动态规划的特点及其应用

安徽 张辰

目 录

（点击进入）

【关键词】动态规划 阶段

【摘要】

动态规划是信息学竞赛中的常见算法，本文的主要内容就是分析它的特点。

文章的第一部分首先探究了动态规划的本质，因为动态规划的特点是由它的本质所决定的。第二部分从动态规划的设计和实现这两个角度分析了动态规划的多样性、模式性、技巧性这三个特点。第三部分将动态规划和递推、搜索、网络流这三个相关算法作了比较，从中探寻动态规划的一些更深层次的特点。

文章在分析动态规划的特点的同时，还根据这些特点分析了我们在解题中应该怎样利用这些特点，怎样运用动态规划。这对我们的解题实践有一定的指导意义。

【正文】

动态规划是编程解题的一种重要的手段，在如今的信息学竞赛中被应用得越来越普遍。最近几年的信息学竞赛，不分大小，几乎每次都要考察到这方面的内容。因此如何更深入地了解动态规划，从而更为有效地运用这个解题的有力武器，是一个值得深入研究的问题。

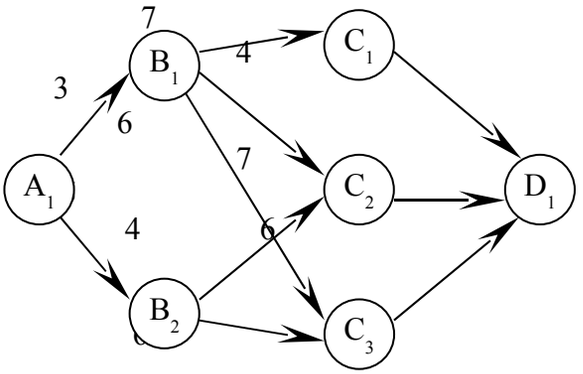
要掌握动态规划的应用技巧，就要了解它的各方面的特点。首要的，是要深入洞悉动态规划的本质。

§1 动态规划的本质

动态规划是在本世纪 50 年代初，为了解决一类多阶段决策问题而诞生的。那么，什么样的问题被称作多阶段决策问题呢？

§1.1 多阶段决策问题

说到多阶段决策问题，人们很容易举出下面这个例子。



［例1］多段图中的最短路径问题：在下图中找出从 A_1 到 D_1 的最短路径。

仔细观察这个图不难发现，它有一个特点。我们将图中的点分为四类（图中的A、B、C、D），那么图中所有的边都处于相邻的两类点之间，并且都从前一类点指向后一类点。这样，图中的边就被分成了三类（A→B、B→C、C→D）。我们需要从每一类中选出一条边来，组成从 A1 到 D1 的一条路径，并且这条路径是所有这样的路径中的最短者。

从上面的这个例子中，我们可以大概地了解到什么是多阶段决策问题。更精确的定义如下：

多阶段决策过程，是指这样的一类特殊的活动过程，问题可以按时间顺序分解成若干相互联系的阶段，在每一个阶段都要做出决策，全部过程的决策是一个决策序列[1]。要使整个活动的总体效果达到最优的问题，称为**多阶段决策问题**。

从上述的定义中，我们可以明显地看出，这类问题有两个要素。一个是阶段，一个是决策。

§1.2 阶段与状态

阶段：将所给问题的过程，按时间或空间特征分解成若干相互联系的阶段，以便按次序去求每阶段的解。常用字母 k 表示阶段变量。[1]

阶段是问题的属性。多阶段决策问题中通常存在着若干个阶段，如上面的例子，

就有 A、B、C、D 这四个阶段。在一般情况下，阶段是和时间有关的；但是在很多问题（我的感觉，特别是信息学问题）中，阶段和时间是无关的。从阶段的定义中，可以看出阶段的两个特点，一是“相互联系”，二是“次序”。

阶段之间是怎样相互联系的？就是通过状态和状态转移。

状态：各阶段开始时的客观条件叫做状态。描述各阶段状态的变量称为状态变量，常用 s_k 表示第 k 阶段的状态变量，状态变量 s_k 的取值集合称为状态集合，用 S_k 表示。[1]

状态是阶段的属性。每个阶段通常包含若干个状态，用以描述问题发展到这个阶段时所处在的一种客观情况。在上面的例子中，行人从出发点 A_1 走过两个阶段之后，可能出现的情况有三种，即处于 C_1、C_2 或 C_3 点。那么第三个阶段就有三个状态 $S_3=\{C_1,C_2,C_3\}$。

每个阶段的状态都是由以前阶段的状态以某种方式“变化”而来，这种“变化”称为状态转移（暂不定义）。上例中 C_3 点可以从 B_1 点过来，也可以从 B_2 点过来，从阶段 2 的 B_1 或 B_2 状态走到阶段 3 的 C_3 状态就是状态转移。状态转移是导出状态的途径，也是联系各阶段的途径。

说到这里，可以提出应用动态规划的一个重要条件。那就是将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的发展，而只能通过当前的这个状态。换句话说，每个状态都是“过去历史的一个完整总结[1]”。这就是**无后效性**。对这个性质，下文还将会有解释。

§1.3 决策和策略

上面的阶段与状态只是多阶段决策问题的一个方面的要素，下面是另一个方面的要素——决策。

决策：当各段的状态取定以后，就可以做出不同的决定，从而确定下一阶段的状态，这种决定称为决策。表示决策的变量，称为决策变量，常用 $u_k(s_k)$ 表示第 k 阶段当状态为 s_k 时的决策变量。在实际问题中，决策变量的取值往往限制在一定范围内，我

们称此范围为允许决策集合。常用 $D_k(s_k)$表示第 k 阶段从状态 s_k 出发的允许决策集合。显然有 $u_k(s_k) \in D_k(s_k)$。[1]

决策是问题的解的属性。决策的目的就是“确定下一阶段的状态”，还是回到上例，从阶段 2 的 B_1 状态出发有三条路，也就是三个决策，分别导向阶段 3 的 C_1、C_2、C_3 三个状态，即 $D_2(B_1)=\{C_1,C_2,C_3\}$。

有了决策，我们可以定义**状态转移**：动态规划中本阶段的状态往往是上一阶段和上一阶段的决策结果，由第 k 段的状态 s_k 和本阶段的决策 u_k 确定第 k+1 段的状态 s_{k+1} 的过程叫状态转移。状态转移规律的形式化表示 $s_{k+1}=T_k(s_k,u_k)$称为**状态转移方程**。

这样看来，似乎决策和状态转移有着某种联系。我的理解，状态转移是决策的目的，决策是状态转移的途径。

各段决策确定后，整个问题的决策序列就构成一个**策略**，用 $p_{1,n}=\{u_1(s_1),u_2(s_2),\ldots,u_n(s_n)\}$表示。对每个实际问题，可供选择的策略有一定范围，称为允许策略集合，记作 $P_{1,n}$，使整个问题达到最有效果的策略就是最优策略。[1]

说到这里，又可以提出运用动态规划的一个前提。即这个过程的最优策略应具有这样的性质：无论初始状态及初始决策如何，对于先前决策所形成的状态而言，其以后的所有决策应构成最优策略[1]。这就是**最优化原理**。简言之，就是“最优策略的子策略也是最优策略”。

§1.4 最优化原理与无后效性

这里，我把最优化原理定位在“运用动态规划的前提”。这是因为，是否符合最优化原理是一个问题的本质特征。对于不满足最优化原理的一个多阶段决策问题，整体上的最优策略 $p_{1,n}$ 同任何一个阶段 k 上的决策 u_k 或任何一组阶段 $k_1\ldots k_2$ 上的子策略 $p_{k1,k2}$ 都不存在任何关系。如果要对这样的问题动态规划的话，我们从一开始所作的划分阶段等努力都将是徒劳的。

