称 M 为 X 到 Y 的**二部图 G 的完备匹配**。若$|X| = |Y|$，则该完备匹配覆盖住 G 的所有顶点，所以该完备匹配也是**完美匹配**。

例如，如图 7.16 所示的 3 个二部图中，图 7.16(a)和图 7.16(b)中取定的匹配 M 都是完备匹配，而图 7.16(c)中不存在完备匹配。

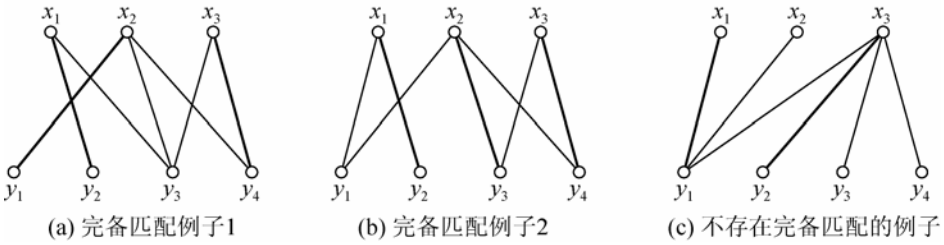


图 7.16　二部图的完备匹配

二部图完备匹配的一个应用例子是：某公司有工作人员 $x_1, x_2, \cdots, x_m$，他们去做工作 $y_1, y_2, y_3, \cdots, y_n$，$n>m$，每个人适合做其中一项或几项工作，问能否恰当地安排使得每个人都分配到一项合适的工作。

7.4.3　最佳匹配

继续对上面的应用例子进行深化：工作人员可以做各项工作，但效率未必一致，现在需要制定一个分工方案，使公司的总效益最大，这就是最佳分配问题。

二部图的最佳匹配：设 $G(V, E)$为加权二部图，它的两个顶点集合分别为 $X = \{ x_1, x_2, \cdots, x_m \}$、$Y = \{ y_1, y_2, \cdots, y_n \}$。$W(x_i, y_k) \geqslant 0$ 表示工作人员 x_i 做工作 y_k 时的效益，权值总和最大的完备匹配称为**二部图的最佳匹配**。

7.4.4　匹配问题求解的基本概念及思路

1. 交错轨

交错轨：设 P 是图 G 的一条轨(即路径)，M 是图 G 中一个给定的匹配，如果 P 的任意两条相邻的边一定是一条属于匹配 M 而另一条不属于 M，则称 P 是关于 M 的一条**交错轨**。

例如，在图 7.17(a)所示的图中，取定 $M = \{ e_4, e_6, e_{10} \}$，则图 7.17(b)、7.17(c)所示的路

径都是交错轨。

特别地，如果轨 P 仅含一条边，那么无论这条边是否属于匹配 M，P 一定是一条交错轨。

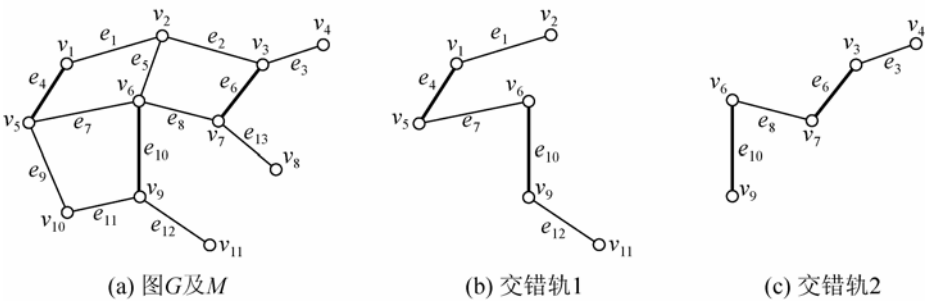


图 7.17　交错轨

2. 可增广轨

可增广轨：对于一个给定的图 G 和匹配 M，两个端点都是未盖点的交错轨称为关于 M 的**可增广轨**。

例如，图 7.17(b)所示的交错轨的两个端点 v_2、v_{11} 都是匹配 M 的未盖点，所以这条轨是可增广轨，而图 7.17(c)所示的交错轨不是可增广轨。

特别地，如果两个未盖点之间仅含一条边，那么单单这条边也组成一条可增广轨。

可增广轨的含义。对于图 G 的一个匹配 M 来说，如果能找到一条可增广轨 P，那么这个匹配 M 一定可以通过下述方法改进成一个多包含一条边的匹配 Ms(即匹配 M 扩充了)：把 P 中原来属于匹配 M 的边从匹配 M 中去掉(粗边改成细边)，而把 P 中原来不属于 M 的边加到匹配 Ms 中去(细边改成粗边)，变化后的匹配 Ms 恰好比原匹配 M 多一条边。

例如，对图 7.17(a)中 G 的一个匹配 M，找到图 7.18(a)所示的一条可增广轨，那么按照前面所述的方法可以将原匹配进行扩充，得到图 7.18(b)所示的新匹配 Ms，Ms 比 M 多一条边。

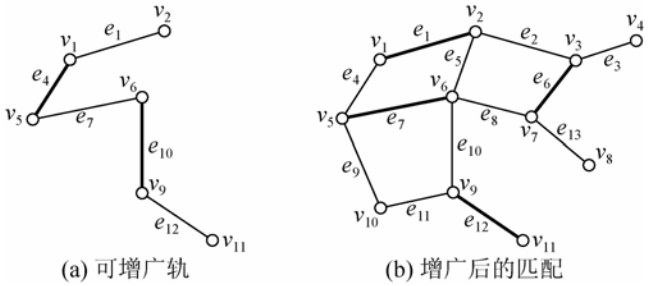


图 7.18　可增广轨及通过可增广轨扩展匹配

3. 求最大匹配的可行方法

定理 7.5　M 为 G 的最大匹配，当且仅当 G 不存在关于 M 的可增广轨。

因此，求最大匹配的一个可行方法如下。

给定一个初始匹配 M(如果没有给定，则 $M=\phi$)，如果图 G 没有未盖点，则肯定不会有可增广轨了，即 M 就是最大匹配；否则对图 G 的所有未盖点 v_i，通过一定的方法搜索以

v_i为端点的可增广轨，从而通过可增广轨逐渐把 M 扩大。(在扩大 M 的过程当中，某些未盖点会逐渐被 M 盖住)

7.5 二部图最大匹配问题的求解

求二部图最大匹配的算法有：①网络流解法；②匈牙利算法；③Hopcroft-Karp 算法(匈牙利算法的改进)。

本节将介绍这几种算法，并通过例题详细介绍算法的实现方法。

7.5.1 网络流解法

1. 基本思路

设二部图为 $G(V, E)$，它的顶点集合 V 所包含的两个子集为 $X = \{ x_1, x_2, \cdots, x_m \}$和 $Y = \{ y_1, y_2, \cdots, y_n \}$，如图 7.19(a)所示。如果把二部图中看成一个网络，边(x_i, y_k)都看成有向边$<x_i, y_k>$，则在求最大匹配时要保证从顶点 x_i 发出的边最多只选一条，进入顶点 y_k 的边最多也只选一条，在这些前提下将尽可能多的边选入到匹配中来。

设想有一个源点 S，控制从 S 到 x_i 的弧$<s, x_i>$的容量为 1，这样就能保证从顶点 x_i 发出的边最多只选一条。同样，设想有一个汇点 T，控制从顶点 y_k 到 T 的弧$<y_k, t>$的容量也为 1，这样就能保证进入顶点 y_k 的边最多也只选一条。另外，设边$<x_i, y_k>$的容量也为 1。

按照上述思路构造好容量网络后，任意一条从 S 到 T 的路径，一定具有 S–x_i–y_k–T 的形式，且这条路径上 3 条弧$<s, x_i>$、$<x_i, y_k>$、$<y_k, t>$的容量均为 1。因此，该容量网络的最大流中每条从 S 到 T 的路径上，中间这一条边$<x_i, y_k>$的集合就构成了二部图的最大匹配。

