算法分析与设计——动态规划法的应用（C++版）

内容提要：动态规划法是算法中一种比较常用的方法。本文就是主要介绍这种方法的基本特征和一些应用。动态规划的实质是分治思想和解决冗余，因此，动态规划是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。它的主要应用为最短路径问题、最佳折半查找树、资源分配问题、多机系统的可靠性设计、最长公共子序列问题 LCS、货郎担问题等。

关键字：动态规划、最短路径问题、最佳折半查找树、资源分配问题、货郎担问题

引言：动态规划（dynamic programming）是一种算法设计技术，它也是运筹学的一个分支，是一种求解决策过程（decision process）最优的通用方法，它是在 20 世纪 50 年代初由美国数学家 R.E.Bellman 所提出。动态规划作为一种重要的工具在应用数学中的价值被大家认同以后，在计算机科学中，人们不仅用它来解决特定类型的最优问题而且最终把它作为一种通用的算法设计技术来使用。动态规划法在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其他方法求解更为方便。动态规划既适用于以时间划分阶段的动态过程的优化问题，也适用于一些与时间无关的表态规划（如线性规划、非线性规划），只要人为地引进时间因素，把它视为多阶段决策过程，也可以用动态规划方法方便地求解。所谓多阶段决策过程是指把一个问题看做是一个前后关联具有链状结构的多阶段过程，这种问题又称为多阶段决策最优化问题。

正文：

1　动态规划法的概念

动态规划的指导思想是，在各种情况下，列出所有可能的局部解，然后按照某些条件，从局部解（或中间结果）中选出那些有可能产生最优解的结果而舍弃其他的可能解，从而大大减少了计算量。

动态规划遵循所谓“最佳原理”的原则，即不论前面的状态和策略如何，后面的最优策略只取决于由最初策略所确定的当前状态。

动态规划的实质是分而治之思想和排队冗余策略，因此，动态规划方法是将问题实例分解为相似的、更小的子问题，存储子问题的解从而避免子问题的重复计算，来解决最优化问题的一种算法策略。据此可知，动态规划法与分治法和贪心法相似，它们都是将问题实例分解为更小的、相似的子问题，并通过求解子问题来产生一个全局最优解。其中贪心法的当前选择可能与已经作出的所有选择有关，但与有待于做出的选择和子问题无关，因此贪心法是自顶向下，一步一步地作出贪心选择。而分治法中是将问题分解为各个独立的子问题，因此一旦递归地求出各子问题的解后，便可自下而上地将子问题的解合并成该问题的解。但分治法的缺陷是，独立子问题的划分比较困难，很可能会做许多不必要的工作，重复求解公共子问题，另外，如果当前选择可能要依赖子问题的解时，则很难通过局部的贪心策略达到全局最优解。

采用动态规划可以克服上述问题。该方法主要应用于最优化问题的求解，这类问题可能会有多种可能的解，每个解对应一个值，动态规划可以找出其中最优（最大或最小）值的解。若存在若干个取最优值的解，它只取一个最优值。在求解过程中，该方法也是通过求解局部子问题的解来达到找出全局最优解的目的，但与分治法和贪心法不同在于，动态规划允许子问题不独立，即子问题中可以包含公共的子问题，也允许其通过自身子问题的解作出选择，该方法对每一个子问题只解一次，并将结果保存起来，以后遇到该子问题时可以直接引用，避免每次碰到时都要重复计算。

因此，动态规划法所针对的问题有一个显著的特征，即它所对应的子问题树中的子问题呈现大量的重复。动态规划法的关键就在于，对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。

1.1 动态规划模型的基本要素

动态规划可以看成是一个多阶段决策过程化问题的求解模型，它通常包含以下要素：

1. 阶段

阶段（step）是对整个过程的自然划分。通常存在按时间顺序和按空间特征来划分阶段这两种方法，针对不同的问题选择合适的划分方法，以便按阶段的次序解优化问题。

2. 状态

对于一个问题，所有可能到达的情况，包括初始情况和目标情况等，称为该问题的状态（state）。

描述状态的变量称状态变量（state variable），状态变量的取值是其所对应状态的当前解。变量允许取值的范围称允许状态集合（set of admissible states）。根据过程演变的具体情况，状态变量可以是离散的或连续的。为了计算的方便有时将连续变量离散化；为了分析的方便有时又将离散变量视为连续的。

3. 决策

当某一阶段的状态确定后，可以作出各种选择从而达到下一阶段的某个状态，这种选择手段称为决策（decision），在最优控制问题中也称为控制。

用于描述某状态当前所做出的决策的变量称为决策变量（decision variable）。决策变量允许取值的范围称允许决策集合（set of admissible decisions）。

4. 策略

策略（policy）是一个决策的集合，在解决问题的时候，将一系列决策记录下来，就是一个策略，其中满足某些条件的策略称之为最优策略（policy of optimality）。

5. 状态转移方程

状态转移方程（equation of state）描述了状态变量之间的数学关系，在确定性过程中，一旦某阶段的状态和决策为已知，下阶段的状态便完全确定。

6. 指标函数和最优值函数

指标函数（objective function）是衡量过程优劣的数量指标，它是关于策略的数量函数，能够用动态规划解决的问题的指标函数应具有可分离性，这一性质保证了最优化原理（principle of optimality）的成立，是动态规划的适用前提。

根据状态转移方程，指标函数还可以表示为状态和策略的函数。在状态给定时指标函数对相应策略的最优值称为最优值函数（optimal value function）。

1.2 动态规划的基本定理和基本方程

在研究动态规划方法的初期，从简单逻辑出发给出了所谓最优性原理，然后在最优策略存在的前提下导出基本方程，再由这个方程求解最优策略。后来在动态规划的实际应用中发现，最优性原理不是对任何决策过程普遍成立，它与基本方程不是无条件等价，二者之间也不存在任何确定的蕴含关系。基本方程在动态规划中起着更为本质的作用。

1. 最优化原理

最优化原理是由美国数学家 R.Bellman 在 1951 年所提出，描述如下：一个最优化策略具有这样的性质，不论初始状态和初始决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略对于其初态和终态而言也必须是最优的。满足最优化原理的问题又称其具有最优子结构性质。