而我把无后效性定位在“应用动态规划的条件”，是因为动态规划是按次序去求每阶段的解，如果一个问题有后效性，那么这样的次序便是不合理的。但是，我们可以通过重新划分阶段，重新选定状态，或者增加状态变量的个数等手段，来是问题满足无后效性这个条件。说到底，还是要确定一个“序”。

在信息学的多阶段决策问题中，绝大部分都是能够满足最优化原理的，但它们往往会在后效性这一点上来设置障碍。所以在解题过程中，我们会特别关心“序”。对于有序的问题，就会考虑到动态规划；对于无序的问题，也会想方设法来使其有序。

§1.5 最优指标函数和规划方程

最优指标函数：用于衡量所选定策略优劣的数量指标称为指标函数，最优指标函数记为 $f_k(s_k)$，它表示从第 k 段状态 s_k 采用最优策略 $p^*_{k,n}$ 到过程终止时的最佳效益值[1]。

最优指标函数其实就是我们真正关心的问题的解。在上面的例子中，$f_2(B_1)$就表示从 B_1 点到终点 D_1 点的最短路径长度。我们求解的最终目标就是 $f_1(A_1)$。

最优指标函数的求法一般是一个从目标状态出发的递推公式，称为规划方程：

$$f_k(s_k)=\operatorname*{opt}_{u_k\in D_k(s_k)} g\left(f_{k+1}\left(T_k(s_k,u_k)\right),u_k\right)$$

其中 s_k 是第 k 段的某个状态，u_k 是从 s_k 出发的允许决策集合 $D_k(s_k)$中的一个决策，$T_k(s_k,u_k)$是由 s_k 和 u_k 所导出的第 k+1 段的某个状态 s_{k+1}，$g(x,u_k)$是定义在数值 x 和决策 u_k 上的一个函数，而函数 opt 表示最优化，根据具体问题分别表为 max 或 min。

$f_n(s_n)=$某个初始值，称为边界条件。

上例中的规划方程就是：

$$f_k(s_k)=\min_{\text{从}s_k\text{出发的某条边}u_k}\left(f_{k+1}(u_k\text{指向的点}s_{k+1})+\text{边}u_k\text{的长度}\right)$$

边界条件为 $f_4(D_1)=0$

这里是一种从目标状态往回推的逆序求法，适用于目标状态确定的问题。在我们的信息学问题中，也有很多有着确定的初始状态。当然，对于初始状态确定的问题，我们也可以采用从初始状态出发往前推的顺序求法。事实上，这种方法对我们来说要更为直观、更易设计一些，从而更多地出现在我们的解题过程中。

我们本节所讨论的这些理论虽然不是本文的主旨，但是却对下面要说的动态规划的特点起着基础性的作用。

§2 动态规划的设计与实现

上面我们讨论了动态规划的一些理论，本节我们将通过几个例子中，动态规划的设计与实现，来了解动态规划的一些特点。

§2.1 动态规划的多样性

[例2] 花店橱窗布置问题（IOI99）试题见附录

本题虽然是本届 IOI 中较为简单的一题，但其中大有文章可作。说它简单，是因为它有序，因此我们一眼便可看出这题应该用动态规划来解决。但是，如何动态规划呢？如何划分阶段，又如何选择状态呢？

<方法 1>以花束的数目来划分阶段。在这里，阶段变量 k 表示的就是要布置的花束数目（前 k 束花），状态变量 s_k 表示第 k 束花所在的花瓶。而对于每一个状态 s_k，决策就是第 k-1 束花应该放在哪个花瓶，用 u_k 表示。最优指标函数 $f_k(s_k)$ 表示前 k 束花，其中第 k 束插在第 s_k 个花瓶中，所能取得的最大美学值。

状态转移方程为 $s_{k-1}=u_k$

规划方程为 $f_k(s_k)=\max_{k\le u_k<s_k}\left(f_{k-1}(u_k)+A(k,s_k)\right)$

（其中 A(i,j)是花束 i 插在花瓶 j 中的美学值）

边界条件 $f_0(s_0)=0 \quad (0 \le s_0 \le V)$ （V 是花瓶总数，事实上这是一个虚拟的边界）

<方法 2>以花瓶的数目来划分阶段。在这里阶段变量 k 表示的是要占用的花瓶数目（前 k 个花瓶），状态变量 s_k 表示前 k 个花瓶中放了多少花。而对于任意一个状态 s_k，决策就是第 s_k 束花是否放在第 k 个花瓶中，用变量 u_k=1 或 0 来表示。最优指标函数 $f_k(s_k)$表示前 k 个花瓶中插了 s_k 束花，所能取得的最大美学值。

状态转移方程为 $s_{k-1}=s_k-u_k$

规划方程为 $f_k(s_k)=\max_{u=0,1}\left(f_{k-1}(s_k-u_k)+u_k \cdot A(s_k,k)\right)$

边界条件为 $f_k(0)=0 \quad (0 \le k \le V)$

两种划分阶段的方法，引出了两种状态表示法，两种规划方式，但是却都成功地解决了问题。只不过因为决策的选择有多有少，所以算法的时间复杂度也就不同。[2]

这个例子具有很大的普遍性。有很多的多阶段决策问题都有着不止一种的阶段划分方法，因而往往就有不止一种的规划方法。有时各种方法所产生的效果是差不多的但更多的时候，就像我们的例子一样，两种方法会在某个方面有些区别。

所以，在用动态规划解题的时候，可以多想一想是否有其它的解法。对于不同的解法，要注意比较，好的算法好在哪里，差一点的算法差在哪里。从各种不同算法的比较中，我们可以更深刻地领会动态规划的构思技巧。

§2.2 动态规划的模式性

这个可能做过动态规划的人都有体会，从我们上面对动态规划的分析也可以看出来。动态规划的设计都有着一定的模式，一般要经历以下几个步骤。

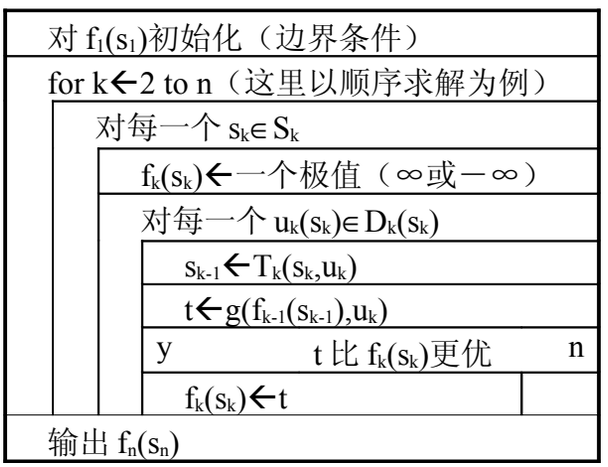
划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。注意这若干个阶段一定要是有序的或者是可排序的，否则问题就无法求解。

选择状态：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

确定决策并写出状态转移方程：之所以把这两步放在一起，是因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以，如果我们确定了决策，状态转移方程也就写出来了。但事实上，我们常常是反过来做，根据相邻两段的各状态之间的关系来确定决策。