2. 网络流的构造及求解

求二部图最大匹配的容量网络构造和求解方法如下。

(1) 从二部图 G 出发构造一个容量网络 G'，步骤如下。

① 增加一个源点 S 和汇点 T。

② 从 S 向 X 的每一个顶点都画一条有向弧，从 Y 的每一个顶点都向 T 画一条有向弧。

③ 原来 G 中的边都改成有向弧，方向是从 X 的顶点指向 Y 的顶点。

④ 令所有弧的容量都等于 1。构造好的容量网络如图 7.19(b)所示。

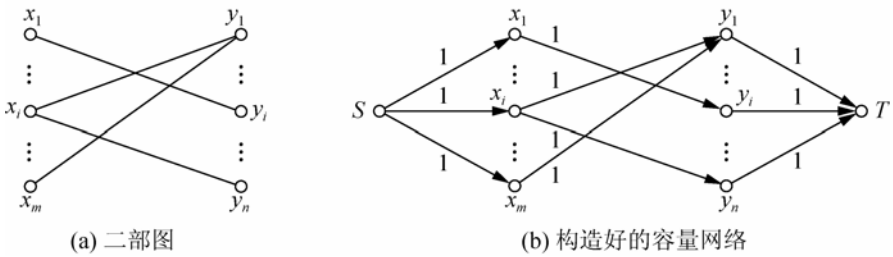


图 7.19 二部图最大匹配的网络流解法：网络流的构造

(2) 求容量网络 G' 的最大流 F。

(3) 最大流 F 求解完毕后，从 X 的顶点指向 Y 的顶点的弧集合中，弧流量为 1 的弧对应二部图最大匹配中的边，最大流 F 的流量对应二部图的最大匹配的边数。

为什么这样构造的容量网络求出来的最大流就是最大匹配？这是因为：①网络中所有的弧容量均为 1，这样原二部图 G 中的边，要么选择(流量为 1)，要么不选择；②尽管在网络中顶点 x_i 可能发出多条边，但在最大流中只能选择一条边，因为从源点 S 流入顶点 x_i 的流量为 1；③尽管在网络中可能有多条边进入顶点 y_k，但在最大流中只能选择一条边，因为从顶点 y_k 流入汇点 T 的流量为 1。

以上第②、③点保证了最大流 F 中属于二部图的边不存在共同顶点。

3. 网络流解法实例

设有 5 位待业者，用 x_1, x_2, x_3, x_4, x_5 表示；另外有 5 项工作，用 y_1, y_2, y_3, y_4, y_5 表示；如果 x_i 能胜任 y_i 工作，则在他们之间连一条边。图 7.20(a)描述了这 5 位待业者各自能胜任工作的情况，很明显，这是一个二部图。现在要求设计一个就业方案，使尽量多的人能就业。这是求二部图最大匹配的问题。

按照前面描述的方法构造网络流：在二部图中增加两个顶点 S 和 T，分别作为源点、汇点；并用有向边把它们与原二部图中顶点相连，令全部边上的容量均为 1，如图 7.20(b)所示。当网络流达到最大时，如果在最大流中弧<x_i, y_j>上的流量为 1，就让 x_i 作 y_j 工作，此即为最大匹配方案。图 7.20(c)是求网络最大流的结果。在图 7.20(d)中，粗线所表示的边就是求得的最大匹配，x_1, x_2, x_3, x_4 分别做 y_2, y_1, y_4, y_5 工作，故最多可安排 4 个人工作。

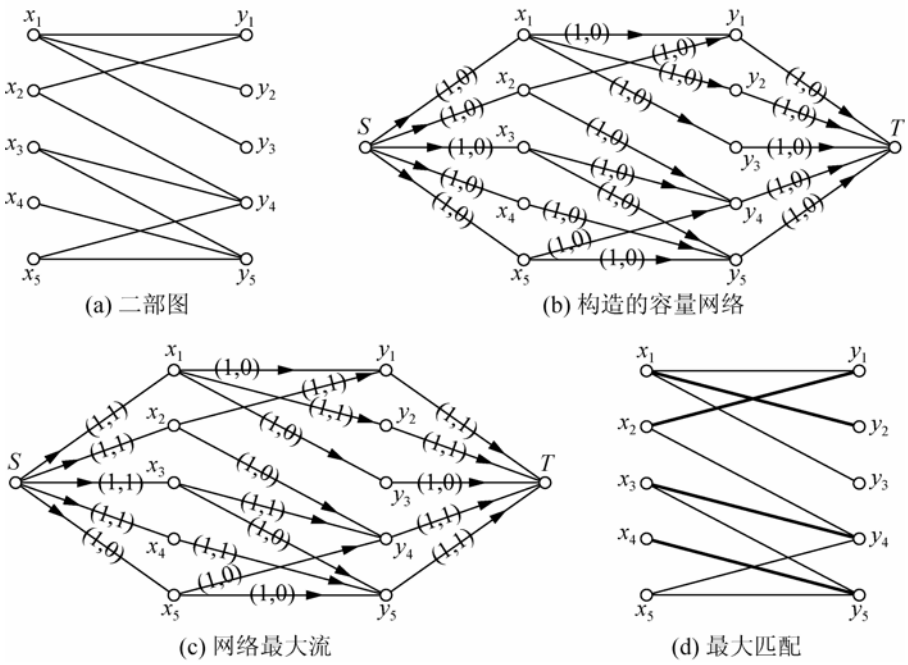


图 7.20　二部图最大匹配的网络流解法实例

7.5.2 匈牙利算法

匈牙利算法的原理为：从当前匹配 M(如果没有匹配，则取初始匹配为 $M=\phi$)出发，检查每一个未盖点，然后从它出发寻找可增广路，找到可增广路，则沿着这条可增广路进行扩充，直到不存在可增广路为止。

根据从未盖点出发寻找可增广路搜索的方法，可以分为：①DFS 增广；②BFS 增广；③多增广路(Hopcroft-Karp 算法)，本书没有介绍，请参考其他教材。

在算法中用到的一些变量及其含义如下。

```
#define MAXN 10          //MAXN 为表示 X 集合和 Y 集合顶点个数最大值的符号常量
int nx, ny;              //X 和 Y 集合中顶点的个数
int g[MAXN][MAXN];       //邻接矩阵，g[i][j]为 1 表示 Xi 和 Yj 有边相连
//cx[i]表示最终求得的最大匹配中与 Xi 匹配的 Y 顶点，cy[i]同理
int cx[MAXN], cy[MAXN];
```

1. DFS 增广

采用 DFS 思想搜索可增广路并求最大匹配的代码如下。

```
//DFS 算法中记录顶点访问状态的数组，mk[i] = 0 表示未访问过，为 1 表示访问过
int mk[MAXN] ;
//从 X 集合中的顶点 u 出发，用深度优先的策略寻找增广路
//(这种增广路只能使当前的匹配数增加 1)
int path( int u )
{
    for( int v=0 ; v<ny ; v++ )     //考虑所有 Yi 顶点 v
    {
        if( g[u][v] && !mk[v] )//v 与 u 邻接，且没有访问过
        {
            mk[v]=1;     //访问 v
            //如果 v 没有匹配，或者 v 已经匹配了，但从 cy[v]出发可以找到一条增广路
//注意如果前一个条件成立，则不会递归调用
            if( cy[v]==-1||path( cy[v] ) )
            {
                cx[u]=v;     //把 v 匹配给 u
                cy[v]=u;     //把 u 匹配给 v
                return 1;    //找到可增广路
            }
        }
    }
    return 0 ; //如果不存在从 u 出发的增广路
}
int MaxMatch( ) //求二部图最大匹配的匈牙利算法
{
    int res=0;  //所求得的最大匹配
    memset(cx,0xff,sizeof(cx)); //从 0 匹配开始增广,将 cx 和 cy 各元素初始化为-1
    memset( cy, 0xff, sizeof(cy) );
    for( int i=0; i<=nx; i++ )
    {
        if( cx[i]==-1 ) //从每个未盖点出发进行寻找增广路
```

```
            {
                memset( mk, 0, sizeof(mk) ) ;
                res+=path(i); //每找到一条增广路，可使得匹配数加 1
            }
        }
        return res;
    }
```

接下来以图 7.21(a)所示的二部图为例解释匈牙利算法求解过程(DFS 增广)。在图 7.21(b)中，从顶点 x_1 出发进行 DFS 搜索后，发现 y_2 跟 x_1 邻接且没有匹配，所以将 x_1 匹配给 y_2，从 x_1 出发的搜索过程结束。图 7.21(c)～7.21(e)为从顶点 x_2 出发进行 DFS 搜索的过程，在图 7.21(c)中，发现 y_2 跟 x_2 邻接但已经匹配给 x_1 了，所以递归地从 x_1 出发进行 DFS 搜索，从而找到下一个跟 x_1 邻接且没有匹配的顶点 y_3，从而将 x_1 改为匹配给 y_3 并返回；返回到 x_2 的搜索过程后，将空出来的 y_2 匹配 x_2，如图 7.21(d)所示，至此从 x_2 出发的搜索过程结束。在图 7.21(f)中，将 x_3 匹配给 y_3。至此，算法求解结束，求得最大匹配为图 7.21(f)，匹配数为 3。