例，如图 1 所示，若路线 e_1 和 e_2 是 V_0 到 V_2 的最优路径，则根据最优化原理，路线 e_2 必是从 V_1 到 V_2 的最优路线。

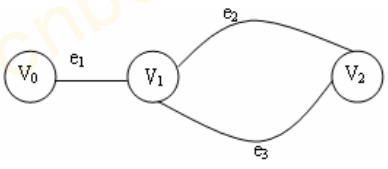


图 1

这可用反证法证明：假设有另一路径 e_3 是 V_1 到 V_2 的最优路径，则 V_0 到 V_2 的路线取 e_1 和 e_3 比 e_1 和 e_2 更优，得出矛盾。从而证明 e_2 必是 V_1 到 V_2 的最优路径。

最优化原理是动态规划的基础，任何问题，如果没有最优化原理的支持，就不可能用动态规划方法计算。

2. 动态规划法的基本定理如下：

对于初始状态 $x_1 \in X_1$，策略 $p_{1n}^* = \{u_1^*, \ldots, u_n^*\}$ 是最优策略的充要条件是对于任意的 k，1<k<=n，有：

$$V_{1n}(x_1, p_{1n}^*) = \varphi\left(\underset{p_{1k-1} \in p_{ik-1}(x_1)}{opt} [V_{1k-1}(x_1, p_{1k-1})], \underset{p_{kn} \in p_{kn}(x_k)}{opt} [V_{kn}(x_k, p_{kn})] \right) \quad (1)$$

3. 动态规划法基本定理的推论如下：

若 $p_{1n}^* \in p_{1n}(x_1)$ 是最优策略，则对于任意的 k，有 1<k<n，它的子策略 p_{kn}^* 对于由 x_1 和 $p_{1,k-1}^*$ 确定的以 x_k^* 为起点的第 k 到 n 后部子过程而言，也是最优策略。

上述推论称为最优化原理，它给出了最优策略的必要条件，通常略述为：不论过去的状态和决策如何，对于前面的决策形成的当前的状态而言，余下的各个决策必定构成最优策略。

根据基本定理的推论可以得到动态规划的基本方程：

$$\begin{cases} f_k(x_k) = \underset{u_k \in U_k(x_k)}{opt} \{\varphi(V_k(x_k, u_k), f_{k+1}(x_{k+1}))\} \quad x_{k+1} = T_k(x_k, u_k), k = 1, 2, \cdots, n \\ f_{n+1}(x_{n+1}) = \delta(x_{n+1}) \end{cases} \quad (2)$$

其中 $f_{n+1}(x_{n+1}) = \delta(x_{n+1})$ 是决策过程的终端条件，δ 为一个已知函数。当 x_{n+1} 只取固定的状态时称固定终端；当 x_{n+1} 可在终端集合 X_{n+1} 中变动时称自由终端。最终要求的最优指标函数满足（3）式：

$$opt\{V_{1n}\} = \underset{x_1 \in X_1}{opt} \{f(x_1)\} \quad (3)$$

（2）式是一个递归公式，如果目标状态确定，当然可以直接利用该公式递归求出最优值，但是一般在实际应用中通常将该递归公式改为递推公式求解，这样一般效率会更高一些。

1.3 动态规划法的基本步骤

设计一个标准的动态规划算法，通常可按以下几个步骤进行：

（1）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

（2）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。

（3）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，如果确定了决策，状态转移方程也就写出来了。

（4）写出规划方程（包括边界条件）：动态规划的基本方程是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。

但是，实际应用当中经常不显式地按照上面步骤设计动态规划，而是按以下几个步骤进行：

（1）分析最优解的性质，并刻划其结构特征。

（2）递归地定义最优值。

（3）以自底向上的方式或自顶向下的记忆化方法（备忘录法）计算出最优值。

（4）根据计算最优值时得到的信息，构造一个最优解。

步骤（1）～（3）是动态规划算法的基本步骤。在只需要求出最优值的情形，步骤（4）可以省略，若需要求出问题的一个最优解，则必须执行步骤（4）。此时，在步骤（3）中计算最优值时，通常需记录更多的信息，以便在步骤（4）中，根据所记录的信息，快速地构造出一个最优解。

1.4 动态规划与其他算法的比较

动态规划可以看作求解决策 $u_1, u_2, \ldots, u_n$，使指标函数 $f(x, u_1, u_2, \ldots, u_n)$ 达到最优（最大或最小）的极值问题，存在状态转移方程、端点条件以及允许状态集、允许决策集等约束条件。动态规划与

静态规划（线性和非线性规划等）所研究的对象本质上都是在若干约束条件下的函数极值问题。原则上，这两种规划在很多情况下可以相互转换。

动态规划与静态规划相比它的优越性在于：

首先，采用动态规划能够得到全局最优解。由于约束条件所确定的约束集合的复杂性，使得即使指标函数非常简单，用非线性规划方法也很难求出全局最优解。而动态规划方法把全过程化为一系列结构相似的子问题，每个子问题的变量个数大大减少，由之所确定的约束集合相对简单许多，易于得到全局最优解。尤其是对于约束集合、状态转移和指标函数不能用分析形式给出的优化问题，可以对每个子过程用枚举法求解，而约束条件越多，决策的搜索范围越小，求解也越容易。对于这类问题，动态规划通常是求全局最优解的惟一方法。

其次，采用动态规划可以得到一系列的解。与非线性规划只能得到全过程的一个最优解不同，动态规划得到的是全过程及所有后部子过程的各个状态的一系列解。有些实际问题需要知道所有的解，当最优策略由于某些原因不能实现时，就可以在其中找出次优解，即使不需要，它们在分析最优策略和最优值对于状态的稳定性时也是很有用的。

再有，能够利用经验提高求解效率。如果实际问题本身就是动态的，由于动态规划方法反映了过程逐段演变的前后联系和动态特征，在计算中可以利用实际知识和经验提高求解效率。比如在策略迭代法中，实际经验能够帮助选择较好的初始策略，提高收敛速度。

由于动态规划没有统一的标准模型，也没有构造模型的通用方法，甚至还没有判断一个问题能否构造动态规划模型的具体准则，大部分情况只能够凭经验，针对不同类的问题，构造具体的模型，不存在一种万能的动态规划模型。但是可能通过对一些有代表性的问题采用动态规划思想进行分析，来掌握这一设计方法。

