第 3 章 动态规划

动态规划(Dynamic Programming)技术已经广泛应用于许多组合优化问题的算法设计中. 例如,图的多起点与多终点的最短路径问题、矩阵链的乘法问题、最大效益投资问题、背包问题、最长公共子序列问题、图像压缩问题、最大子段和问题、最优二分检索树问题、RNA的最优二级结构问题等.

一般的组合优化问题都有对应的目标函数和约束条件. 例如,第 1 章提到的货郎问题. 设 $C=\{c_1,c_2,\cdots,c_m\}$ 是城市集合,距离 $d(c_i,c_j)=d(c_j,c_i)\in\mathbf{Z}^+$,$1\leqslant i<j\leqslant m$. 求 1,2,…,$m$ 的排列 $k_1,k_2,\cdots,k_m$ 使得旅行路线最短,即

$$\min\left\{\sum_{i=1}^{m-1}d(c_{k_i},c_{k_{i+1}})+d(c_{k_m},c_{k_1})\right\}$$

其中,$\sum_{i=1}^{m-1}d(c_{k_i},c_{k_{i+1}})+d(c_{k_m},c_{k_1})$ 称为目标函数,约束条件是 $k_1,k_2,\cdots,k_m$ 必须构成 1,2,…,m 的排列. 组合优化问题的解分布在搜索空间中,其中满足约束条件的解称为可行解,而在可行解中使得目标函数达到最小(或最大)值的解称为最优解. 例如,货郎问题需要找到最短的旅行路线,是极小化问题;而背包问题需要找到一组能够放入背包并且使背包价值达到最大的物品,是极大化问题. 所谓求解组合优化问题就是找到该问题的最优解.

实践中的组合优化问题的搜索空间往往比较大,由于中间有许多重复计算,很多都是指数量级的. 蛮力算法对于规模较大的问题实际上是不可能运行的. 动态规划技术把求解过程变成一个多步判断的过程,每一步都对应于某个子问题. 算法细心地划分子问题的边界,从小的子问题开始,逐层向上求解. 通过子问题之间的依赖关系,有效利用前面已经得到的结果,最大限度减少重复工作,以提高算法效率. 对于许多蛮力算法是指数时间的问题,动态规划技术都取得了突破进展,将时间复杂度降到 $O(n^2)$ 或 $O(n^3)$ 等多项式级别. 但是,任何一种算法设计技术都有一定的适用范围,动态规划技术需要较大的存储空间来存储子问题计算的中间结果,以备后面使用. 对于输入规模很大的情况,这个代价可能比时间的消耗更难于承受.

3.1 动态规划的设计思想

为了理解动态规划的基本思想,先看一个简单的例子.

3.1.1 多起点、多终点的最短路径问题

例 3.1 在实际中经常会遇到路径选择问题. 如图 3.1 所示,有 5 个起点 $S_1,S_2,\cdots,S_5$,5 个终点 $T_1,T_2,\cdots,T_5$,其余结点是途经结点. 结点之间用边连接,边上的整数表示长度. 从起点 S_i 可以通过 4 条从左到右的边 $S_i\to A_j\to B_k\to C_l\to T_m$ 而到达终点 T_m,这就是一条从起点 S_i 到终点 T_m 的路径,路径的长度就是这 4 条边的长度之和. 我们的问题是:给定道路图,在所有起点到终点的路径中找一条长度最短的路径.

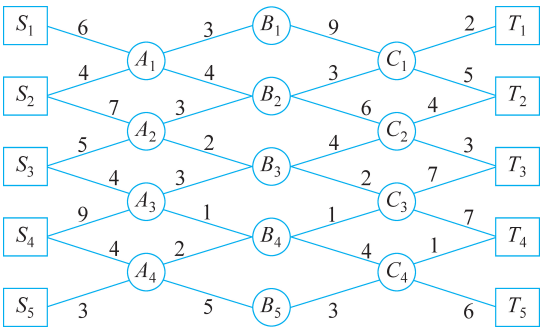

图 3.1 道路图

求解这个问题的蛮力算法就是穷举每一个起点到每一个终点的所有可能的路径,然后计算每条路径的长度,从中找出最短路径. 每条路径由 4 条边构成,除了最上边和最下边的某些结点外,处于中间的结点都有 2 条边可选,于是,从起点出发的路径大致达到 $O(m2^n)$ 量级,其中 m 为起点个数,n 为每条路径的长度,也就是路网的层数.

下面用动态规划方法求解这个问题. 从终点向起点回推,把求解过程分成 4 步,每一步对应的子问题的终点不变,但起点逐步前移,使得前步已经求解的问题恰好是后面新问题的子问题,到最后一步求解的最大的子问题正好是原始问题. 具体来说,所有子问题的终点都是 $T_m(m=1,2,3,4,5)$,但起点不同. 第一步对应子问题的起点是 $C_l(l=1,2,3,4)$,第二步对应子问题的起点是 $B_k(k=1,2,3,4,5)$,第三步对应子问题的起点是 $A_j(j=1,2,3,4)$,第四步对应子问题的起点是 $S_i(i=1,2,3,4,5)$. 这实际上就是原始问题. 在每一步需要求解的是:从当前起点到终点的最短路径及其长度.