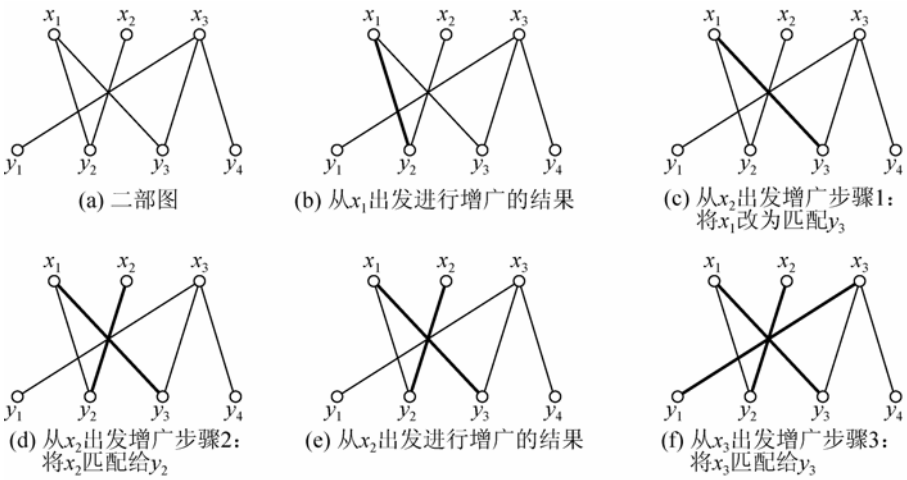


图 7.21　匈牙利算法求解过程(DFS 增广)

DFS 增广的特点如下。

(1) 优点：实现简洁，理解容易。

(2) 适用：稠密图，由于边多，DFS 找增广路很快。

(3) 复杂度：$O(n^3)$。

2. BFS 增广

采用 BFS 思想搜索可增广路并求最大匹配的代码如下。

```
int pred[MAXN];     //是用来记录交错轨的，同时也用来记录 Y 集合中的顶点是否遍历过
int queue[MAXN];    //实现 BFS 搜索用到的队列(用数组模拟)
int MaxMatch( )
{
    int i, j, y;
    int cur, tail;                          //表示队列头和尾位置的下标
```

```
    int res=0;                              //所求得的最大匹配数
    memset( cx , 0xff , sizeof(cx) );  //初始化所有点为未被匹配的状态
    memset( cy , 0xff , sizeof(cy) );
    for( i=0; i<nx; i++ )
    {
        if( cx[i] != -1 )  continue;
        //对 X 集合中的每个未盖点 i 进行一次 BFS 找交错轨
        for( j=0; j<ny; j++ )  pred[j]=-2;  //-2 表示初始值
        cur=tail=0;                 //初始化 BFS 的队列
        for( j=0; j<ny; j++ )       //把 i 的邻接点顶都入队列
        {
            if( g[i][j] )
            {
                pred[j]=-1; queue[tail++]=j; //-1 表示遍历到，是邻接顶点
            }
        }
        while( cur<tail )  //BFS
        {
            y=queue[cur];
            if( cy[y]==-1 )  break; //找到一个未被匹配的点，则找到了一条交错轨
            cur++;
            //y 已经被匹配给 cy[y]了，从 cy[y]出发，将它的邻接顶点入队列
            for( j=0; j<ny; j++ )
            {
                if( pred[j]==-2&&g[cy[y]][j] )
                {
                    pred[j]=y;
                    queue[tail++]=j;
                }
            }
        }
        if( cur==tail )  continue;  //没有找到交错轨
        while( pred[y]>-1 )         //更改交错轨上匹配状态
        {
            cx[ cy[ pred[y] ] ]=y;
            cy[y]=cy[ pred[y] ];
            y=pred[y];
        }
        cy[y]=i;  cx[i]=y;
        res++;  //匹配数加 1
    }
    return res;
}
```

接下来以图 7.22(a)所示的二部图为例解释匈牙利算法求解过程(BFS 增广)。在图 7.22(c)中，x_2 唯一的邻接顶点 y_2 已经匹配给 x_1 了，这是从 x_1 出发进行 BFS 搜索，将 x_1 的所有邻接顶点入队列，从而找到一条增广路 $x_2 \rightarrow y_2 \rightarrow x_1 \rightarrow y_3$。在图 7.22(d)中更改交错轨上的匹配状态，使得匹配数增加 1。最终求得的最大匹配如图 7.22(f)所示，匹配数为 3。

这种方法的特点如下。

(1) 适用：稀疏二部图，边少，增广路短。

(2) 复杂度：$O(n^3)$。

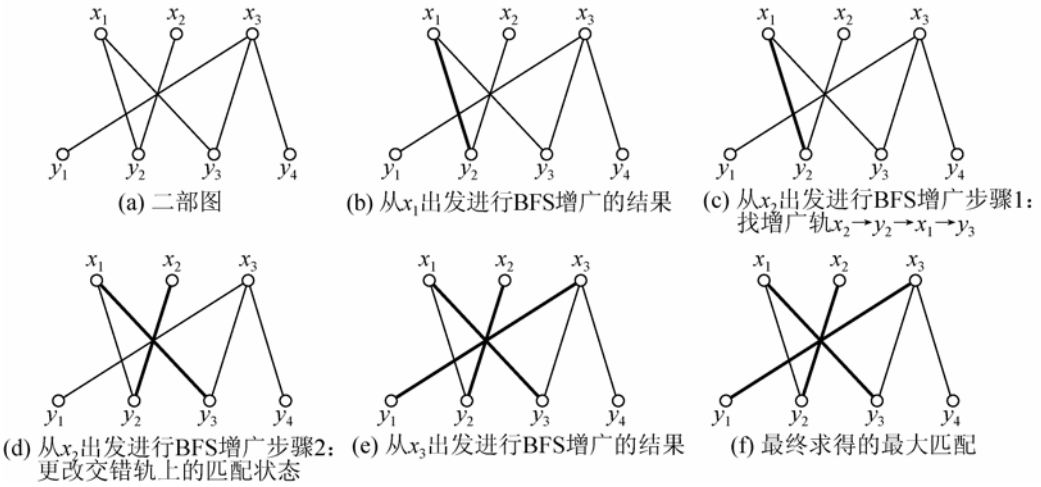


图 7.22　匈牙利算法求解过程(BFS 增广)

7.5.3　例题解析

以下通过 4 道例题的分析，详细介绍以上算法在求解二部图最大匹配的思想及其实现方法。

例 7.8　放置机器人(Place the Robots)

题目来源：

ZOJ Monthly, October 2003, ZOJ1654

题目描述：

Robert 是一个著名的工程师。一天，他的老板给他分配了一个任务。任务的背景是：给定一个 $m \times n$ 大小的地图，地图由方格组成，在地图中有 3 种方格——墙、草地和空地，他的老板希望能在地图中放置尽可能多的机器人。每个机器人都配备了激光枪，可以同时向 4 个方向(上、下、左、右)开枪。机器人一直待在最初始放置的方格处，不可移动，然后一直朝 4 个方向开枪。激光枪发射出的激光可以穿透草地，但不能穿透墙壁。机器人只能放置在空地。当然，老板不希望机器人互相攻击，也就是说，两个机器人不能放在同一行(水平或垂直)，除非他们之间有一堵墙隔开。

给定一张地图，程序需要输出在该地图中可以放置的机器人的最大数目。

输入描述：

输入文件的第一行为一个整数 T，$T \leqslant 11$，代表输入文件中测试数据的数目。对每个测试数据，第 1 行为两个整数，m 和 n，$1 \leqslant m, n \leqslant 50$，分别代表地图的行和列的数目。接下来有 m 行，每行有 n 个字符，每个字符都是'#', '*', 或 'o'，分别代表墙壁、草地、空地。

输出描述：

对每个测试数据，首先在第 1 行输出该测试数据的序号，格式为："Case :id"，其中 id 是测试数据的序号，从 1 开始计数。第 2 行输出在该地图中可以放置的机器人最大数目。

样例输入：

```
2
5 5
o***#
*###*
oo#oo
***#o
#o**o
4 4
o***
*###
oo#o
***o
```

样例输出：

```
Case :1
4
Case :2
3
```

分析：

样例输入中两个测试数据所描述的地图如图 7.23(a)和 7.23(b)所示。在图 7.23(a)中，最多可以放置 4 个机器人，一种放置方案是在(0, 0)、(2, 1)、(2, 3)和(3, 4) 4 个位置(行、列位置序号从 0 开始计起)上放置机器人。在图 7.23(b)中，最多可以放置 3 个机器人，一种放置方案是在(0, 0)、(2, 1)和(2, 3) 3 个位置上放置机器人。