现在已经对动态规划问题有了初步的了解，现在来谈谈动态规划法的应用。

2　计算二项式系数

计算二项式系数是把动态规划应用于非最优化问题的一个标准例子。回忆一下在初等排列组合中学过的知识，二项式系数，记作 C(n, k)，是来自于一个 n 元素集合的 k 元素组合（子集）的数量（0≤k≤n）。“二项式系数”这个名字来源于这些数字，出现在二项式公式之中：

$$(a+b)^n=C(n,0)a^n+...+C(n,i)a^{n-i}b^i+...+C(n,n)b^n$$

在二项式系数的多种特性之中，只关心两种：

$$C(n, k)=C(n-1, k-1)+C(n-1, k)，\text{当 } 0<k<n \text{ 时} \qquad (4)$$

$$C(n, k)=1，\text{当 } k=0 \text{ 或 } k=n \text{ 时} \qquad (5)$$

计算 C(n-1, k-1)和 C(n-1, k)是两个较小的交叠问题，递推式（4）的特点就是以这样的形式来表示 C(n, k)的计算问题，使得可以用动态规划技术来对它求解。为了做到这一点，把二项式系数记录在一张 n+1 行 k+1 列的表中，行从 0 到 n 计数，列从 0 到 k 计数。如图 2 所示。

	0	1	2	...	k-1	k
0	1					
1	1	1				
2	1	2	1			
.						
.						
.						
k	1					1
.						
.						
.						
n-1	1				C(n-1,k-1)	C(n-1,k)
n						C(n,k)

图 2　二项式系数记录

为了计算 C(n, k)，一行接一行填充上图 2 中的表，从行 0 开始，到行 n 结束。每一行（0≤i≤n）

从左向右填充，第一个数字填 1，因为 C(n, 0)=1。行 0 直到行 k 也是以 1 结束在表的主对角线上：当（0≤i≤k）时，C(I, i)=1。用公式（4）计算其他单元格的值，即把前一行前一列那个单元格的值和前一行当前列那个单元格的值相加（帕斯卡三角形），是一种令人着迷的数学结构，常常会与组合要领一起来研究，大家对它有所了解的话，就会发现，这里得到的就是这种三角形。下面的代码实现了这个算法：

该算法的时间效率如何呢？显然，该算法的基本操作是加法，所以把 A(n,k)记作该算法在计算 C(n,k)时所作的加法总次数。在这里用公式（4）计算每个单元格只需要一次加法，因为表格的前 k+1 行构成了一个三角形，而余下的 n-k 行构成了一个矩形，必须把求和表达式 A(n,k)分成两个部分：

$$A(n,k)=\sum_{i=1}^{k}\sum_{j=1}^{i-1}1+\sum_{i=k+1}^{n}\sum_{j=1}^{k}1=\sum_{i=1}^{k}(i-1)+\sum_{i=k+1}^{n}k$$

$$=(k-1)k/2+k(n-k)=O(nk)$$

3　最佳折半查找树

人们会经常遇到这样一类问题，已有某个集合 S，不时地要将一些元素插入集合 S 或从 S 中删除某些元素；有时也需要确知某元素是否在 S 中。假定 S 中的元素取自有线性序的某个大全集，就可以把以上几个问题抽象为在集合 S 上执行插入、查找、删除组成的操作序列。

这几种运算往往是相互穿插组合并交替进行的，其共同特点是要求查找某个元素的准确位置，或者从 S 中删除，或者将其插入 S 中。表和队列对其查找，删除或插入都不甚方便。一种称为折半查找树（Binary search tree）的结构对这些运算比较适用。

定义：关于集合 S 的一棵折半查找树是一棵加标二叉树（Labeled binary tree），它的每个顶点 v 用 S 中的一个元素 l(v)标定，使得:

（1）对于每个元素 a∈S，恰有一个顶点 v 满足 l(v)=a。

（2）在以 v 为根的子树中，v 的左子树中的任何顶点 u 的标号数 l(u)>l(v)；v 的右子树中的任何顶点 w 的标号数 l(w)＜l(v)。

要确定某个元素 a 是否在一棵折半查找树表示的集合 S 中，首先应将元素 a 与根节点元素比较，相同则 a∈S。如果 a 小于根节点的元素，就将 a 和这个根的左儿子比较；如果 a 大于根节点的元素，就拿 a 和这个根的右儿子比较。这样递归地查找下去，最终能确定，这种方法叫做折半查找。

在一棵折半查找树上查找元素 a 的算法如下所示：

在一棵折半查找树上搜索一个元素 a 所需的时间，在最坏情况下，与树的高度成正比。如果希望最坏情况下的查找次数少， 就应当使查找树的高度最小。这样，只要使任何顶点为根的子树中，这个根的左右子树的高度尽可能相等（最多差 1），就能达到目的。也就是使得以任何顶点为根的树中，其左右子树中的节点数尽可能相等。

设 a_1、a_2、…、a_n 是集合 S 中按递增序排列的元素。

p_i 是查找操作中查找 a_i 的概率。

q_0 是查找操作中查找 a 的概率，其中 $a<a_1$

q_i 是查找操作中查找 a 的概率，其中 $a_i<a<a_i+1$

q_n 是查找操作中查找 a 的概率，其中 $a>a_n$

给折半查找树添上 n+1 片虚叶，并命名为 0，1，2，…，n

则虚叶 0 表示 $a<a_1$

虚叶 i 表示 $a_i<a<a_i+1$

虚叶 n 表示 $a>a_n$

若查找的元素为 a：

若 $a=a_i$，比较次数为 a_i 的深度。

若 $a_i<a<a_i+1$，比较次数为虚叶 i 的深度。

若 $a>a_n$，比较次数为虚叶 N 的深度。

因此，折半查找树的平均深度即是折半查找的平均时间复杂性，为：

$$\sum_{i=1}^{n} p_i * (depth(a_i)+1) + \sum_{j=0}^{n} q_j * depth(j)$$

3.1 查找树的期望深度

设 $a_1, a_2, \ldots, a_n$ 是集合 S 中按递增顺序排列的元素，p_i 是 σ 中指令 MEMBER(a_i, S)出现的概率。设 p_0 是指令 MEMBER(a_0, S)在 σ 中出现的概率，其中 $a < a_1$；p_i 是指令 MEMBER(a_i, S)在 σ 出现的概率，其中 $a_i < a < a_{i+1}$，$1 \leqslant i \leqslant n$；$p_n$ 是对于 $a > a_n$，MEMBER(a, S)在 σ 中出现的概率。为了定义一棵折半查找树的耗费，给一棵折半查找树添上 n+1 片虚拟叶来表示集合 U-S 中的元素，并命名为 0, 1, 2, …, n。在执行指令序列 σ 时，元素落到这 n+1 片虚拟叶的概率分别为 $q_0, q_1, \ldots, q_n$。