写出规划方程（包括边界条件）：在第一部分中，我们已经给出了规划方程的通用形式化表达式。一般说来，只要阶段、状态、决策和状态转移确定了，这一步还是比较简单的。

动态规划的主要难点在于理论上的设计，一旦设计完成，实现部分就会非常简单。大体上的框架如下：



这个 N-S 图虽然不能代表全部，但足可以概括大多数。少数的一些特殊的动态规划，其实现的原理也是类似，可以类比出来。我们到现在对动态规划的分析，主要是在理论上、设计上，原因也就在此。

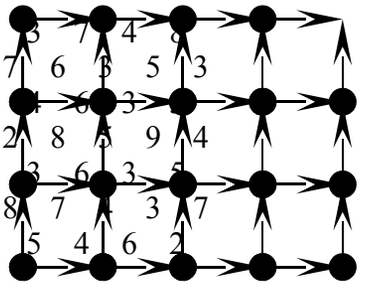
掌握了动态规划的模式性，我们在用动态规划解题时就可以把主要的精力放在理论上的设计。一旦设计成熟，问题也就基本上解决了。而且在设计算法时也可以按部就班地来。

但是“物极必反”，太过拘泥于模式就会限制我们的思维，扼杀优良算法思想的产生。我们在解题时，不妨发挥一下创造性，去突破动态规划的实现模式，这样往往会收到意想不到的效果。[3]

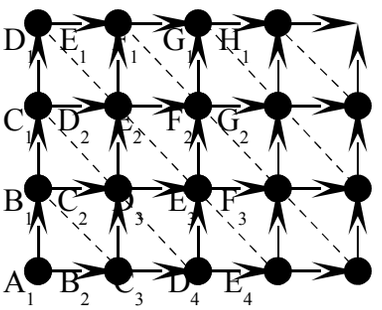
§2.3 动态规划的技巧性

上面我们所说的动态规划的模式性，主要指的是实现方面。而在设计方面，虽然它较为严格的步骤性，但是它的设计思想却是没有一定的规律可循的。这就需要我们不断地在实践当中去掌握动态规划的技巧，下面仅就一个例子谈一点我自己的体会。

[例3] 街道问题：在下图中找出从左下角到右上角的最短路径，每步只能向右方或上方走。



这是一道简单而又典型的动态规划题，许多介绍动态规划的书与文章中都拿它来做例子。通常，书上的解答是这样的：

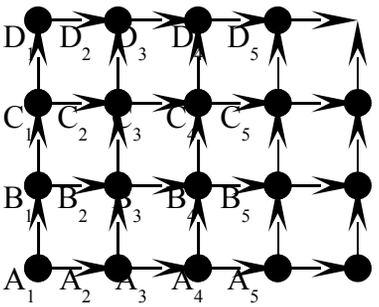


按照图中的虚线来划分阶段，即阶段变量 k 表示走过的步数，而状态变量 s_k 表示当前处于这一阶段上的哪一点（各点所对应的阶段和状态已经用 k_s 在地图上标明）。这时的模型实际上已经转化成了一个特殊的多段图。用决策变量 $u_k=0$ 表示向右走，$u_k=1$ 表示向上走，则状态转移方程如下：

$$s_{k-1}:\begin{cases} s_k-1+u_k & (k\le row) \\ s_k+u_k & (k>row) \end{cases}$$

（这里的 row 是地图竖直方向的行数）

我们看到，这个状态转移方程需要根据 k 的取值分两种情况讨论，显得非常麻烦。相应的，把它代入规划方程而付诸实现时，算法也很繁。因而我们在实现时，一般是



不会这么做的，而代之以下面方法：

将地图中的点规则地编号如上，得到的规划方程如下：

$$f_{i,j} := \min\begin{cases} f_{i-1,j} + Distance_{(i-1,j),(i,j)} \\ f_{i,j-1} + Distance_{(i,j-1),(i,j)} \end{cases}$$

（这里 Distance 表示相邻两点间的边长）

这样做确实要比上面的方法简单多了，但是它已经破坏了动态规划的本来面目，而不存在明确的阶段特征了。如果说这种方法是以地图中的行（A、B、C、D）来划分阶段的话，那么它的“状态转移”就不全是在两个阶段之间进行的了。

也许这没什么大不了的，因为实践比理论更有说服力。但是，如果我们把题目扩展一下：在地图中找出从左下角到右上角的两条路径，两条路径中的任何一条边都不能重叠，并且要求两条路径的总长度最短。这时，再用这种“简单”的方法就不太好办了。

如果非得套用这种方法的话，则最优指标函数就需要有四维的下标，并且难以处理两条路径“不能重叠”的问题。

而我们回到原先“标准”的动态规划法，就会发现这个问题很好解决，只需要加一维状态变量就成了。即用 $s_k=(a_k,b_k)$分别表示两条路径走到阶段 k 时所处的位置，相应的，决策变量也增加一维，用 $u_k=(x_k,y_k)$分别表示两条路径的行走方向。状态转移时将两条路径分别考虑：

$$\begin{cases} (k=row) & \begin{cases} a_{k-1}:=a_k-1-x_k \\ b_{k-1}:=b_k-1-y_k \end{cases} \\ (k\neq row) & \begin{cases} a_{k-1}:=a_k+x_k \\ b_{k-1}:=b_k+y_k \end{cases} \end{cases}$$

在写规划方程时，只要对两条路径走到同一个点的情况稍微处理一下，减少可选的决策个数：

$$f_k(s_k):=\begin{cases} \min\limits_{u_k:=(0,1),(1,0)} \left\{ f_{k-1}\left[T_k(s_k,u_k)\right]+\text{两条边长} \right\} & a_k:=b_k \\ \min\limits_{u_k:=(0,0),(0,1),(1,0),(1,1)} \left\{ f_{k-1}\left[T_k(s_k,u_k)\right]+\text{两条边长} \right\} & a_k\neq b_k \end{cases}$$

从这个例子中可以总结出设计动态规划算法的一个技巧：状态转移一般是在相邻的两个阶段之间（有时也可以在不相邻的两个阶段间），但是尽量不要在同一个阶段内进行。

动态规划是一种很灵活的解题方法，在动态规划算法的设计中，类似的技巧还有很多。要掌握动态规划的技巧，有两条途径：一是要深刻理解动态规划的本质，这也是我们为什么一开始就探讨它的本质的原因；二是要多实践，不但要多解题，还要学会从解题中探寻规律，总结技巧。

§3 动态规划与一些算法的比较

动态规划作为诸多解题方法中的一种，必然和其他一些算法有着诸多联系。从这些联系中，我们也可以看出动态规划的一些特点。

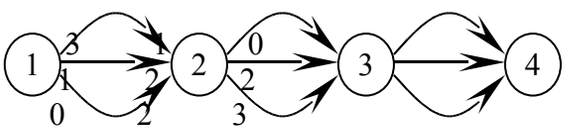
§3.1 动态规划与递推

——动态规划是最优化算法

由于动态规划的“名气”如此之大，以至于很多人甚至一些资料书上都往往把一种与动态规划十分相似的算法，当作是动态规划。这种算法就是递推。实际上，这两种算法还是很容易区分的。

按解题的目标来分，信息学试题主要分四类：判定性问题、构造性问题、计数问题和最优化问题。我们在竞赛中碰到的大多是最优化问题，而动态规划正是解决最优化问题的有力武器，因此动态规划在竞赛中的地位日益提高。而递推法在处理判定性问题和计数问题方面也是一把利器。下面分别就两个例子，谈一下递推法和动态规划在这两个方面的联系。