在问题的原型中，草地、墙这些信息不是本题所关心的，本题关心的只是空地和空地之间的联系。因此，很自然想到了下面这种简单的模型：以空地为顶点，在有冲突的空地间连边。

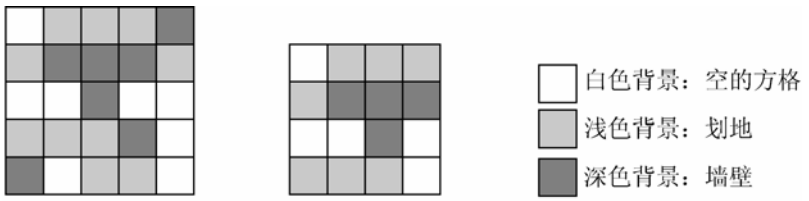


图 7.23　放置机器人：两个测试数据所描述的地图

这样对图 7.23(a)所示的地图，把所有的空地用数字标明，如图 7.24(a)所示；把所有有冲突的空地间用边连接后得到图 7.23(b)。于是，问题转化为求图的最大独立集问题：求最大顶点集合，集合中所有顶点互不连接(即互不冲突)。但是最大点独立集问题是一个 NP 问题，没有有效的算法能求解。

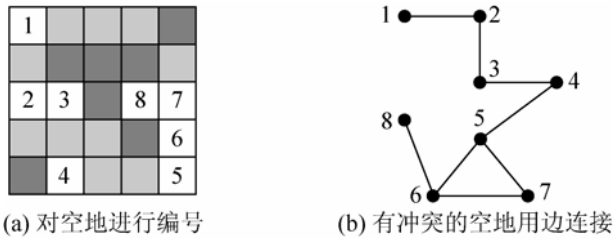


图 7.24　放置机器人——模型一：最大点独立集问题

将每一行被墙隔开且包含空地的连续区域称作“块”。显然，在一个块之中，最多只能放一个机器人。把这些块编上号，如图 7.25(a)所示。需要说明的是，最后一行，即第 4 行有两个空地，但这两个空地之间没有墙壁，只有草地，所以这两个空地应该属于同一“块”。

同样，把竖直方向的块也编上号，如图 7.25(b)所示。

把每个横向块看作二部图中顶点集合 X 中的顶点，竖向块看作集合 Y 中的顶点，若两个块有公共的空地(注意，每两个块最多有一个公共空地)，则在它们之间连边。例如，横向块 2 和竖向块 1 有公共的空地，即(2, 0)，于是在 X 集合中的顶点 2 和 Y 集合中的顶点 1 之间有一条边。这样，问题转化成一个二部图，如图 7.25(c)所示。

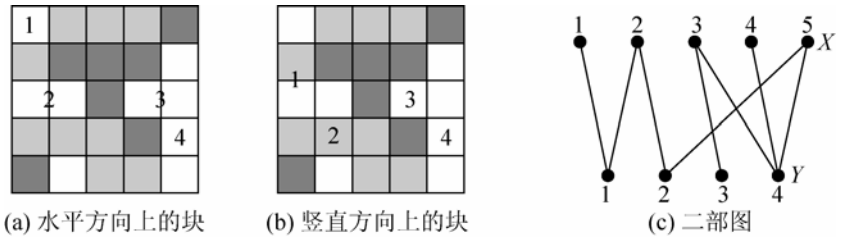


图 7.25　放置机器人——模型二：二部图最大匹配问题

由于每条边表示一个空地(即一个横向块和一个竖向块的公共空地)，有冲突的空地之间必有公共顶点。例如边(x_1, y_1)表示空地(0, 0)、边(x_2, y_1)表示空地(2, 0)，在这两个空地上不能同时放置机器人。所以问题转化为在二部图中找没有公共顶点的最大边集，这就是最大匹配问题。

接下来以 7.23(a)所示的地图为例解释二部图的构造过程：在下面的代码中，二维数组 *xs* 和 *ys* 分别用来给水平方向和垂直方向上“块”进行编号，编号的结果如图 7.26(b)、7.26(c)所示；二维数组 *g* 用来对水平方向上和垂直方向上的块进行连接，若两个块有公共的空地，则在它们之间连边，如果 *g*[*i*][*j*]==1，那么水平方向上的第 *i* 个块跟垂直方向上的第 *j* 个块有公共的空地，要连边；连边后的结果如图 7.26(d)所示；这样构造的二部图如图 7.26(e)所示。

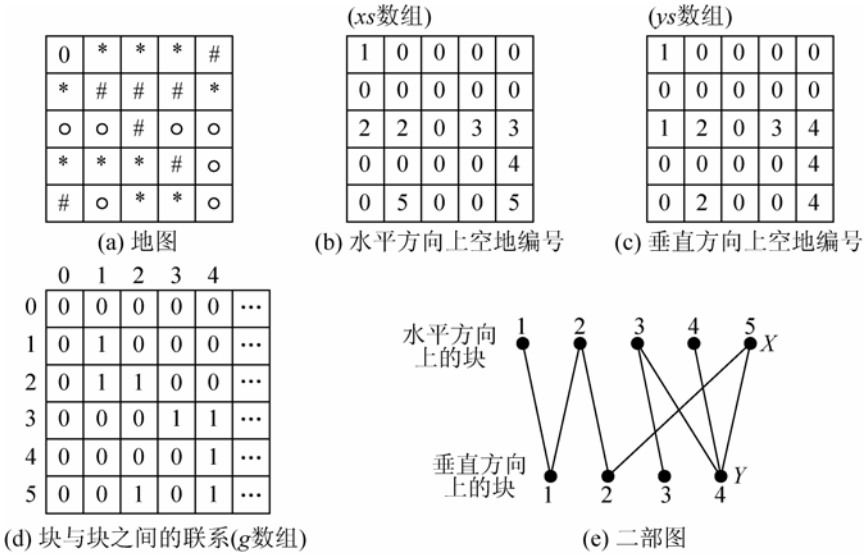
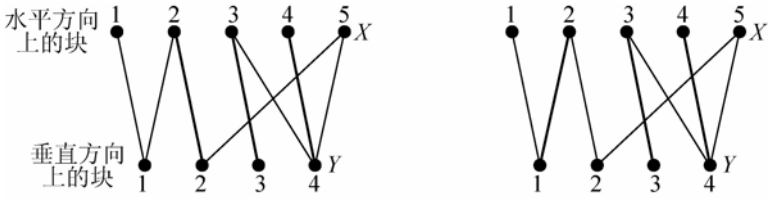


图 7.26　放置机器人：算法执行过程

如果从 0 匹配开始增广，求得的最大匹配如图 7.27(a)所示，粗线边代表匹配中的边。

假设从给定的一个初始匹配(如图 7.27(b)所示)出发进行增广(可以通过在求最大匹配前给 x、y 数组赋值来实现)，其过程为：从顶点 x_1 出发寻找可增广路，它的邻接顶点 y_1 已经匹配了，所以从 y_1 匹配顶点即 x_2 出发递归寻找可增广路，x_2 的邻接顶点 y_2 没有匹配，所以找到一条可增广路(x_1, y_1, x_2, y_2)；沿着这条可增广路，按照匈牙利算法可以使得当前匹配增加 1；改进后的匹配就是最大匹配了。