这里，

$$q_0 + \sum_{i=1}^{n} (p_i + q_i) = 1$$

如果元素 a=l(v)，那么，执行指令 MEMBER(a, S)时所访问过的顶点个数比顶点 v 的深度多 1。如果 $a \notin S$ 且 $a_i < a < a_{i+1}$，那么执行指令 MEMBER(a, S)时所记问过的顶点个数等于虚拟叶 i 的深度。所以一棵折半查找树的期望耗费（或期望深度）可以定义为：

$$\sum_{i=1}^{n} p_i \times (DEPTH(a_i)+!) + \sum_{i=0}^{n} q_i \times DEPTH(j)$$

如果对集合 S 构造了一棵有最小期望耗费的折半查找树 T，就可以使用在一棵折半查找树上查找元素 a 的算法实现对各元素的查找，且使执行 MEMBER 指令序列 σ 的期望比较次数最小。

当给定 p_i 和 q_j 之后，需要构造出一棵最佳折半查找树。如果能确定最佳查找树的根元素 a_i，然后分别构造出 a_i 的最佳左子树和最佳右子树，问题就得到解决，a_i 的左右子树的构造方法是类似的。看来，还没有一个容易的方法来确定根元素 a_i，除非对 n 个元素都计算一下它们的左右子树，这需要考虑 2n 个子问题。要构造出 S 的一切折半查找树，从中选出最佳的树，计算量无疑太大。因为由 n 个元素可能构造出的折半查找树的个数 N(n)满足

$$N(n) = \begin{cases} 1 & n \leq 1 \\ \sum_{i=1}^{n} N(i-1)N(n-i) & n \geq 2 \end{cases}$$

可以证明这个递归方程的解是：

$$N(n) = \frac{(2n)!}{(n!)^2 (n+1)}$$

这个函数增长极快。当 n 稍大时，要构造出一切折半查找树是不可能的。这就迫使我们用动态规划来求解，以便应用最佳原理，淘汰那些在计算过程中出现的不会是最佳的树。

3.2 最佳折半查找树的动态规划算法

设 T_{ij} 是元素子集$\{a_{i+1}, a_{i+2}, \ldots, a_j\}$的最小耗费查找树，$0 \leqslant i < j \leqslant n$。$C_{ij}$ 是 T_{ij} 的耗费，r_{ij} 是 T_{ij} 的根。T_{ij} 的权 W_{ij} 定义为：

$$W_{ij} = q + (p_{i+1}+q_{i+1}) + \ldots + (p_j+q_j)$$

W_{ij} 的涵义是，执行指令序列 σ 时，要查找的元素落在树 T_{ij} 上的概率。

设 T_{ij} 是由根元素 a_k 和它的左右子树 $T_{i,\,k-1}$ 和 T_{kj} 组成。其中，左子树是由元素$\{a_{i+1}, a_{i+2}, \ldots, a_j\}$组成的最小耗费树，$T_{ij}$ 是由元素$\{a_{i+1}, a_{i+2}, \ldots, a_j\}$组成的最小耗费树。当 i=k-1 时，左子树就是一棵空树；当 k=j 时，右子树是一棵空树。T_{ii} 的权 $W_{ii}=q_i$，它的耗费 $C_{ii}=0$。

在子树 T_{ij} 中，左右子树的每个节点的深度是它们在 $T_{i,\,k-1}$ 和 T_{kj} 中的深度加 1。于是 T 的耗费 C_{ij} 可以表示成

$$C_{ij} = W_{i,\,k-1} + p_k + W_{kj} + C_{i,\,k-1} + C_{kj}$$
$$= W_{ij} + C_{i,\,k-1} + C_{kj}$$

这里所取的 k 值要使和 $C_{i,\,k-1}+C_{kj}$ 最小。因此，为了找出一棵最优树 T_{ij}，要对每个 $k(i < k \leqslant j)$，计算以 a_k 为根的左右子树的耗费，然后选出一棵有最小耗费的树。以下是计算最佳子树的根的动态

规划的算法：

构造最优折半查找树的算法是：

经证明，只要事先计算了 r_{ij} 的表，通过对过程 BUILDTREE 的调用构造一棵最佳树，总共只需调用 n 次，每次调用只需常数时间。所以对 BUILDTREE 的总调用时间是 O(n)。

这个算法还可以改进。只要把动态规划算法中对 m 的搜索范围限制在 $T_{i,\,j-1}$ 和 $T_{i+1,\,j}$ 的根 $r_{i,\,j-1}$ 和 $r_{i+1,\,j}$ 之间，仍可以保证找出一棵最佳树。修改后的算法可以在 $O(n^2)$的时间完成全部计算。

4 资源分配问题

现在研究一个将有限资源分配给若干个工程的问题。 设资源总数是 a，工程个数为 n，给每项工程分配的资源数目不同，获得的利润也不相同，给定各工程的资源利润表和资源总数，应当怎么分配资源，才能获得最大的利润呢？问题要求给出资源总数 a 的一个分划 $x_1, x_2, \ldots, x_n$，$0 \leqslant x_i \leqslant a$ 且 $x_1+x_2+\ldots+x_n \leqslant a$，使得利润

$$G(a)=G_1(x_1)+G_2(x_2)+\ldots+G_n(x_n)$$

最大。这里 $G_i(x_i)$是把资源数 x_i 分配给第 i 项工程能获得的利润，$1 \leqslant i \leqslant n$。$G_i(x)$是给定的已知函数。如果 $G_i(x)$是 x 的线性函数，则是一般线性规划问题；若 $G_i(x)$不是 x 的线性函数，则只能用动态规划或其他方法来求问题的最佳分配。资源总数可以是投资总额，工程项目或产品种类等，故资源分配问题是许多实际问题的抽象。

某工厂生产某种产品，每单位（万件）的成本为 1（千元），每次开工的固定成本为 3（千元），工厂每季度的最大生产能力为 10（万件）。经调查，市场对该产品的需求量第一、二、三、四季度分别为 3、5、6、4（万件）。如果工厂在前两季度将全年的需求都生产出来，则通过少付三、四季度的固定成本费来降低成本，但是对于第三、四季度才能上市的产品需付存储费，每季每万件的存储费为 0.5（千元）。并且规定年初和年末这种产品均无库存。试制订一个生产计划，安排每个季度的产量，使一年的总费用（生产成本和存储费）最少。

非常明显，这是一个多阶段问题，以季度为单位来划分阶段。状态定义为每阶段开始时的存储量 s_k，决策为每个阶段的产量 p_k，记每个阶段的需求量（已知）为 r_k，则状态转移方程为：

$$s_{k+1} = s_k + p_k - r_k, \quad s_k \geq 0, \ k = 1, 2, \cdots, n$$

设每个阶段开工固定成本费用为 a，生产单位数量产品的成本为 b，每阶段单位数量产品的存储费用为 c，阶段指标为阶段的生产成本费用和存储费用之和，即：

$$v_k(s_k, p_k) = cs_k + \begin{cases} a + bp_k & p_k > 0 \\ 0 & p_k = 0 \end{cases}$$