[例4] mod 4 最优路径问题：在下图中找出从第 1 点到第 4 点的一条路径，要求路径长度 mod 4 的余数最小。



这个图是一个多段图，而且是一个特殊的多段图。虽然这个图的形式比一般的多段图要简单，但是这个最优路径问题却不能用动态规划来做。因为一条从第 1 点到第 4 点的最优路径，在它走到第 2 点、第 3 点时，路径长度 mod 4 的余数不一定是最小，也就是说最优策略的子策略不一定最优——这个问题不满足最优化原理。

但是我们可以把它转换成判定性问题，用递推法来解决。判断从第 1 点到第 k 点的长度 mod 4 为 s_k 的路径是否存在，用 $f_k(s_k)$来表示，则递推公式如下：

$$f_1(s_1):\begin{cases} true & (s_1:0) \\ false & (s_1:1,2,3) \end{cases}$$ （边界条件）

$$f_k(s_k)=\begin{bmatrix} f_{k-1}((s_k-len_{k,1}) \bmod 4) \\ f_{k-1}((s_k-len_{k,2}) \bmod 4) \\ f_{k-1}((s_k-len_{k,3}) \bmod 4) \end{bmatrix}$$

（这里 $len_{k,i}$ 表示从第 k-1 点到第 k 点之间的第 i 条边的长度，方括号表示“或(or)”运算）

最后的结果就是可以使 $f_4(s_4)$ 值为真的最小的 s_4 值。

这个递推法的递推公式和动态规划的规划方程非常相似，我们在这里借用了动态规划的符号也就是为了更清楚地显示这一点。其实它们的思想也是非常相像的，可以说是递推法借用了动态规划的思想解决了动态规划不能解决的问题。

有的多阶段决策问题（像这一题的阶段特征就很明显），由于不能满足最优化原理等使用动态规划的先决条件，而无法应用动态规划。在这时可以将最优指标函数的值当作“状态”放到下标中去，从而变最优化问题为判定性问题，再借用动态规划的思想，用递推法来解决问题。

[例5] 钉子与小球（NOI99）试题见附录

这个题目一看就不觉让人想起一道经典的动态规划题。下面先让我们回顾一下这个问题。

数字三角形（IOI94）在下图中求从顶至低某处的一条路径，使该路径所经过的数字的总和最大，每一步只能向左下或右下走。

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

在这个问题中，我们按走过的行数来划分阶段，以走到每一行时所在的位置来作为状态，决策就是向左下走（用 0 表示）或向右下走（用 1 表示）。

状态转移方程：$s_{k-1}=s_k-u_k$

规划方程：$f_k(s_k)=\max_{u_k=0,1} f_{k-1}(s_k-u_k)+Data_k(s_k)$

边界条件：$f_k(0)=0 \quad f_k(k+1)=0 \quad (0\le k\le 总行数)$

这是一个比较简单的最优化问题，我们还可以把这个问题改成一个更加简单的整数统计问题：求顶点到每一点的路径总数。把这个总数用 $f_k(s_k)$ 表示，那么递推公式就是：

$$f_k(s_k)=\sum_{u_k=0}^{1} f_{k-1}(s_k-u_k)$$

在这里，虽然求和公式只有两项，但我们仍然用 Σ 的形式表示，就是为了突出这个递推公式和上面的规划方程的相似之处。这两个公式的边界条件都是一模一样的。

再回到我们上面的“钉子与小球”问题，这是一个概率统计问题。我们继续沿用上面的思想，用 $f_k(s_k)$ 表示小球落到第 k 行第 s_k 个钉子上的概率，则递推公式如下：

$$f_k(s_k)=\sum_{u_k=0}^{1}\frac{f_{k-1}(s_k-u_k)\cdot Exist_{k-1}(s_k-u_k)}{2}+f_{k-2}(s_k-1)\cdot(1-Exist_{k-2}(s_{k-1}))$$

（这里函数 $Exist_k(s_k)$ 表示第 k 行第 s_k 个钉子是否存在，存在则取 1，不存在则取 0）

边界条件 $f_1(1)=1$
$f_k(0)=0 \quad f_k(k+1)=0 \quad (1\le k\le 总行数)$

可以看出这个公式较之上面的两个式子虽然略有变化，但是其基本思想还是类似的。在解这个问题的过程中，我们再次运用了动态规划的思想。

一般说来，很多最优化问题都有着对应的计数问题；反过来，很多计数问题也有着对应的最优化问题。因此，我们在遇到这两类问题时，不妨多联系、多发展，举一反三，从比较中更深入地理解动态规划的思想。

其实递推和动态规划这两种方法的思想本来就很相似，也不必说是谁借用了谁的思想。关键在于我们要掌握这种思想，这样我们无论在用动态规划法解最优化问题，或是在用递推法解判定型、计数问题时，都能得心应手、游刃有余了。

§3.2 动态规划与搜索

——动态规划是高效率、高消费算法

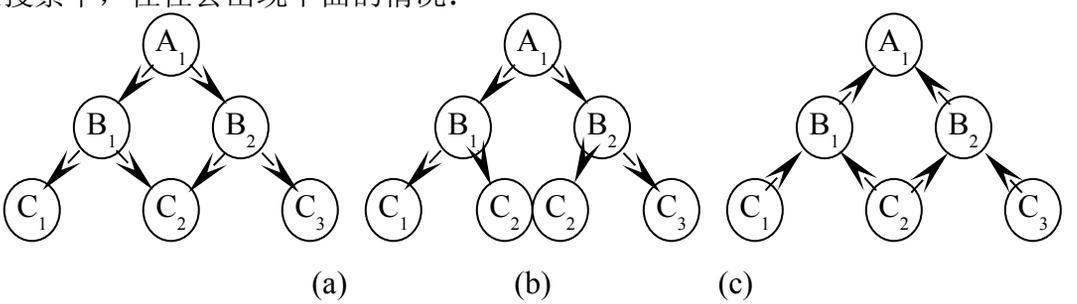
同样是解决最优化问题，有的题目我们采用动态规划，而有的题目我们则需要用搜索。这其中有没有什么规则呢？

我们知道，撇开时空效率的因素不谈，在解决最优化问题的算法中，搜索可以说是“万能”的。所以动态规划可以解决的问题，搜索也一定可以解决。

把一个动态规划算法改写成搜索是非常方便的，状态转移方程、规划方程以及边界条件都可以直接“移植”，所不同的只是求解顺序。动态规划是自底向上的递推求解，而搜索则是自顶向下的递归求解（这里指深度搜索，宽度搜索类似）。

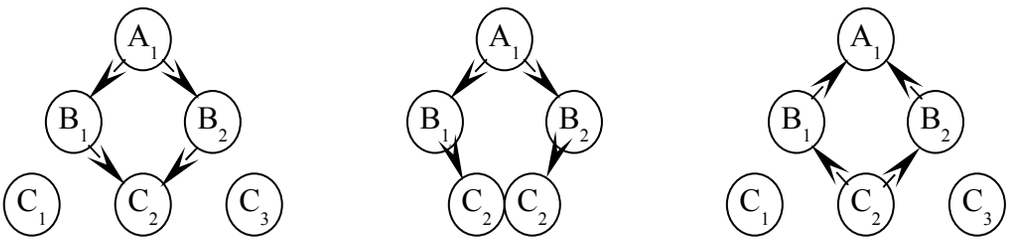
反过来，我们也可以把搜索算法改写成动态规划。状态空间搜索实际上是对隐式图中的点进行枚举，这种枚举是自顶向下的。如果把枚举的顺序反过来，变成自底向上，那么就成了动态规划。（当然这里有个条件，即隐式图中的点是可排序的，详见下一节。）