(a) 初始匹配为空，求得的最大匹配　　(b) 初始匹配不为空

图 7.27　放置机器人：最大匹配求解

代码如下：

```
#define MAX 51
int m, n;           //地图的大小 m×n,(1<=m, n<=50)
char map[MAX][MAX]; //地图
//x[i]表示与 Xi 匹配的 Y 顶点,y[i]表示与 Yi 匹配的 X 顶点
int x[MAX*MAX], y[MAX*MAX];
int xs[MAX][MAX], ys[MAX][MAX]; //水平方向上"块"的编号，垂直方向上"块"的编号
int xn, yn1;        //水平方向上的"块"个数、垂直方向上的"块"个数
//对水平方向上和垂直方向上的块进行连接(若两个块有公共的空地，则在它们之间连边)
//如果 g[i][j]==1,那么水平方向上的第 i 个块跟垂直方向上的第 j 个块有公共的空地
bool g[MAX*MAX][MAX*MAX];
//DFS 算法中记录顶点访问的状态，如果 t[i]=0 表示未访问过，如果为 1 表示访问过
int t[MAX*MAX];
//从 X 集合中的顶点 u 出发用深度优先策略寻找增广路
//(这种增广路只能使当前的匹配数增加 1)
bool path( int u )
{
    int v;
    for( v=1; v<=yn1; v++ )     //考虑所有 Yi 顶点
    {
        if( g[u][v] && !t[v] ) //v 跟 u 邻接并且 v 未访问过
        {
            t[v]=1;
            //如果 v 没有匹配,
            //或者如果 v 已经匹配了，但从 y[v]出发可以找到一条增广路
            if( !y[v]||path(y[v]) )
            {
                x[u]=v;  y[v]=u;    //把 v 匹配给 u, 把 u 匹配给 v
                return 1;
            }
        }
    }
```

```
    return 0;   //如果不存在从u出发的增广路
}
void MaxMatch( )    //求二部图的最大匹配算法
{
    int i, ans=0;   //最大匹配数
    memset(x, 0, sizeof(x) ); memset( y, 0, sizeof(y) ); //从0匹配开始增广
    for( i=1; i<=xn; i++ )
    {
        if( !x[i] ) //从每个未盖点出发进行寻找增广路
        {
            memset( t, 0, sizeof(t) );
            if( path(i) )   //每找到一条增广路，可使得匹配数加1
                ans++;
        }
    }
    printf( "%d\n", ans );
}
int main( )
{
    int k, kase;    //kase的个数
    int i, j;       //循环变量
    int number;     //用来对水平方向和垂直方向上的"块"进行编号的序号
    int flag;       //一个"块"开始的标志
    scanf( "%d", &kase );
    for( k=0; k<kase; k++ )
    {
        printf( "Case :%d\n", k+1 );
        scanf( "%d%d", &m, &n );    //读入地图大小
        memset( xs, 0, sizeof(xs) );  memset( ys, 0, sizeof(ys) );
        for( i=0; i<m; i++ )        //读入地图
        {
            scanf( "%s", map[i] );
        }
        number=0;   //用来对水平方向和垂直方向上的"块"进行编号的序号
        for( i=0; i<m; i++ )    //对水平方向上的块进行编号
        {
            flag=0;
            for( j=0; j<n; j++ )
            {
                if( map[i][j]=='o' )
                {
                    if( flag==0 )  number++;
                    xs[i][j]=number;  flag=1;
                }
                else if( map[i][j]=='#' )  flag=0;
            }
        }
        xn=number;  number=0;
        for( j=0; j<n; j++ )    //对垂直方向上的块进行编号
        {
            flag=0;
```

```
            for( i=0; i<m; i++ )
            {
                if( map[i][j]=='o' )
                {
                    if( flag==0 ) number++;
                    ys[i][j]=number;  flag=1;
                }
                else if( map[i][j]=='#' )  flag=0;
            }
        }
        yn1=number;
        memset( g, 0, sizeof(g) );
        for( i=0; i<m; i++ )
        {
            for( j=0; j<n; j++ )
            {
                //对水平方向上和垂直方向上的块进行连接
                //(若两个块有公共的空地，则在它们之间连边)
                if( xs[i][j] )  g[xs[i][j]][ys[i][j]]=1;
            }
        }
        MaxMatch( );
    }
    return 0;
}
```

例 7.9　机器调度(Machine Schedule)

题目来源：

Asia 2002, Beijing(Mainland China), ZOJ1364, POJ1325

题目描述：

众所周知，机器调度是计算机科学中非常典型的一个问题，已经被研究很长时间了。各种机器调度问题在以下方面差别很大：必须满足的约束条件，以及期望得到的调度时间表。现在考虑一个针对两台机器的机器调度问题。

假设有两台机器，A 和 B。机器 A 有 *n* 种工作模式，分别称为 mode_0, mode_1, …, mode_n-1。同样机器 B 有 *m* 种工作模式，分别为 mode_0, mode_1, …, mode_m-1。刚开始时，A 和 B 都工作在模式 mode_0。

给定 *k* 个作业，每个作业可以工作在任何一个机器的特定模式下。例如，作业 0 可以工作在机器 A 的模式 mode_3 或者机器 *B* 的 mode_4 模式；作业 1 可以工作在机器 A 的模式 mode_2 或者工作在机器 *B* 的模式 mode_4 等。因此，对作业 *j*，调度中的约束条件可以表述成一个 3 元组(*i*, *x*, *y*)，意思是作业 *i* 可以工作在机器 A 的 mode_x 模式或者机器 B 的 mode_y 模式。

很显然的是，为了完成所有的作业，必须时不时切换机器的工作模式，但不幸的是，机器工作模式的切换只能通过手动重启机器完成。试编写程序实现：改变机器的顺序，给每个作业分配合适的作业，使得重启机器的次数最少。

输入描述：

输入文件包含多个测试数据，每个测试数据的第一行为 2 个整数：*n*, *m* 和 *k*，其中 *n*、

$m < 100$，$k < 1\,000$。接下来有 k 行给出了 k 个作业的约束，每一行为一个三元组：i、x、y。

输入文件的最后一行为一个 0，表示输入结束。

输出描述：

对输入文件中的每个测试数据，输出一行，为一个整数，表示需要重启机器的最少次数。

样例输入：

```
5 5 10
0 1 1
1 1 2
2 1 3
3 1 4
4 2 1
5 2 2
6 2 3
7 2 4
8 3 3
9 4 3
0
```

样例输出：

```
3
```

分析：

首先构造二部图：把 A 的 n 个 mode 和 B 的 m 个 mode 看作图的顶点，如果某个任务可以在 A 的 mode_i 或 B 的 mode_j 上完成，则从 A_i 到 B_j 连接一条边，这样构造了一个二部图。

例如对题目样例输入中的测试数据，构造的二部图如图 7.28 所示。

原先机器 A 和机器 B 都是工作在模式_0，切换到机器 A 的模式_1，可以完成作业(0, 1, 1)、(1, 1, 2)、(2, 1, 3)、(3, 1, 4)，再切换到机器 A 的模式_2，可以完成作业(4, 2, 1)、(5, 2, 2)、(6, 2, 3)、(7, 2, 4)，最后切换到机器 B 的模式_3，可以完成作业(8, 3, 3)、(9, 4, 3)。所以需要手动重启机器 3 次。

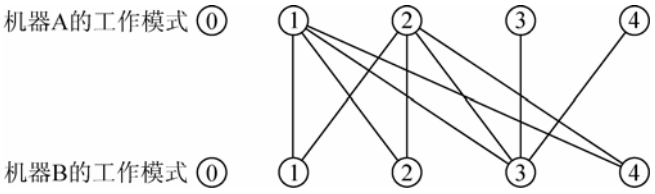


图 7.28　机器调度：二部图的构造

本题要求二部图的最小点覆盖集问题，即求最小的顶点集合，“覆盖”住所有的边。转换成求二部图的最大匹配问题，因为：二部图的点覆盖数 α_0 ＝匹配数 β_1。

另外，机器 A 和机器 B 最初工作在模式_0，所以对于那些可以工作在机器 A 的模式_0 或者机器 B 的模式_0 的作业，在完成这些作业时是不需要重启机器的。

代码如下：

```
#define maxn 105
int nx;                 //机器 A 的工作模式个数
int ny;                 //机器 B 的工作模式个数
int jobnum;             //作业个数
```

```
int g[maxn][maxn];      //所构造的二部图
int ans;                //最大匹配数
int sx[maxn],sy[maxn]; //path 函数所表示的 DFS 算法中用来标明顶点访问状态的数组
int cx[maxn], cy[maxn];//求得的匹配情况,X 集合中的顶点 i 匹配给 Y 集合中的顶点 cx[i]
//从 X 集合中的顶点 u 出发用深度优先的策略寻找增广路
//(这种增广路只能使当前的匹配数增加 1)
int path( int u )
{
    sx[u]=1;
    int v;
    //考虑所有 Yi 顶点(机器 A 和 B 最初工作在模式 0,
    //所以完成可以工作在模式 0 的作业时部需要重启机器)
    for( v=1; v<=ny; v++ )
    {
        if( (g[u][v]>0) && (!sy[v]) )  //v 跟 u 邻接并且 v 未访问过
        {
            sy[v]=1;
            //如果 v 没有匹配
            //或者如果 v 已经匹配了，但从 y[v]出发可以找到一条增广路
            if( !cy[v] || path(cy[v]) )
            {
                //在回退过程修改增广路上的匹配，从而可以使匹配数增加 1
                cx[u]=v;  cy[v]=u;  //把 v 匹配给 u, 把 u 匹配给 v
                return 1;
            }
        }
    }
    return 0;
}
int solve( )//求二部图的最大匹配算法
{
    ans=0;
    int i;
    memset( cx, 0, sizeof(cx) );  memset( cy, 0, sizeof(cy) );
    //机器 A 和 B 最初工作在模式 0,所以完成可以工作在模式 0 的作业时部需要重启机器
    for( i=1; i<=nx; i++ )
    {
        if( !cx[i] )
        {
            memset( sx, 0, sizeof(sx) );  memset( sy, 0, sizeof(sy) );
            ans+=path(i);
        }
    }
    return 0;
}
int main( )
{
    int i, j, k, m;
    while( scanf( "%d", &nx ) )
    {
        if( nx==0 )  break;
```

```
            scanf( "%d%d", &ny, &jobnum );
            memset( g, 0, sizeof(g) );
            for( k=0; k<jobnum; k++ )
            {
                scanf( "%d%d%d", &m, &i, &j );
                g[i][j]=1;  //构造二部图
            }
            solve( );
            printf( "%d\n", ans );
        }
        return 0;
    }
```

例 7.10　课程(Courses)