指标函数 v_{kn} 为 v_k 之和，最优值函数 $f_k(s_k)$为从第 k 阶段的状态 s_k 出发到过程终结的最小费用，满足

$$f_k(s_k) = \min_{u_k \in U_k}[v_k(s_k, p_k) + f_{k+1}(s_{k+1})], \qquad k = n, n-1, \cdots, 1$$

其中允许决策集合 p_k 由每阶段的最大生产能力决定，设过程终结时允许存储量为 s^0_{n+1}，则终端条件为：

$$f_{n+1}(s^0_{n+1}) = 0$$

将以上各式代入到标准动态规划的框架中，就可以求得最优解。

现在给出资源分配问题的算法如下，其中：m 是资源总数，n 是工程项目数，G(i, j)是对工程 i 投资 j 的利润。这里 $1 \leqslant i \leqslant n$，$0 \leqslant j \leqslant m$，X[i]是分配给工程 i 的投资数，M 是总利润。

```
Void Binomial(n,k)
{
For (i=0;i<n;i++)
    For (j=0;j<min(j,k);j++)
        If (j==0)||(j==k)
            C[i][j]=1;
```

```
        Else
            C[i][j]=C[i-1][j-1]+C[i-1][j];
Return(C[n][k]);
}
int search(char a, root *s)
{ //若找到返回 1，否则返回-1
  if(!strcmp(a,l(s))) return 1;
  else if(strcmp(a,l(s))<0)
        if(v->lchild) return(search(a, s->lchild));
        else   return –1;
  else   if(s->rchild) return(search(a,s->rchild));
          else   return –1;
}
for(i=0; i<=n; i++) {W[i][i]=q[i]; C[i][i]=0; }
for(h=1; h<=n; h++)                    //h 表示子树中结点的个数，不包括虚叶
  for(i=0;   i<=n-h;   i++)            //i 表示子树 Ti，j 中的 i
{
j=i+h;
  W[i][j]=W[i][j-1]+p[j]+q[j];        //按 aj 为根计算
        设 m 是使 Ci,k-1+Ck,j 最小的 k 值                    //i≤k≤j
        Ci,j=Ci,k-1+Ck,j+Wi,j;
        ri,j=am;
       }
void BUILDTREE(i，j);
  {   建立 Tij 的根 Vij,并用 rij 标记 Vij;
      令 m 是 rij 所对应的元素 am 的下标;
      if （i＜m-1）
            BUILDTREE (I,m-1);{构造 Vij 的最优左子树}
      if （m＜j）
             BUILDTREE (m,j); {构造 Vij 的最优右子树}
  }
 void schedule( )
 {   int I,j;
     for（i=0,i<=m, i++）
      {   F[1,i]=G(1,i);
          P(1,i)=i
       }
     M(1)=max{F[1,0],F[1,1],…,F[1,m]};
      Q(1)=k; //k 是使 M(1)取最大的 F[1，k]的第二下标值
       for(i=2;i<=n;i++)
        {      F[I,0]=G(I,0)+F[i-1,0];
               P(I,0)=0;
                for(j=1;j<=m;j++)
                 {   设 k 是使得 G(I,k)+F[i-1,j-k]取最大的 k 值(0≤k≤j);
                     F[I,j]=G(I,k)+F[i-1,j-k];
                     P(I,j)=k              //给第 i 项工程的资源数
```

```
        }
            M(i)=ax{F[I,0],F[I,1],…,F[I,m]};
            Q(i)=r              //r 是使得 M(i)取最大值的 F[i，r]的第二下标//
    }
      M=max{M(1),M(2),…,M(n)};
       设 s 是使 M 取最大的 M(s)的下标
         if(s!=n)
           for(i=s+1;i<= n;i++)
           X[i]=0;          //后段的 X(i)全为 0//
     X[s]=P(s,Q(s));
    j=m-X[s];
    for（i=s-1,i>=1;i--）
     {    X[i]=P(I,j);
          j=j-X[i]
     }
}
```

可以用归纳法证明上述算法的正确性。当 i=1 时，显然 M(1)是给第一项工程投资的最大利润。从而，F(1, k)是对第一项目投资为 k 的最佳方案。

设 F(I-1, 0), F(i-1, 1), …, F(i-1, m)是对前 i-1 个项目分别投资 0, 1, …, m 的最佳方案，按给定的利润表，第三个 for 语句算出的 F(i, j)是投资 j 给前 i 项工程的最佳方案。M(i)是对前 i 项工程投资的最大收益，而 F(i, r)是最佳方案。当 i=n 时，获得了将总额 m 投资给第一项，前二项，前三项，…，前 n 项工程的最佳方案。M 是最大利润。根据对前 i 项工程获得最大利润的投资数 q(i)←r 及对前 i 项工程投资 i 获得最大利润时，第 i 项工程与前 i-1 项工程的投资分配数 k 和 j--k 的记录，程序算法中的第 21 行以后正好找出对各项工程的投资数。

算法的时间总耗费是 $O(m^2 \cdot n)$。

动态规划是不是比穷举法要好呢？像这样一类资源分配问题，可以将穷举算法的时间耗费与 $O(nm^2)$做个比较。把 m 个资源分配给 n 个项目，允许有的项目分配 0 个资源，分配方案有(C_m^{m+n-1})种（等价于把 m 个相同的球放入 n 个不同的匣子中，既允许有的匣子空着，也允许一个匣中放多个球的问题）。当 m 和 n 较大时，$O(C_m^{m+n-1})$远远大于 $O(m^2 n)$。

资源分配问题也可以表示成多级图来处理（例如将 m 个资源分配给 n 项工程的问题）。设把 j 个资源(j, m）分配给第 i 项工程可获纯利 N(i, j)。问题可以用一个 n+1 级图来表示。以级 i 对应工程项目 i，$1 \leqslant i \leqslant n$。对 $2 \leqslant i \leqslant n$，每一级有 m+1 个顶点 V(i, j)，$0 \leqslant j \leqslant m$，第 1 级和第 n+1 级都只有一个顶点，它们分别是 V(1, 0)=s 和 V(n+1, m)=t。对于 $2 \leqslant i \leqslant n$，顶点 V(i, j)表示把资源总数 j 分配给项目 1, 2, …, i-1 的状态。在这个图中，所有的边都具有(V(i, j), (i+1, l))的形式。一条边(V(i, j), V(i+1, l))上有一个权 N(i, l-j)来表示把资源 l-j 分配给工程 i 所获的利润。此外图中还有形如(V(i, j), V(n+1, m))的一些边，每条边给定一个权为 max{N(n, p)}。一个最佳资源分配方案是图中从源点到终点一条有最大权的道路确定。只要将所有边的权反号，就变成求最小值问题。