正因为动态规划和搜索有着求解顺序上的不同，这也造成了它们时间效率上的差别。在搜索中，往往会出现下面的情况：



对于上图(a)这样几个状态构成的一个隐式图，用搜索算法就会出现重复，如上图(b)所示，状态 C_2 被搜索了两次。在深度搜索中，这样的重复会引起以 C_2 为根整个的整个子搜索树的重复搜索；在宽度搜索中，虽然这样的重复可以立即被排除，但是其时间代价也是不小的。而动态规划就没有这个问题，如上图(c)所示。

一般说来，动态规划算法在时间效率上的优势是搜索无法比拟的。（当然对于某些题目，根本不会出现状态的重复，这样搜索和动态规划的速度就没有差别了。）而从理论上讲，任何拓扑有序（现实中这个条件常常可以满足）的隐式图中的搜索算法都可以改写成动态规划。但事实上，在很多情况下我们仍然不得不采用搜索算法。那么，动态规划算法在实现上还有什么障碍吗？



考虑上图(a)所示的隐式图，其中存在两个从初始状态无法达到的状态。在搜索算法中，这样的两个状态就不被考虑了，如上图(b)所示。但是动态规划由于是自底向上求解，所以就无法估计到这一点，因而遍历了全部的状态，如上图(c)所示。

一般说来，动态规划总要遍历所有的状态，而搜索可以排除一些无效状态。更重要的事搜索还可以剪枝，可能剪去大量不必要的状态，因此在空间开销上往往比动态规划要低很多。

如何协调好动态规划的高效率与高消费之间的矛盾呢？有一种折衷的办法就是记忆化算法。记忆化算法在求解的时候还是按着自顶向下的顺序，但是每求解一个状态，就将它的解保存下来，以后再次遇到这个状态的时候，就不必重新求解了。这种方法综合了搜索和动态规划两方面的优点，因而还是很有实用价值的。

§3.3 动态规划与网络流

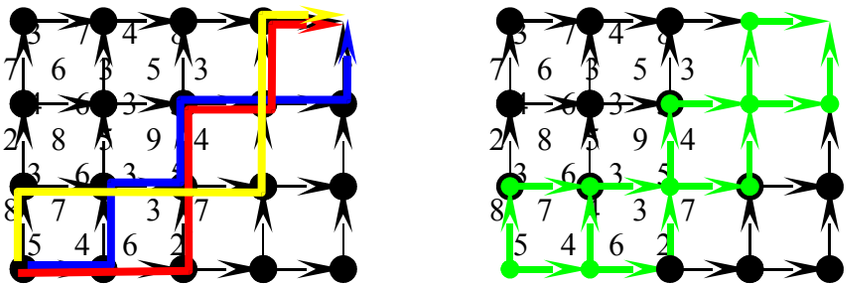
——动态规划是易设计易实现算法

由于图的关系复杂而无序，一般难以呈现阶段特征（除了特殊的图如多段图，或特殊的分段方法如 Floyd），因此动态规划在图论中的应用不多。但有一类图，它的点却是有序的，这就是有向无环图。

在有向无环图中，我们可以对点进行拓扑排序，使其体现出有序的特征，从而据此划分阶段。在有向无还图中求最短路径的算法[4]，已经体现出了简单的动态规划思想。但动态规划在图论中还有更有价值的应用。下面先看一个例子。

[例6] N 个人的街道问题：在街道问题（参见例 3）中，若有 N 个人要从左下角走向右上角，要求他们走过的边的总长度最大。当然，这里每个人也只能向右或向上走。下面是一个样例，左图是从出发地到目的地的三条路径，

右图是他们所走过的边，这些边的总长度为 5 + 4 + 3 + 6 + 3 + 3 + 5 + 8 + 8 + 7 + 4 + 5 + 9 + 5 + 3 = 78（不一定是最大）。



这个题目是对街道问题的又一次扩展。仿照街道问题的解题方法，我们仍然可以用动态规划来解决本题。不过这一次是 N 个人同时走，状态变量也就需要用 N 维来表示，。相应的，决策变量也要变成 N 维，$u_k=(u_{k,1},u_{k,2},\ldots,u_{k,N})$。状态转移方程不需要做什么改动：

$$\begin{cases} s_{k+1,j} := s_{k,j} + 1 + u_{k,j} & (k \le row) \\ s_{k+1,j} := s_{k,j} + u_{k,j} & (k > row) \end{cases} \quad (1 \le i \le N)$$

在写规划方程时，需要注意在第 k 阶段，N 条路径所走过的边的总长度的计算，在这里我就用 $g_k(s_k,u_k)$来表示了：

$$f_k(s_k) = \max_{u_{k,i}=0,1\ (1\le i\le N)} \left(f_{k-1}(T_k(s_k,u_k)) + g_k(s_k,u_k) \right)$$

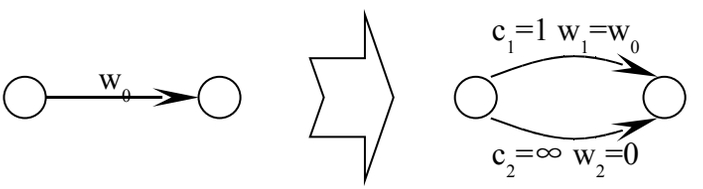
边界条件为 $f_1((1,1,\cdots,1)) = 0$

可见将原来的动态规划算法移植到这个问题上来，在理论上还是完全可行的。但是，现在的这个动态规划算法的时空复杂度已经是关于 N 的指数函数，只要 N 稍微大一点，这个算法就不可能实现了。

下面我们换一个思路，将 N 条路径看成是网络中一个流量为 N 的流，这样求解的目标就是使这个流的费用最大。但是本题又不同于一般的费用流问题，在每一条边 e 上的流费用并不是流量和边权的乘积 $f(e)\cdot w(e)$，而是用下式计算：

$$\begin{cases} w(e) & f(e)>0 \\ 0 & f(e)=0 \end{cases}$$

为了使经典的费用流算法适用于本题，我们需要将模型稍微转化一下：



如图，将每条边拆成两条。拆开后一条边上有权，但是容量限制为 1；另一条边没有容量限制，但是流过这条边就不能计算费用了。这样我们就把问题转化成了一个标准的最大费用固定流问题。

这个算法可以套用经典的最小费用最大流算法，在此就不细说了。（参见附录中的源程序）

这个例题是我仿照 IOI97 的“障碍物探测器”一题[6]编出来的。“障碍物探测器”比这一题要复杂一些，但是基本思想是相似的。类似的题目还有 99 年冬令营的“迷宫改造”[7]。从这些题目中都可以看到动态规划和网络流的联系。

推广到一般情况，任何有向无环图中的费用流问题在理论上说，都可以用动态规划来解决。对于流量为 N（如果流量不固定，这个 N 需要事先求出来）的费用流问题用 N 维的变量 $s_k=(s_{k,1},s_{k,2},\ldots,s_{k,N})$来描述状态，其中 $s_{k,i}\in V(1\le i\le N)$。相应的，决策也用 N 维的变量 $u_k=(u_{k,1},u_{k,2},\ldots,u_{k,N})$来表示，其中 $u_{k,i}\in E(s_{k,i})(1\le i\le N)$，$E(v)$表示指向 v 的弧集。则状态转移方程可以这样表示：

$s_{k-1,i} = u_{k,i}$ 的弧尾结点

规划方程为 $f_k(s_k) = \underset{u_{k,i} \in E(s_{k,i})}{\text{opt}} \left(f_k(T_k(s_k, u_k)) + \sum_{i=1}^{N} w(u_k) \right)$