题目来源：

Southeastern Europe 2000, ZOJ1140, POJ1469

题目描述：

考虑 N 个学生和 P 门课程。每个学生见习 0、1 或多门课程。试判断是否能从这些学生当中选出 P 名学生，组成一个委员会，并同时满足以下条件。

(1) 委员会中的每名学生代表一门不同的课程(如果一名学生见习了某门课程，则他/她可以代表这门课程)。

(2) 每门课程在委员会中有一名代表。

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，代表输入文件中测试数据的数目。接下来就是 T 个测试数据。每个测试数据的格式如下。

P N

Count1 Student1_1_Student1_2 ... Student1_Count1

Count2 Student2_1_Student2_2 ... Student2_Count2

……

CountP StudentP_1 StudentP_2 ... StudentP_CountP

也就是说，每个测试数据的第 1 行为两个正整数 P 和 N，其中 P(1≤P≤100)代表课程的数目，N(1≤N≤300)代表学生的数目；接下来有 P 行，描述了这些课程，从课程 1 到课程 P，每行描述了一门课程；每行首先是一个整数 $Count_i$，0≤$Count_i$≤N，表示见习第 i 门课程的学生人数，接着是一个空格，然后 $Count_i$ 个整数(用空格隔开)，代表见习这门课程的学生的序号；学生的序号从 1～N。

输出描述：

对输入文件中的每个测试数据，输出一行：如果能组成符合条件的委员会，则输出"YES"，否则输出"NO"。

样例输入：

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

样例输出：

```
YES
NO
```

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

分析：

很明显，本题要求解的是二部图的最大匹配。不难发现，只要匹配可以“盖住”每门课程，即匹配数与课程数量相等，委员会就可以组成。

样例输入中两个测试数据所描述的二分图如图 7.29(a)和 7.29(b)所示。在图 7.29(a)中，能找到这样匹配(粗线边组成一个匹配)，所以输出 YES，而在图 7.29(b)中，找不到这样的匹配，所以输出 NO。

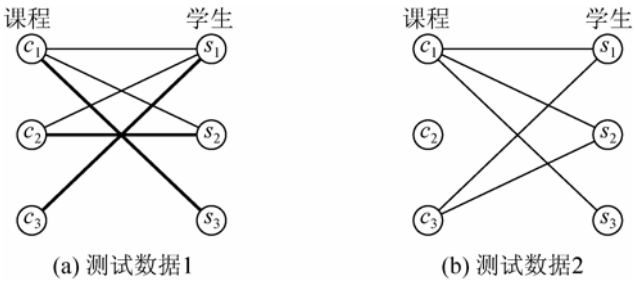


图 7.29　课程：两个测试数据

代码如下：

```
#define M 301
int P, N;   //课程数; 学生数
bool course[M][M], used[M];
int match[M];
bool DFS( int k )   //DFS 增广
{
    int i, temp;
    for( i=1; i<=N; i++ )
    {
        if( course[k][i] && !used[i] )
        {
            used[i]=1;
            temp=match[i];
            match[i]=k;
            if( temp==-1 || DFS( temp ) )
                return 1;
            match[i]=temp;
        }
    }
    return 0;
}
int MaxMatch()  //求二部图最大匹配
{
    int i, MatchNum=0;
    memset( match, -1, sizeof( match ) );
```

```
        for( i=1; i<=P; i++ )
        {
            memset( used, 0, sizeof( used ) );
            if( DFS( i ) ) MatchNum++; //累加匹配数
            if( MatchNum==P ) break;
        }
        return MatchNum;    //返回最大匹配数
    }
    int main( )
    {
        int i, j, num, t, n;
        scanf( "%d", &n );
        while( n-- )
        {
            //读入数据
            scanf( "%d%d", &P, &N );
            memset( course, 0, sizeof( course ) );
            for( i=1; i<=P; i++ )
            {
                scanf( "%d", &num );
                for( j=0; j<num; j++ )
                {
                    scanf( "%d", &t );
                    course[i][t]=1;
                }
            }
            //如果匹配数与课程数相等,则可以组成委员会
            if( MaxMatch()==P ) printf( "YES\n" );
            else printf( "NO\n" );
        }
        return 0;
    }
```

例 7.11　破坏有向图(Destroying the Graph)

题目来源：

Northeastern Europe 2003, Northern Subregion, ZOJ2429, POJ2125

题目描述：

Alice 和 Bob 正在玩一个游戏。首先，Alice 画一些包含 N 个顶点、M 条弧的有向图；然后，Bob 试图破坏它。在每一步，Bob 可以从图中移去一个顶点，并且移去所有进入该顶点的弧或者所有由该顶点发出的弧。在画图的时候，Alice 给每个顶点赋予了两个权值：W_i+和 W_i−。如果 Bob 移去所有进入第 i 个顶点的弧，则 Bob 需要支付 W_i+美元给 Alice；如果 Bob 移去所有由第 i 个顶点发出的弧，则 Bob 需要支付 W_i−美元给 Alice。

计算 Bob 移去图中所有的弧，需要支付的最少费用。

输入描述：

每个测试数据描述了 Alice 所画的一个有向图：第 1 行为两个整数 N 和 M，$1 \leqslant N \leqslant 100$，$1 \leqslant M \leqslant 5\,000$；第 2 行为 N 个整数，代表每个顶点的权值 W_i+；第 3 行也是 N 个整数，代表每个顶点的权值 W_i−；所有的权值为正整数，不超过 10^6；接下来有 M 行，每行为两个

整数，这 M 行描述了有向图中的 M 条弧。有向图中可能包含回路和重边。

输出描述：

每个测试数据的第 1 行为 W，表示 Bob 移去所有弧所需支付的最少费用；第 2 行为一个整数 K，表示 Bob 走的步数；接下来输出 K 行，描述 Bob 的每一步，每一行首先是顶点的序号，然后是一个字符'+'或'−'，字符'+'表示 Bob 移去该所有进入该顶点的弧，字符'−'表示 Bob 移去所有由该顶点发出的弧。

样例输入：	**样例输出：**
1	5
	3
3 6	1+
1 2 3	2-
4 2 1	2+
1 2	
1 1	
3 2	
1 2	
3 1	
2 3	

分析：

首先，每条弧<u, v>可能会被作为顶点 u 发出的弧被移去，也可能会被作为进入顶点 v 的弧被移去，所以移去所有的弧有很多方案。

本题中每个顶点有两个属性，删除入边的权值和删除出边的权值，可以将每个顶点拆为两个顶点，一个处理入边(该顶点称为入点)，另一个处理出边(该顶点称为出点)。如此一来，拆分后的顶点各分有一个属性值。整个图也转化为二部图。

具体来说，构造二部图的方法如下。

(1) 原图中每个顶点 V，拆分成两个点 V^-，V^+。

(2) 原图中每条边(U, V)，改成(U^-, V^+)。

例如，样例输入中测试数据所描绘的有向图如图 7.30(a)所示，对该图构造的二部图如图 7.30(b)所示。

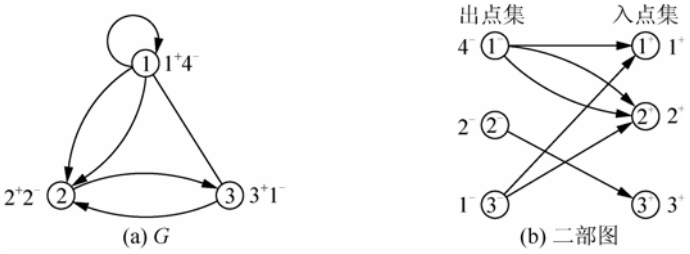


图 7.30　破坏有向图：二部图的构造

由原图构造成二部图，边数不变，顶点数加倍。拆分后的两个顶点分担了原来的顶点的责任：①删除原来顶点 V 的出边，相当于删除与 V^- 相连的边；②删除原来顶点 V 的入边，相当于删除与 V^+ 相连的边。

本题要求解的是移去图中所有的弧，需要支付的最少费用，即在图 7.30(b)所示的二部图中，选择一个最小权值的点集，“覆盖”住所有边，这是求二部图中的最小点权覆盖集。

最小点权覆盖集(Minimum Weight Vertex Covering Set)：设有无向图 $G(V, E)$，对于 $\forall u \in V$，都对应一个非负权值 w，称为顶点 u 的点权；点权之和最小的点覆盖集，称为最小点权覆盖集。