5　多机系统的可靠性设计

这一节要讨论怎样使用动态规划求一个多重函数的最佳解。问题是要求设计一个由多机（或多部件）连接而成的一个大型系统，如果 r_i 是部件 D_i 的可靠性系数，那么整个系统的可靠性系数就是 $\prod r_i$，$1 \leqslant i \leqslant n$。即使系统中每一个部件的个别可靠性都很好（各 r_i 都很接近于 1），但整个系统的可靠性未必能满足要求。设 n=10，r=0.99，$1 < i < 10$，$\prod r_i = 0.904$。这样的可靠性在许多情况下是不符合要求的。为了提高系统的可靠性，如果各部件的可靠性不能提高，配置多重部件就成为必要手段。将同一部件的多模块并置在整个系统的某一级中，由控制线路选择决定，每一级的某一个部件被正确地联入系统。一旦某一级的部件 D 发生故障，控制线路将使同一级的备份部件替换故障部件，使整个系统仍照常工作。

如果第 i 级的部件 D_i 共有 m_i 个，这一级的所有部件同时发生故障的可能性是$(1-r_i)^{mi}$。从而第 i

级的可靠性就是 $1-(1-r_i)^{mi}$。这样一来，当 $r_i=0.99$，$m_i=2$ 时，每一级的可靠性将达到 0.999。实际上每一级的可靠性总要比 $1-(1-r_i)^{mi}$ 略小一些。因为控制线路本身并不是绝对可靠的，同一级部件的各备份之间的故障也不是完全互不相关。不妨设第 i 级的可靠性实际上由函数 $\Phi_i(m_i)$确定，$1\leqslant i\leqslant n$。那么，一个多级系统的可靠性就是 $\prod_{i=1}^{n}\phi_i(m_i)$。

现在是用多重部件来配置一个系统，使得系统的可靠性最大。所谓可靠性最大是有一定的约束条件的。现实也不允许只考虑系统的可靠性而无限地增加每一级的备份，而应把系统可靠性和系统造价综合起来考虑。设 c 是第 i 级的一个部件 D 的代价，而 C 是整个系统所允许的最大造价，希望解决以下的极大问题：找出 $m_1, m_2, \ldots, m_n$ 的一个适当序列，使得

$$\prod_{i=1}^{n}\phi_i(m_i)$$

达到最大，同时满足

$$\sum_{i=1}^{n}c_i m_i \leqslant C$$

式中，m_i 是大于等于 1 的整数，$1\leqslant i\leqslant n$。

不妨设一切 $c_i>0$，$1\leqslant i\leqslant n$。由于一切 $m_i\geqslant 1$，不妨设 $1\leqslant m\leqslant u_i$，这里

$$u_i=\left\lfloor (C+c_i-\sum_{i=1}^{n}c_j)/c_i \right\rfloor \qquad (6)$$

因为不允许任何一级没有部件。一组最佳解 $m_1, m_2, \ldots, m_n$ 是一系列判定的结果，每次确定一个 m_i。设 $f_i(x)$是 $\prod_{i=1}^{n}\phi_i(m_i)$ 的最大值，即是前 i 级造价不超过 x 的最佳方案。其中，所有 m_j 满足

$$\sum_{i\leq j\leq i}c_j m_j \leq x \qquad 1\leqslant m_j\leqslant u_j,\ 1\leqslant j\leqslant i$$

那么，$f_n(C)$就是问题的最佳解。最后，可以从{1, 2, …, u }中选择和确定 m_n 的正确值。剩下的只是对剩余资金 $C-c_n m_n$ 确定出最佳的值 m_{n-1} 等。根据最佳原理，有

$$f_n(C)=\max_{1\leq m_n\leq v_n}\{\phi_n(m_n)f_{n-1}(C-c_n m_n)\}$$

对一切 $i\geqslant 1$，通用的计算式是：

$$f_i(x)=\max_{1\leq m_i\leq u_i}\{\phi_i(m_i)f_{i-1}(x-c_i m_i)\}$$

显然，对一切 x，$f_0(x)=1$。按以上公式，可以像资源分配问题那样，通过多步抉择，求出一个最佳方案。设 S_{ij} 包括一切形如(f, x)的二元序偶，这里 $f=f_i(x)$，$j\leqslant m_i$。对于从序列 $m_1, m_2, \ldots, m_n$ 中确定的每一个不同的 x，最多有一个形如(f, x)的序偶包括在 S_{ij} 中。如果 $f_1\geqslant f_2$ 且 $x_1\leqslant x_2$，按最佳原理，(f_1, x_1)就优于(f_2, x_2)，(f_2, x_2)将被(f_1, x_1)从 S_2 中淘汰掉，按最佳原理从 $S_{i1}, S_{i2}, \ldots, S_{in}$ 中淘汰那些不可能达到最佳的局部解之后，把 S_{ij} 中所有可能达到最佳的局部解记入 S_i 中，然后再计算 S_{i+1}, $S_{i+j}, \ldots$，最后，将从 S_n 中找到最佳方案，并通过 $S_{n-1}, S_{n-2}, \ldots, S_1$ 依次确定 m_i 之值$(1\leqslant i\leqslant n)$。

【例 1】考虑一个三部件组成的三级系统。部件 D_1, D_2, D_3 的价值是 $c_1=3000$ 元，$c_2=1500$ 元，$c_3=2000$ 元，系统总造价不能超过 10500 元，各部件的可靠性系数分别是 $r_1=0.9$，$r_2=0.8$ 和 $r_3=0.5$。

按公式（6），有 $u_1=2$，$u_2=3$，$u_3=3$。开始：

$S_0=\{(1, 0)\}$

$S_{11}=\{(f_0\cdot r_1,\ 0+c_1)\}=\{(0.9, 3000)\}$

$S_{12}=\{(0.99, 6000)\}$

在 $S_{12}=\{(0.99,\ 6000)\}$式中，(0.99, 6000)表示取 $m_1=2$ 时，第一级造价为 6000 元，局部可靠性 $f_1(x)=\phi_1(m_1)=1-(1-r_1)^2=0.99$。可得：