边界条件为 $f_1((1,1,\cdots 1)) = 0$

但是，由于动态规划算法是指数级算法，因而在实现中的局限性很大，仅可用于一些 N 非常小的题目。然而在竞赛解题中，比如上面说到的 IOI97 以及 99 冬令营测试时，我们使用动态规划的倾向性很明显（“障碍物探测器”中，我们用的是贪心策略，求 N=1 或 N=2 时的局部最优解[8]）。这主要有两个原因：

1. 虽然网络流有着经典的算法，但是在竞赛中不可能出现经典的问题。如果要运用网络流算法，则需要经过一番模型转化，有时这个转化还是相当困难的因此在算法的设计上，灵活巧妙的动态规划算法反而要更为简单一些。

2. 网络流算法实现起来很繁，这是被人们公认的。因而在竞赛的紧张环境中，实现起来有一定模式的动态规划算法又多了一层优势。

正由于动态规划算法在设计和实现上的简便性，所以在 N 不太大时，也就是在动态规划可行的情况下，我们还是应该尽量运用动态规划。

§4 结语

本文的内容比较杂，是我几年来对动态规划的参悟理解、心得体会。虽然主要的篇幅讲的都是理论，但是根本的目的还是指导实践。

动态规划，据我认为，是当今信息学竞赛中最灵活、也最能体现解题者水平的一类解题方法。本文内容虽多，不能涵盖动态规划之万一。“纸上得来终觉浅，绝知此事要躬行。”要想真正领悟、理解动态规划的思想，掌握动态规划的解题技巧，还需要在实践中不断地挖掘、探索。实践得多了，也就能体会到渐入佳境之妙了。

动态规划，
算法之常，
运用之妙，

存乎一心。

【附录：部分试题与源程序】

1. “花店橱窗布置问题”试题

LITTLE SHOP OF FLOWERS

PROBLEM

You want to arrange the window of your flower shop in a most pleasant way. You have F bunches of flowers, each being of a different kind, and at least as many vases ordered in a row. The vases are glued onto the shelf and are numbered consecutively 1 through V, where V is the number of vases, from left to right so that the vase 1 is the leftmost, and the vase V is the rightmost vase. The bunches are moveable and are uniquely identified by integers between 1 and F. These id-numbers have a significance: They determine the required order of appearance of the flower bunches in the row of vases so that the bunch i must be in a vase to the left of the vase containing bunch j whenever i < j. Suppose, for example, you have bunch of azaleas (id-number=1), a bunch of begonias (id-number=2) and a bunch of carnations (id-number=3). Now, all the bunches must be put into the vases keeping their id-numbers in order. The bunch of azaleas must be in a vase to the left of begonias, and the bunch of begonias must be in a vase to the left of carnations. If there are more vases than bunches of flowers then the excess will be left empty. A vase can hold only one bunch of flowers.

Each vase has a distinct characteristic (just like flowers do). Hence, putting a bunch of flowers in a vase results in a certain aesthetic value, expressed by an integer. The aesthetic values are presented in a table as shown below. Leaving a vase empty has an aesthetic value of 0.

		VASES				
		1	2	3	4	5
Bunches	1 (azaleas)	7	23	-5	-24	16
	2 (begonias)	5	21	-4	10	23
	3 (carnations)	-21	5	-4	-20	20

According to the table, azaleas, for example, would look great in vase 2, but they would look awful in vase 4.

To achieve the most pleasant effect you have to maximize the sum of aesthetic values for the arrangement while keeping the required ordering of the flowers. If more than one arrangement has the maximal sum value,

any one of them will be acceptable. You have to produce exactly one arrangement.

ASSUMPTIONS

1 <= F <= 100 where F is the number of the bunches of flowers. The bunches are numbered 1 through F.
F <= V <= 100 where V is the number of vases.
-50 <= A_{ij} <= 50 where A_{ij} is the aesthetic value obtained by putting the flower bunch i into the vase j.

INPUT

The input is a text file named flower.inp.
The first line contains two numbers: F, V.
The following F lines: Each of these lines contains V integers, so that A_{ij} is given as the jth number on the (i+1)st line of the input file.

OUTPUT

The output must be a text file named flower.out consisting of two lines:
The first line will contain the sum of aesthetic values for your arrangement.
The second line must present the arrangement as a list of F numbers, so that the k’th number on this line identifies the vase in which the bunch k is put.

EXAMPLE

flower.inp:
3 5
7 23 -5 -24 16
5 21 -4 10 23
-21 5 -4 -20 20

flower.out:
53
2 4 5

EVALUATION

Your program will be allowed to run 2 seconds.
No partial credit can be obtained for a test case.

2. “钉子与小球”试题

有一个三角形木板,竖直立放，上面钉着 n(n+1)/2 颗钉子，还有(n+1)个格子（当 n=5 时如图 1）。每颗钉子和周围的钉子的距离都等于 d，每个格子的宽度也都等

于 d，且除了最左端和最右端的格子外每个格子都正对着最下面一排钉子的间隙。

让一个直径略小于 d 的小球中心正对着最上面的钉子在板上自由滚落，小球每碰到一个钉子都可能落向左边或右边（概率各 1/2），且球的中心还会正对着下一颗将要碰上的钉子。例如图 2 就是小球一条可能的路径。

我们知道小球落在第 i 个格子中的概率 $p_i = C_n^i / 2^n = \frac{n!}{i!(n-i)!} \Big/ 2^n$，其中 i 为格子的编号，从左至右依次为 0,1,...,n。