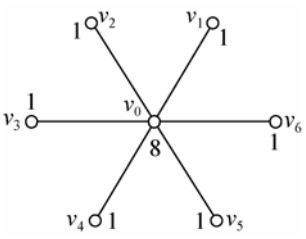


图 7.31　最小点权覆盖集

从定义可以看出，最小点权覆盖集肯定是极小点覆盖集，但不一定是最小点覆盖集。例如，在图 7.31 中，顶点旁的数字为顶点的点权，$\{v_0\}$和$\{v_1, v_2, v_3, v_4, v_5, v_6\}$都是极小点覆盖集，前者的点权之和为 8，后者的点权之和为 6。因此$\{v_0\}$是最小点覆盖集，但不是最小点权覆盖集，$\{v_1, v_2, v_3, v_4, v_5, v_6\}$是最小点权覆盖集。

最小点权覆盖集的求解可以借鉴二分图匹配的最大流解法。在加入额外的源 s 和汇 t 后，将匹配以一条 s–u–v–t 形式的流路径“串联”起来。匹配的限制是在顶点上，恰当地利用了流的容量限制。而点覆盖集的限制在边上，最小割是最大流的对偶问题，对偶往往是将问题的性质从顶点转边，从边转顶点。可以尝试着转化到最小割模型。

基于以上动机，建立一个源 s，向出点集中每个顶点 u 连边；建立一个汇点 t，从入点集中每个顶点 v 向汇点 t 连边。任意一条从 s 到 t 的路径，一定具有 s–u–v–t 的形式。割的性质是不存在一条从 s 到 t 的路径。故此路径上的三条边<s, u>、<u, v>、<v, t>中至少有一条边在割中。若人为地令<u, v>不可能在割中，即令其容量为正无穷大 $c(u, v)=\infty$。则条件简化为<s, u>、<v, t>中至少有一条边在最小割中，正好与点覆盖集限制条件的形式相符(每条边<u, v>，至少有一个顶点在顶点集 V'中，其中 V'就是一个点覆盖集)。最小点权覆盖集的目标是最小化点权之和，恰好也是最小割的优化目标。

根据以上分析，将原问题的图 G 转化为网络 $N=(V_N, E_N)$的最小割模型如下。

(1) 在图 G 的基础上增加源点 s 和汇点 t。

(2) 将二部图中每条边<u, v>$\in E$ 替换为容量为 $c(u, v)=\infty$的有向边<u, v>$\in E_N$。

(3) 增加源 s 到出点集每个顶点 u 的有向边<s, u>$\in E_N$，容量为权值 $Wu-$。

(4) 增加入点集每个顶点 v 到汇 t 的有向边<v, t>$\in E_N$，容量为权值 $Wv+$。

例如，对图 7.30(b)所示的二部图，构造的网络流模型如图 7.32(a)所示。

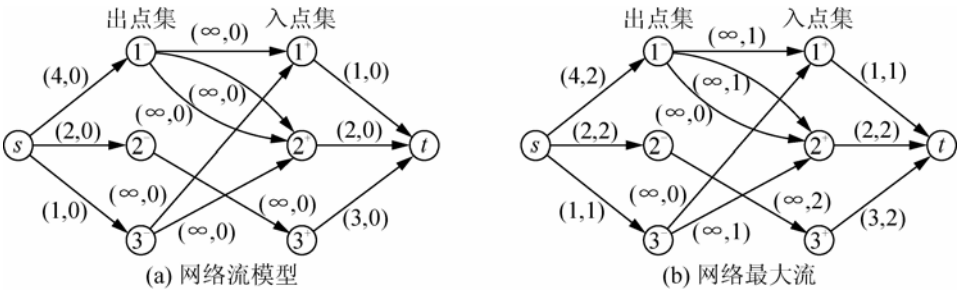


图 7.32　破坏有向图：构造网络流模型

网络流模型构造好以后，原问题的最小割等效于网络最大流，在图 7.32(b)中，求得网络

最大流为 5，因此移去所有弧的最小费用为 5。

网络最大流求解完毕后，所有割边对应的带权顶点就是操作的点，左边的边是'-'操作，右边是'+'操作。

代码如下：

```
using namespace std;
#define MAXN 2000
#define INF 10000000
struct Edge{ int next, c, f, other; }N;
vector <Edge> map[MAXN];
vector <int> level_map[MAXN];
vector <int> ans;
int q[MAXN*1], level[MAXN], pre[MAXN], hash[MAXN], d[MAXN];
int s, t;
int min( int a, int b ) { return a<b?a:b; }
void Add( int u, int v, int c )  //插入边
{
    N.next=v; N.c=c; N.other=map[v].size(); N.f=0;
    map[u].push_back( N );
    N.next=u; N.c=0; N.other=map[u].size()-1; N.f=0;
    map[v].push_back( N );
}
bool BFS()  //BFS 构建层次网络
{
    int head=0, tail=0, cur, i;
    //初始化
    for( i=s; i<=t; i++ )  level_map[i].clear();
    memset( level, -1, sizeof( level ) );
    q[tail++]=s;
    level[s]=0;
    while( head<tail )
    {
        cur=q[head++];
        for( i=0; i<map[cur].size(); i++ )
        {
            N=map[cur][i];
            if( N.c>N.f )
            {
                if( level[N.next]==-1 )
                {
                    q[tail++]=N.next;
                    level[N.next]=level[cur]+1;
                }
                if( level[N.next]==level[cur]+1 )
                {
                    level_map[cur].push_back( i );
                }
            }
        }
    }
```

```
    if (level[t]!=-1)  return 1;   //汇点在层次网络中
    else return 0;                 //汇点不在层次网络中
}
int Dinic() //Dinic 算法求最大流
{
    int i, j , ans=0, len;
    while( BFS() )
    {
        memset( hash, 0, sizeof( hash ) );  //初始化
        while( !hash[s] )  //在当前层次网络进行连续增广
        {
            d[s]=INF;
            pre[s]=-1;
            for( i=s; i!=t&&i!=-1; i=j )
            {
                len=level_map[i].size();
                while( len && hash[map[i][level_map[i][len-1]].next] )
                {
                    level_map[i].pop_back(); len--;
                }
                if( !len )
                {
                    hash[i]=1;
                    j=pre[i];
                    continue;
                }
                j=map[i][level_map[i][len-1]].next;
                pre[j]=i;
                d[j]=min( d[i], map[i][level_map[i][len-1]].c-map[i]
                          [level_map[i][len-1]].f );
            }
            if( i==t )
            {
                ans+=d[t];
                while( i!=s )  //调整流量
                {
                    j=pre[i];
                    len=level_map[j][level_map[j].size()-1];
                    map[j][len].f+=d[t];
                    if(map[j][len].f==map[j][len].c)level_map[j].pop_back();
                    map[i][map[j][len].other].f-=d[t];
                    i=j;
                }
            }
        }
    }
    return ans;  //返回最大流
}
void DFS( int u )  //DFS 遍历,建立关联关系
{
    int i, k;  hash[u]=1;
```

```
        for( i=0; i<map[u].size(); i++ )
        {
            k=map[u][i].next;
            if( !hash[k] && map[u][i].c-map[u][i].f>0 )
                DFS( k );   //递归调用
        }
    }
    int main()
    {
        int n, m, i, j, k, N, tmp, answer;
        while( scanf( "%d %d", &n, &m )!=EOF )
        {
            //读入数据,初始化
            N=n+n+1;
            s=0, t=N;
            for( i=s; i<=t; i++ )  map[i].clear();
            for( i=1; i<=n; i++ )
            {
                scanf( "%d", &tmp );
                Add( n+i, N, tmp );
            }
            for( i=1; i<=n; i++ )
            {
                scanf( "%d", &tmp );
                Add( 0, i, tmp );
            }
            for( k=0; k<m; k++ )
            {
                scanf( "%d %d", &i, &j );
                Add( i, n+j, INF );
            }
            answer=Dinic(); //求得最大流
            memset( hash, 0, sizeof( hash ) ); //初始化
            DFS( 0 );   //DFS 遍历,建立关联关系
            ans.clear();
            for( i=1; i<=n; i++ )
            {
                if( !hash[i] ) ans.push_back( i );     //左边为'-'操作
                if( hash[n+i] ) ans.push_back( n+i );  //右边为'+'操作
            }
            //输出
            printf( "%d\n%d\n", answer, ans.size() );
            for( i=0; i<ans.size(); i++ )
            {
                if( ans[i]<=n ) printf( "%d-\n", ans[i] );
                else printf( "%d +\n", ans[i]-n );
            }
        }
        return 0;
    }
```

练　习

7.1　Tom 叔叔继承的土地(Uncle Tom's Inherited Land)

题目来源：

South America 2002, Practice, ZOJ1516

题目描述：

老叔叔 Tom 从他的老老叔叔那里继承过来一块土地。最初，这块土地是长方形的。然而，很久以前，他的老老叔叔决定把这块土地分成方形土地的网格。他将其中的一些方块挖成池塘，因为他喜欢打猎，所以他想把野鸭吸引到他的池塘里来。由于池塘挖得太多了，导致在土地里可能形成了一些不连通的小岛。