$S_1=\{(0.9, 3000),\ (0.99, 6000)\}$

第二步，计算 S_2。

$S_{21}=\{(0.9\times 0.8, 3000+1500),\ (0.99\times 0.8, 6000+1500)\}$

$=\{(0.72, 4500),\ (0.792, 7500)\}$

$S_{22}=\{(0.864, 6000)\}$

按组合，S_{22} 中还应有一项，即取 $m_1=2$，$m_2=2$，得(0.9504, 9000)。但由此所剩余的资金只有 1500，

不够第 3 级取 $m_3=1$ 的需要，故未列入。

$S_{23}=\{(0.8928, 7500)\}$

把 S_{21}，S_{22}，S_{23} 组合起来，得到：

$S_2=\{(0.72, 4500)$，$(0.864, 6000)$，$(0.8928, 7500)\}$

这里，S_{21} 中的第二项(0.792, 7500)已被(0.864, 6000)或(0.8928, 7500)所淘汰。

第三步，计算 S_3。

$S_{31}=\{(0.36, 6500)$，$(0.432, 8000)$，$(0.4464, 9500)\}$

$S_{32}=\{(0.54, 8500)$，$(0.648, 10000)\}$

$S_{33}=\{(0.63, 10500)\}$

把 S_{31}，S_{32}，S_{33} 组合起来，得到：

$S_3=\{(0.36, 6500)$，$(0.432, 8000)$，$(0.54, 8500)$，$(0.648, 10000)\}$

这里，S_{33} 中的方案(0.63, 10500)被 S_{32} 中的第二项(0.648, 10000)所淘汰。

从 S_3 中，能得到的最佳设计方案是，系统可靠性系数 0.647，造价为 10000 元。通过 S_i 的计算过程，可以找出 $m_3=2$，$m_2=2$，$m_1=1$。

至此，可以对可靠性设计问题构造一个形式化的动态规划算法。算法中还可以修改某些限制条件。例如在计算 S_i 时，m_i 的上界值不仅不能超过 u_i，而且可从 S_{i-1} 的序偶(f，x)中得出剩余资金 C'，m_i 的值实际上不能超过 $\left\lfloor (c' - \sum_{i+1\le j\le n} c_j)/c_i \right\rfloor$，也可以对系统的可靠性系数提出一个下界。在计算 S_i 时，一旦发现某个序偶(f，x)中的可靠性系数 f 低于要求的下界值，就不予考虑而从 S_i 中删掉。

6 背包问题

用设计一个背包问题的动态规划算法来作为本节的开始：给定 n 个重量为 $w_1, \ldots, w_n$、价值为 $v_1, \ldots, v_n$ 的物品和一个承重量为 W 的背包，求这些物品中最有价值的一个子集，并且要能够装到背包中。在这里假设所有的重量和背包的承重量都是正整数：而物品的数量不必是整数。

为了设计一个动态规划算法，需要热传导出一个递推关系，用较小子实例的解的形式来表示背包问题的实例的解。考虑一个由前 i 个物品($1\leqslant i\leqslant n$)定义的实例，物品的重量分别为 $w_1, \ldots, w_i$，价值分别为 $v_1, \ldots, v_i$，背包的承重量为 $j(1\leqslant j\leqslant W)$。设 V[i][j]为该实例的最优解的物品总价值。可以把前 i 个物品中的子集和那些不包括第 i 个物品的子集。然后有下面的结论：

（1）根据定义，在不包括第 i 个物品的子集中，最优子集的价值是 V[i-1][j]。

（2）在包括第 i 个物品的子集中（因此，$j-w_i\geqslant 0$），最优子集是由该物品和前 i-1 个物品中能够放进承重量为 $j-w_i$ 的背包的最优子集组成，这种最优子集的总价值等于 $v_i+V[i-1][j-w_i]$。

因此，在前 i 个物品中最优解的总价值等于这两个价值中的较大值。当然，如果第 i 个物品不能放进背包，从前 i 个物品中选出的最优子集的总价值等于从前 i-1 个物品中选出的最优子集的总价值。

根据上面的描述可以很容易地这样定义初始条件：

当 $j\geqslant 0$ 时，V[0][j]=0；当 $i\geqslant 0$ 时，V[i][0]=0　　（7）

目标是求 V[n][W]，即 n 个给定物品中能够放进承重量为 W 的背包的子集的最大总价值，以及最优子集本身。

图 3 给出了涉及物品总价值的序列图。

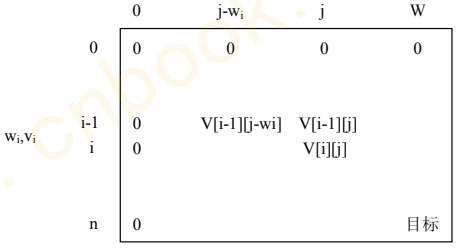


图 3　物品总价值

当 i, j>0 时，为了计算第 i 行第 j 列的单元格 V[i][j]，拿前一行同一列的单元格与 v_i 加上前一行左边 w_i 列的单元格的和作比较，计算出两者的较大值。根据上述的步骤，可以填此图的行和列中的数据。

考虑下面数据（如图 4 所示）给出的实例：

物品	重量	价值
1	2	12 美元
2	1	10 美元
3	3	20 美元
4	2	15 美元

承重量 W=5

图 4　实例图

图 5 给出了用公式 1、2 填写的动态规划表。

W1=2, v1=12
W2=1, v2=10
W3=3, v3=20
W4=2, v4=15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

图 5　动态规划表

因此，最大的总价值为 V[4][5]=37 美元。可以通过回溯的方法计算来求得最优子集的组成元素。因为 V[4][5]≠V[3][5]，物品 4 以及填满背包余下 5-2=3 个单位承重量的一个最优子集都包括在最优解中，而后者是由 V[3][3]来表示的，因为 V[3][3]=V[2][3]，物品 3 不是最优子集的一部分，因为 V[2][3]≠V[1][3]，物品 2 是最优选择的一部分，这个最优子集用元素 V[1][3-1]来指定余下的组成部分，同样道理，因为 V[1][2]≠V[0][2]，物品 1 是最优解{物品 1，物品 2，物品 4}的最后一个部分。

该算法的时间效率和空间效率都属于 O(nW)。用来求最优解的具体组成的时间效率属于 O(n+W)。

7　货郎担问题

货郎担问题（Traveling Salesman Problem）是有名的难题，起源于商业经济发达的欧美。原始问题是这样提出的：某推销员要到若干城市去推销商品，已知各城市之间的路程（或旅费）。他要选定一条从驻地出发，经过每个城市一遍最后回到驻地的路线，使总的路程（或旅费）最小。问题开始提出时不少人都认为很简单。后来人们从实践中才逐步认识到其计算复杂性是输入的指数函数，属于相当难解的问题之一。下面是这一问题的形式描述。