现在的问题是计算拔掉某些钉子后，小球落在编号为 m 的格子中的概率 p_m。假定最下面一排钉子不会被拔掉。例如图 3 是某些钉子被拔掉后小球一条可能的路径。

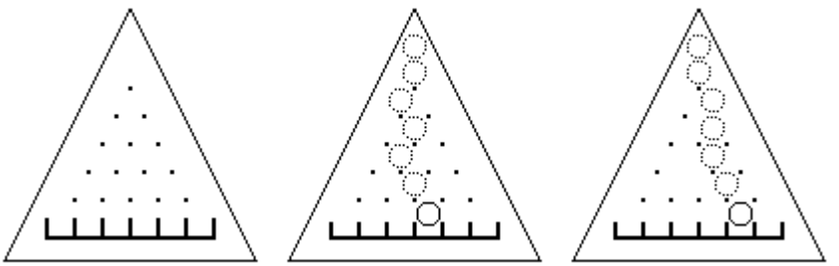


图 1. 图 2. 图 3

输入

第 1 行为整数 n（2<=n<=50）和 m（0<=m<=n）。

以下 n 行依次为木板上从上至下 n 行钉子的信息，每行中‘*’表示钉子还在，‘.’表示钉子被拔去，注意在这 n 行中空格符可能出现在任何位置。

输出

仅一行，是一个既约分数(0 写成 0/1)，为小球落在编号为 m 的格子中的概率 p_m。

既约分数的定义：A/B 是既约分数，当且仅当 A、B 为正整数且 A 和 B 没有大于 1 的公因子。

样例输入

```
5 2
*
* .
* * *
* . * *
* * * * *
```

样例输出

```
7/16
```

3. 例 2“花店橱窗布置问题”方法 1 的源程序

```
program IOI99_LittleShopOfFlowers;
var
    F,V :byte;  {花束和花瓶的数目}
```

```
    A   :array [1..100,1..100] of shortint; {A[i,j]花束i放在花瓶j中的美学值}
    Best:array [0..100,0..100] of integer;
        {Best[i,j]前i束花放在前j个花瓶中的最优解}
    Previous:array [1..100,1..100] of byte;
        {Previous[i,j]在Best[i,j]的最优解中，花束i-1的位置}

procedure ReadIn;   {读入}
var
    i,j :byte;
begin
    reset(input);
    readln(F,V);
    for i:=1 to F do
    begin
        for j:=1 to V do
            read(A[i,j]);
        readln;
    end;
    close(input);
end;

procedure Work; {规划主过程}
var
    i,j,t   :byte;
begin
    fillchar(Best[0],sizeof(Best[0]),0); {边界条件}
    for i:=1 to F do
        for j:=i to V+i-F do
        begin
            Best[i,j]:=low(Best[i,j]);
            for t:=i-1 to j-1 do{枚举花束i-1的位置}
                if Best[i-1,t]>Best[i,j] then
                begin   {更新当前最优解}
                    Best[i,j]:=Best[i-1,t];
                    Previous[i,j]:=t;
                end;
            inc(Best[i,j],A[i,j]);
        end;
end;

procedure Print;{打印最优解}
var
    i,j :byte;
    Put :array [1..100] of byte;
```

```
begin
    i:=F;
    for j:=F+1 to V do  {选择全局最优解}
        if Best[F,j]>Best[F,i] then
            i:=j;   {i最后一束花的位置}
    rewrite(output);
    writeln(Best[F,i]);
    for j:=F downto 1 do
    begin
        Put[j]:=i;
        i:=Previous[j,i];
    end;
    for i:=1 to F do
        write(Put[i],' ');
    writeln;
    close(output);
end;

begin
    assign(input,'flower.inp');
    assign(output,'flower.out');
    ReadIn;
    Work;
    Print;
end.
```

4. 例2“花店橱窗布置问题”方法2的源程序

```
program IOI99_LittleShopOfFlowers;
var
    F,V :byte;  {花束和花瓶的数目}
    A   :array [1..100,1..100] of shortint; {A[i,j]花束i放在花瓶j中的美学值}
    Best:array [0..100,0..100] of integer;
        {Best[i,j]前i束花放在前j个花瓶中的最优解}
    Choice  :array [0..100,0..100] of boolean;
        {Choice[i,j] 花束i是否放在花瓶j中}

procedure ReadIn;   {读入}
var
    i,j :byte;
begin
    reset(input);
    readln(F,V);
    for i:=1 to F do
```

```
    begin
        for j:=1 to V do
            read(A[i,j]);
        readln;
    end;
    close(input);
end;

procedure Work; {规划主过程}
var
    i,j :byte;
begin
    fillchar(Best[0],sizeof(Best[0]),0); {边界条件}
    for i:=1 to F do
    begin
        Best[i,i]:=Best[i-1,i-1]+A[i,i];{唯一的选择}
        Choice[i,i]:=true;
        for j:=i+1 to V+i-F do
        begin
            Choice[i,j]:=Best[i-1,j-1]+A[i,j]>Best[i,j-1];
            if Choice[i,j]
            then Best[i,j]:=Best[i-1,j-1]+A[i,j] {i 放在 j 中}
            else Best[i,j]:=Best[i,j-1];{i 不放在 j 中}
        end;
    end;
end;

procedure Print;{打印最优解}
var
    i,j :byte;
    Put :array [1..100] of byte;{Put[i]花束 i 所在的花瓶}
begin
    rewrite(output);
    writeln(Best[F,V]);
    i:=F;
    for j:=V downto 1 do{倒推求 Put}
        if Choice[i,j] then
        begin
            Put[i]:=j;
            dec(i);
        end;
    for i:=1 to F do
    begin
        write(Put[i]);
```

```
        if i<F then write(' ');
    end;
    writeln;
    close(output);
end;

begin
    assign(input,'flower.inp');
    assign(output,'flower.out');
    ReadIn;
    Work;
    Print;
end.
```

5. 例3“街道问题”的扩展，找两条不重叠的最短路径

```
program N_Street;
const
    MaxSize=90; {地图的最大尺寸}
type
    TPlanarArr=array [1..MaxSize,1..MaxSize] of integer;
var
    Row,Col :byte;       {地图的行数和列数}
    Best,B1 :TPlanarArr;{动态规划中本阶段和上阶段的最优指标函数}
    Street  :array [0..1] of TPlanarArr;{地图中横向和纵向的边}

procedure ReadIn;   {读入}
var
    i,j :byte;
begin
    reset(input);
    readln(Row,Col);
    for i:=1 to Row do
    begin
        for j:=1 to Col-1 do
            read(Street[0,i,j]);
        readln;
    end;
    for j:=1 to Col do
    begin
        for i:=1 to Row-1 do
            read(Street[1,i,j]);
        readln;
    end;
    close(input);
```

```
end;

procedure Work; {规划过程，和文章中不同，这里是逆序求解}
var
    k,       {阶段}
    s1,s2,   {状态（二维）}
    u1,u2,   {决策（二维）}
    t1,t2,   {由s和t导出的上阶段的状态（二维）}
    m,       {本阶段的状态总数}
    m1,m2    :byte;  {Row和Col当中较小的数和较大的数}
    e1,e2    :integer;    {u1,u2对应的两条边长}
    function GetET(s,u:byte; var e:integer; var t:byte):boolean; {求e,t}
    begin
        GetET:=false;
        if  (k>=Row) and (s=1) and (u=1) or
            (k>=Col) and (s=m) and (u=0) then
            exit;   {判断越界}
        GetET:=true;
        if k<Row{根据k分两种情况}
        then begin
            e:=Street[u,k+1-s,s];
            t:=s+1-u;
        end
        else begin
            e:=Street[u,Row+1-s,k-Row+s];
            t:=s-u;
        end;
    end;
begin
    if Row<Col
    then begin
        m1:=Row; m2:=Col;
    end
    else begin
        m1:=Col; m2:=Row;
    end;
    Best[1,1]:=0;
    for k:=Row+Col-2 downto 1 do{逆序求解}
    begin
        if k<m1
        then m:=k
        else if k>m2
            then m:=Row+Col-k
            else m:=m1;
```

```
        B1:=Best;
        for s1:=1 to m do
            for s2:=1 to m do
            begin
                Best[s1,s2]:=$7000; {初始化为一个较大的数}
                for u1:=0 to 1 do
                    if GetET(s1,u1,e1,t1) then
                        for u2:=0 to 1 do
                            if ((s1<>s2) or (u1<>u2)) and
                                GetET(s2,u2,e2,t2) and
                                (B1[t1,t2]+e1+e2<Best[s1,s2]) then
                                Best[s1,s2]:=B1[t1,t2]+e1+e2;    {更新最优解}
            end;
    end;
end;

begin
    assign(input,'input.txt');
    assign(output,'output.txt');
    ReadIn;
    Work;
    rewrite(output);
    writeln(Best[1,1]);
    close(output);
end.
```