Tom 想卖掉这块土地，但当地政府对不动产的出售有相应的政策。Tom 叔叔被告知，依照他的老老叔叔的要求，这块土地只能以两块方块土地组成的长方形土地块一起进行出售，而且，池塘是不许卖的。请帮忙计算 Tom 叔叔可以出售的最大长方形土地块数目(没卖出的方块土地将被规划成休闲公园)。

例如图 7.33(a)所示的 4×4 土地，最多可以出售 4 个长方形土地块，图 7.33(c)给出了两个可行解。

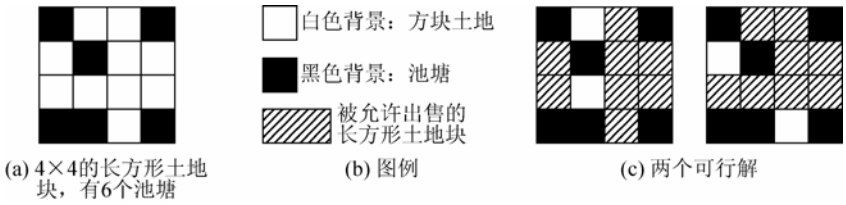


图 7.33　Tom 叔叔继承的土地

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 行包含两个整数：N 和 M，分别代表长方形土地行和列的数目 $1 \leqslant N$、$M \leqslant 100$；第 2 行为整数 K，代表已经被挖成池塘的方块土地数目，$(N \times M) - K \leqslant 50$；接下来有 K 行，每行为两个整数 X 和 Y，描述了 K 块被挖成池塘的方块土地的位置，$1 \leqslant X \leqslant N$，$1 \leqslant Y \leqslant M$。输入文件的最后一行为 $N = M = 0$，代表输入结束。

输出描述：

对输入文件中的每个测试数据，程序必须输出一行，为一个整数，代表 Tom 可以卖出的长方形土地块数目。

样例输入：

```
4 4
6
1 1
1 4
2 2
4 1
4 2
4 4
0 0
```

样例输出：

```
4
```

7.2 女生和男生(Girls and Boys)

题目来源：

Southeastern Enrope 2000, ZOJ1137, POJ1466

题目描述：

大二的时候，有人开始研究同学之间的浪漫关系。浪漫关系被定义为一个男孩和一个女孩之间的关系。为了研究的需要，有必要找出满足条件的最大集合：集合内任何两个学生都没有发生浪漫关系。程序需要输出该集合中学生的人数。

输入描述：

输入文件中包含若个测试数据。每个测试数据代表一组研究对象，格式如下：第 1 行是一个整数 *n*，代表学生人数；接下来有 *n* 行，每行描述了一个学生，遵循如下格式。

学号: (关系的数目) 学号 1、学号 2、学号 3 ……

或者是如下格式

学号: (0)

学生的学号是 0 到 *n*−1 的整数。

输出描述：

对输入文件中的每个测试数据，输出相互之间没有发生浪漫关系的最大学生集合中的人数。

样例输入：

```
7
0: (3) 4 5 6
1: (2) 4 6
2: (0)
3: (0)
4: (2) 0 1
5: (1) 0
6: (2) 0 1
3
0: (2) 1 2
1: (1) 0
2: (1) 0
```

样例输出：

```
5
2
```

注解及提示：

如果将每个人用一个顶点表示，有关系的人之间连一条线，则构成一个二部图(男生之间不会有关系，女生之间也不会有关系)。例如，题目中两个测试数据所对应的二部图如图 7.34 所示。

(a) 测试数据1　　(b) 测试数据2

图 7.34　女生和男生

本题需要求最大点独立集，将问题转换成求二部图的最大匹配。

7.3　姓名中的秘密(What's In a Name)

题目描述：

East Central North America 2001, ZOJ1059, POJ1043

题目描述：

FBI 对一个犯罪团伙的藏匿点进行监视，这个藏匿点是这个团伙的通信中心，团伙成员通过邮件进行联系。利用尖端的解密软件和缜密的窃听技术，FBI 能够解密从藏匿点发出的任何邮件。然而，在办理逮捕许可证前，他们必须将邮件中的用户名跟罪犯的真实姓名对应起来。但是，犯罪团伙很狡猾，他们采用随机的字符串作为他们的用户名。FBI 知道每个罪犯只使用一个 ID。另外，FBI 通过监视镜头记录下藏匿点的人员进出情况，由此得到一日志(以时间序)。在很多情况下，这足以将每个罪犯的姓名跟用户名对应起来。

输入描述：

输入文件包含多个测试数据。输入文件中的第 1 行为一个整数 *N*，代表测试数据的数目。接下来是一个空行，然后是 *N* 个测试数据，测试数据之间有一个空行隔开。

每个测试数据的第 1 行为一个正整数 *n*，代表罪犯的人数，*n* 的最大值为 20；接下来一行为 *n* 个用户名，用空格隔开；接下来是以时间顺序排列的犯罪团伙进出日志，日志中每条记录的格式为：类型参数，其中类型为 *E*、*L* 或 *M*，*E* 表示进入房间，后面跟的参数(字符串)表示进来的人的姓名；*L* 表示离开房间，后面跟的参数(字符串)表示离开的人的姓名；*M* 表示 FBI 截取到一封邮件，后面跟的字符串表示一个用户名，即当前用这个用户名的人在藏匿点里面；最后一行只有一个字母 *Q*，代表日志的结束。注意，不是所有的用户名都在日志中，但每个人的姓名都会在日志中至少出现一次。在日志记录前，藏匿点是空的。所有的姓名和用户名只包含小写字母字符，长度不超过 20。注意，包含用户名的一行可能会超过 80 个字符。

输出描述：

对输入文件中的每个测试数据，输出占 *n* 行，为 *n* 个罪犯姓名和用户名的对应关系列表。列表按罪犯姓名的字典序进行排序；每行的格式为“姓名：用户名”，如果根据日志无法确定某个罪犯的用户名，则用字符串"???"代替他的用户名。

每个测试数据的输出之间有一个空行。

样例输入：

```
1

7
bigman mangler sinbad fatman bigcheese frenchie capodicapo
E mugsy
E knuckles
M bigman
M mangler
L mugsy
E clyde
```

样例输出：

```
bonnie:fatman
booth:???
clyde:frenchie
knuckles:bigman
moriarty:???
mugsy:mangler
ugati:sinbad
```

```
E bonnie
M bigman
M fatman
M frenchie
L clyde
M fatman
E ugati
M sinbad
E moriarty
E booth
Q
```

7.4　空袭(Air Raid)

题目来源：

Asia 2002, Dhaka(Bengat), ZOJ1525, POJ1422

题目描述：

考虑一个小镇，所有的街道都是单向的，这些街道都是从一个十字路口通往另一个十字路口。已知从任何一个十字路口出发，沿着这些街道行走，都是不能再回到同一个十字路口的，也就是说，不存在回路。

在这些假定下，试编写一个程序计算袭击这个小镇需要派出伞兵的最少数目。这些伞兵要走遍小镇的所有十字路口，每个十字路口只由一个伞兵走到。每个伞兵在一个十字路口着陆，沿着街道可以走到其他十字路口。每个伞兵选择的起始十字路口没有限制。

输入描述：

输入文件的第 1 行为一个整数 T，代表测试数据的数目。每个测试数据描述了一个小镇，格式为：第 1 行为一个正整数 n，表示小镇中十字路口的数目，$0<n\leqslant 120$，十字路口的序号用 1～n 标明；第 2 行为一个正整数 m，表示街道的数目；接下来有 m 行，每行描述了小镇中的一条街道，每行为两个整数，分别表示这条街道所连接的两个十字路口的序号。

输出描述：

对输入文件中的每个测试数据，输出占一行，为一个整数，表示需要派出伞兵的最小数目。

样例输入：

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

样例输出：

```
2
1
```

7.5　小行星(Asteroids)

题目来源：

USACO 2003 Norember Gold, POJ3041

题目描述：

贝茜想驾驶她的太空飞船航行于一个 $N \times N$ 的网格，$1 \leqslant K \leqslant 10\,000$，网格中分布了 K 个危险的小行星，这些小行星位于网格中的方格里。

幸运的是，贝茜有一种强大的武器，只要一枚子弹，就可以使某一行或某一列上的所有小行星气化。这种武器太昂贵了，所以她必须很节俭地使用这种武器。给定网格中所有小行星的位置，求贝茜至少需要发射多少枚子弹，从而消除所有的小行星。

输入描述：

每个测试数据的第 1 行为两个整数，N 和 K；第 2 行～第 K+1 行，每行为两个整数，R 和 C，$1 \leqslant R$、$C \leqslant N$，代表每个小行星的行和列位置。

输出描述：

输出一行，为贝茜需要发射子弹的最少数目。

样例输入：

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

样例输出：

```
2
```