设 G=(V，E)是一个有向图。图中各条边的耗费 $C_{ij}>0$。当$(i, j)\notin E$ 时，定义 $C_{ij}=\infty$。设$\|V\|=n>1$。一条周游路线是包括 V 中的每个顶点在内的一条有向回路，一条周游路线的耗费是这条路线上所有边的耗费之和。所谓货郎担问题就是要在 G 中找出一条有最小耗费的周游路线。

不失普遍性，考虑以顶点 1 为始点和终点的一条周游路线。每条这样的路线均可表示为这种形式：对于某个 $k\in V-\{1\}$，路线包含了一条边$(1, k)$和从顶点 k 到 1 的一条道路，这条道路必须经过集 $V-\{1, k\}$的每个顶点各一次。不难看出，如果以顶点 1 为始点和终点的某条周游路线是最佳的，那么，这条路径上从顶点 k 到顶点 1 的部分路径（经过 $V--\{1, k\}$的每个顶点各一次），必是从 k 到 1 的一条最短路径，因此，最佳原理是适用的。设 g（i, S）是从顶点 i 出发，经过 S 中除去顶点 i 之外的其他顶点各一次并回到顶点 1 的一条最短路径的长，那么 $g(1, V-\{1\})$就是一条最佳旅游路线的长。按最佳原理，有

$$g(1, V-\{1\})=\min\{C1k+g(k, V--\{1, k\})\} \qquad (8)$$

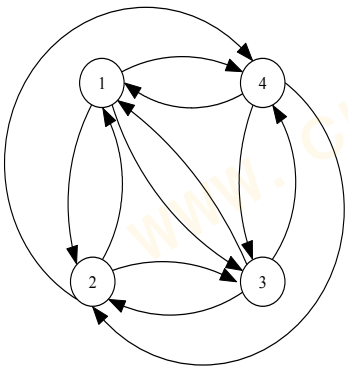
一般地，当始点和终点分别取 i，S 时，有

$$g(i, s)=\min\{c_{ij}+g(j, S--\{j\})\} \qquad (9)$$

如果对所有选定的 k，已知 $g(k, V--\{1, k\})$，从（8）可以求得 $g(1, V-\{1\})$。各 g(i, S)的值可以从

式（9）逐步求得。令 $g(i, \Phi)=c_{i1}$，$1 \leqslant i \leqslant n$，这是初始状态，之后，逐次对元素个数为 1 的集合 S，能求得一切 g(i, S)，然后求得 $\|S\|=2$ 的一切 g(i, S)。当 $\|S\|<n-1$ 时，对任何 g(i, S)，必须满足 $i \neq 1$，$1 \notin S$，$i \notin S$，最后可以求得问题的最佳解。

【例 2】对图 6（a）中的有向图求最佳解。图 6（b）是边的代价矩阵。



(a)

$$\begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$

(b)

图 6　有向图和矩阵图

初始，$g(2, \Phi)=c_{21}=5$；$g(3, \Phi)=c_{31}=6$；$g(4, \Phi)=c_{41}=8$。

先计算 $\|S\|=1$ 的情形。由公式（9），可得

$g(2, \{3\})=c_{23}+g(3, \Phi)=15$；$g(2, \{4\})=18$

$g(3, \{2\})=18$；$g(3, \{4\})=20$

$g(4, 2\})=13$；$g(4, \{3\})=15$

对 $\|S\|=2$ 和 $i \neq 1$，$1 \notin S$，$i \notin S$，有

$g(2, \{3, 4\})=\min\{c_{23}+g(3, \{4\})，c_{24}+g(4, \{3\})\}=25$

$g(3, \{2, 4\})=\min\{c_{32}+g(2, \{4\})，c_{34}+g(4, \{2\})\}=25$

$g(4, \{2, 4\})=\min\{c_{42}+g(2, \{3\})，c_{43}+g(3, \{2\})\}=23$

最后，由（8）式可得

$g(1, \{2, 3, 4\})=\min\{c_{12}+g(2, \{3, 4)，c_{13}+g(3, \{2, 4\})$

$c_{14}+g(4, \{2, 3\})\}=\min\{35, 40, 43\}=35$

图 6 中一条最短的周游路线的长为 35。这条路线可按在 g(i, S)的计算中，使得取值最小的那些 j 值而求出，记为 J(i，S)。则，

$J(1, \{2, 3, 4\}=2$；$J(2, \{3, 4\})=4$；$J(4, \{3\})=3$

因此，这条周游路线是 1, 2, 4, 3, 1。

设 N 是未计算 g(1, V--{1})前，需计算的 g(i, S)的个数。对每一个值 $\|S\|$，i 有 n-1 种取法。不包括 1 和 i 取大小为 k 的不同集合个数是 $\binom{n-2}{k}$。因此

$$N=\sum_{k=0}^{n-2}(n-1)\binom{n-2}{k}=(n-1)2^{n-2}$$

因为对每一个 $\|S\|=k$，求出 g(i, S)的值要 k-1 次比较，所以按公式（9）和（8）求出一条最短周游路线所花的时间是 $O(n^2 2^n)$。这比从(n-1)!条不同的周游路线中找出一条最佳路线的算法要好得多。但遗憾的是这一动态规划算法需要 $O(n2^n)$的空间。当 n 较大时，空间难以满足。

货郎担问题有许多具体应用。例如邮递路问题；在一条流水作业生产线上，一只机械手来拧紧某些螺母而耗时最少的问题（这里假定机械手拧紧各螺母的时间与路线无关）。这都是很直观的货郎担问题。前者要求设计一条有最短行程的邮车周游回路，后者要求设计一条机械手移动的最短周游路线。此外，货郎担问题的第三个实例是，由同样的机器设备生产多种不同产品，生产是周期性地进行的，在每一个生产周期里，要轮流生产 n 种不同的产品，当设备由生产产品 i 变为生产产品 j 时，需要额外增加转产开支 c_{ij}。我们希望制定一个生产计划，使得所付的额外开支最小，这虽没有前两例那么直观，但仍然可以按货郎担问题处理。以 n 个顶点表示 n 种不同产品，顶点 i 和 j 之间

的耗费 c_{ij} 表示从生产产品 i 变到生产产品 j 所需的额外开支。一般说来，$c_{ij} \neq c_{ji}$，$c_{ij} >= 0$。求最佳生产计划问题与货郎担问题等价。