6. 例4“mod 4最优路径问题”的源程序

```
program BestPath_mod4;
const
    MaxN=100;    {点数上限}
    MaxE=100;    {两点之间的边数上限}
var
    N    :byte;  {点数}
    E    :array [1..MaxN,0..MaxE] of integer;
         {E[i,0]第i点到第i+1点之间的边数}
         {E[i,j]第i点到第i+1点之间的第j条边长}
    Best:array [1..MaxN,0..3] of boolean;
         {Best[k,s]从第1点到第k点长度mod 4的值为s的路径是否存在}

procedure ReadIn;    {读入}
var
    i,j :byte;
begin
    reset(input);
```

```
    readln(N);
    for i:=1 to N-1 do
    begin
        read(E[i,0]);
        for j:=1 to E[i,0] do
            read(E[i,j]);
        readln;
    end;
    close(input);
end;

procedure Work; {主递推过程}
var
    k,s,u   :byte;  {阶段、状态、决策}
begin
    fillchar(Best,sizeof(Best),false);
    Best[1,0]:=true;{边界条件}
    for k:=2 to N do
        for s:=0 to 3 do
            for u:=1 to E[k-1,0] do
                Best[k,s]:=Best[k,s] or Best[k-1,($10000+s-E[k-1,u]) mod 4];
                    {加上10000(h)为防止出现负数}
end;

procedure Print;{这里只输出最优路径mod 4的值，路径就不输出了}
var
    i   :byte;
begin
    for i:=0 to 3 do
        if Best[N,i] then
            break;
    rewrite(output);
    writeln(i);
    close(output);
end;

begin
    assign(input,'input.txt');
    assign(output,'output.txt');
    ReadIn;
    Work;
    Print;
end.
```

7. 例5“钉子与小球”的源程序

```
{$A+,B-,D+,E+,F-,G+,I+,L+,N+,O-,P-,Q-,R-,S-,T-,V+,X+,Y+}
{$M 65520,0,655360}
program Ball;
var
    N,M :byte;
    Nail:array [1..50,1..50] of byte;
    Poss:array [0..50,0..50] of comp;

procedure ReadIn;
var
    i,j :byte;
    c   :char;
begin
    reset(input);
    readln(N,M);
    for i:=1 to N do
        for j:=1 to i do
        begin
            repeat
                read(c);
            until c in ['*','.'];
            if c='*'
            then Nail[i,j]:=1
            else Nail[i,j]:=0;
        end;
    close(input);
end;

procedure Work;
var
    i,j :byte;
begin
    fillchar(Poss,sizeof(Poss),0);
    Poss[0,0]:=1;
    for i:=1 to N do
        Poss[0,0]:=Poss[0,0]*2;
    for i:=0 to N-1 do
        for j:=0 to i do
            if Nail[i+1,j+1]=0
            then Poss[i+2,j+1]:=Poss[i+2,j+1]+Poss[i,j]
            else begin
                Poss[i+1,j]:=Poss[i+1,j]+Poss[i,j]/2;
                Poss[i+1,j+1]:=Poss[i+1,j+1]+Poss[i,j]/2;
```

```
            end;
end;

procedure Print;
var
    a,b :comp;
begin;
    rewrite(output);
    a:=Poss[N,M];
    if a=0
    then writeln('0/1')
    else begin
        b:=a/2;
        while b*2=a do
        begin
            a:=b; b:=a/2;
            Poss[0,0]:=Poss[0,0]/2;
        end;
        writeln(a:0:0,'/',Poss[0,0]:0:0);
    end;
    close(output);
end;

begin
    assign(input,'input.txt');
    assign(output,'output.txt');
    ReadIn;
    Work;
    Print;
end.
```

8. 例6的源程序,“N个人的街道问题”，网络流算法

```
{$A+,B-,D+,E+,F-,G+,I+,L+,N+,O-,P-,Q+,R+,S+,T-,V-,X+,Y+}
{$M 65520,0,655360}
program N_Street;
const
    MaxSize=100;{地图的最大尺寸}
var
    Row,Col,     {地图的行数和列数}
    N   :byte;  {人数N}
    Horiz,  {横向的边}
    Vert:   {纵向的边}
        array [1..MaxSize,1..MaxSize] of word;
    F   :array [1..MaxSize,1..MaxSize,0..1] of byte;
```

```
            {F[i,j,d]从点(i,j)向方向d去的流量}
    Result  :word;  {最优结果}

procedure ReadIn;   {读入}
var
    i,j :byte;
begin
    reset(input);
    readln(Row,Col,N);
    for i:=1 to Row do
    begin
        for j:=1 to Col-1 do
            read(Horiz[i,j]);
        readln;
    end;
    for j:=1 to Col do
    begin
        for i:=1 to Row-1 do
            read(Vert[i,j]);
        readln;
    end;
    close(input);
end;

procedure Work;
var
    k,i,j   :byte;
    W   :array [1..MaxSize,1..MaxSize] of word;
    P   :array [1..MaxSize,1..MaxSize] of byte;
    procedure GetMaxPath;   {求最大费用路，用Bellman-Ford迭代法}
    var
        i,j :byte;
        t   :integer;
        Finish  :boolean;   {迭代终止标志}
    begin
        fillchar(W,sizeof(W),0);
        repeat
            Finish:=true;
            for i:=Row downto 1 do
                for j:=Col downto 1 do
                begin
                    if i<Row then
                    begin   {横向边}
                        t:=W[i+1,j];
```

```
        if F[i,j,1]=0 then
            inc(t,Vert[i,j]);
        if t>W[i,j] then
        begin
            W[i,j]:=t;
            P[i,j]:=1;
            Finish:=false;
        end;
        if F[i,j,1]>0 then
        begin   {横的逆向边}
            t:=W[i,j];
            if F[i,j,1]=1 then
                dec(t,Vert[i,j]);
            if t>W[i+1,j] then
            begin
                W[i+1,j]:=t;
                P[i+1,j]:=3;
                Finish:=false;
            end;
        end;
    end;
    if j<Col then
    begin   {纵向边}
        t:=W[i,j+1];
        if F[i,j,0]=0 then
            inc(t,Horiz[i,j]);
        if t>W[i,j] then
        begin
            W[i,j]:=t;
            P[i,j]:=0;
            Finish:=false;
        end;
        if F[i,j,0]>0 then
        begin   {纵的逆向边}
            t:=W[i,j];
            if F[i,j,0]=1 then
                dec(t,Horiz[i,j]);
            if t>W[i,j+1] then
            begin
                W[i,j+1]:=t;
                P[i,j+1]:=2;
                Finish:=false;
            end;
        end;
```

```
                    end;
                end;
        until Finish;
    end;
begin
    fillchar(F,sizeof(F),0);
    Result:=0;
    for k:=1 to N do{求的是流量为N的最大费用流}
    begin
        GetMaxPath;
        inc(Result,W[1,1]); {累计费用}
        i:=1;j:=1;
        repeat  {更新最大费用路上的流量}
            case P[i,j] of
                0:  begin
                        inc(F[i,j,0]);
                        inc(j);
                    end;
                1:  begin
                        inc(F[i,j,1]);
                        inc(i);
                    end;
                2:  begin
                        dec(j);
                        dec(F[i,j,0]);
                    end;
                3:  begin
                        dec(i);
                        dec(F[i,j,1]);
                    end;
            end;
        until (i=Row) and (j=Col);
    end;
end;

begin
    assign(input,'input.txt');
    assign(output,'output.txt');
    ReadIn;
    Work;
    rewrite(output);
    writeln(Result);
    close(output);
end.
```

【参考文献】

1. 胡运权 郭耀煌 《运筹学教程》 清华大学出版社 1998.6.

2. 张辰《“花店橱窗布置问题”解题报告》（2000集训队第一次作业）

3. 张辰《“机场跑道”解题报告》（2000集训队第一次作业）

4. 潘金贵 等 《现代计算机常用数据结构和算法》 南京大学出版社 1994.3.

5. 解可新 韩立兴 林友联 《最优化方法》 天津大学出版社 1997.1.

6. 《“障碍物探测器”试题》 IOI’98中国集训队冬令营特刊1998.1.

7. 《“迷宫改造”试题》 信息学奥林匹克1999年第1期

8. 倪兆中 《“障碍物探测器”题解》 信息学奥林匹克1998年第1-2期