第一步要确定从任何 C_l 到终点的最短路径. 先看 C_1,到终点只有两条路,向上走到 T_1 或向下走到 T_2,令 $F(C_1)$表示从 C_1 到终点的最短路径长度,于是

$$F(C_1)=\min\{C_1T_1,C_1T_2\}=\min\{2,5\}=2$$

把这个结果标记在 C_1 的上方,记作“u,2”,其中 u 表示到终点的最短路应该“向上”,2 表示这条最短路的长度. 接着考虑 C_2. 与在 C_1 所做的判断类似,可以在 C_2 标记为“d,3”,其中“d”表示到终点的最短路应该“向下”. 同样地可以把 C_3 和 C_4 依次标记为“u,7”(或“d,7”,因为向下与向上的路径一样长)和“u,1”. 至此第一步的判断全部完成. 在这一步判断中使用的只是从 C_l 到 T_m 所有可能的边长. 如果以 $F(C_l)$表示从 C_l 到终点的最短路径的长度,那么判断公式是:

$$F(C_l)=\min_m\{C_lT_m\}$$

第二步要确定从任何 B_k 到终点的最短路径. 先看 B_1,从 B_1 只能向下到 C_1,接着从

C_1 走最短路到终点. 于是

$$F(B_1)=B_1C_1+F(C_1)=9+2=11$$

把这个结果标记在 B_1 的上方,记作"d,11". 接着考虑 B_2. 与在 B_1 所做的不同,从 B_2 既可以到 C_1,后接从 C_1 到终点的最短路;也可以先到 C_2,后接从 C_2 到终点的最短路. 这两条中哪条更短,就应该选哪条. 于是,从 B_2 出发到终点的最短路长度应该是:

$$F(B_2)=\min\{B_2C_1+F(C_1),B_2C_2+F(C_2)\}=\min\{3+2,6+3\}=5$$

这是选择先向上走的结果,因此在 B_2 标记为"u,5". 同样地可以把 B_3,B_4 和 B_5 依次标记为"u,7""d,5""u,4". 至此第二步的判断全部完成. 在这一步判断中使用的信息是:从 B_k 到 C_l 所有可能的边长以及上一步所做的标记. 递推的判断公式是:

$$F(B_k)=\min_l\{B_kC_l+F(C_l)\}$$

类似地可以完成后面两步的判断过程,相关的递推公式给在下面:

$$F(A_j)=\min_k\{A_jB_k+F(B_k)\}$$

$$F(S_i)=\min_j\{S_iA_j+F(A_j)\}$$

各步判断所做的标记给在图 3.2 中. 到全部判断完成以后,从起点到终点的最短路径已经找到了. 观察在 $S_1,S_2,\cdots,S_5$ 的标记,最短路的长度是 10,一条从 S_3 开始,追寻标记 d 向下到 A_3,接着按 A_3 处的标记 d 向下到 B_4,再按 B_4 处的标记 d 向下到 C_4,最后按 C_4 处的标记 u 向上到 T_4. 还有另一条是:S_5,A_4,B_4,C_4,T_4. 两条最短路径都用粗线在图 3.2 中标出.

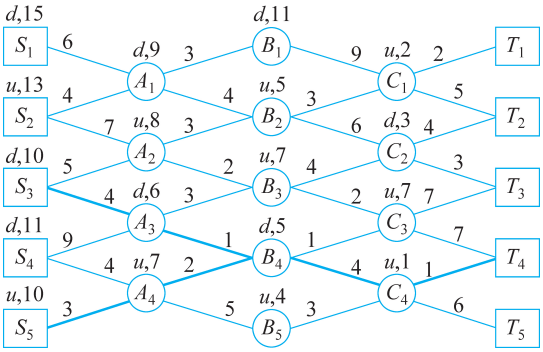

图 3.2 结点的标记及最短路

与蛮力算法相比,这种算法的好处是:在判断时只考虑由前面子问题的最优解(是当前子问题最优解的组成部分)可能的延伸结果,从而把许多不可能成为最优解的部分路径尽早从搜索中删除,因此能够提高效率. 根据上面的递推公式,除终点外,对每个结点只需要做 2 次加法(对 C_l 层结点不做加法)和 1 次比较,因此算法的时间复杂度可以降到 $O(mn)$,其中 m 代表每层的结点个数,n 是层数.

3.1.2 使用动态规划技术的必要条件

先把上面的最短路径问题的动态规划算法简单总结一下,其主要特征是:求解的问题是多阶段决策(优化)问题;求解过程是多步判断,从小到大依次求解每个子问题,最后求解的子问题就是原始问题. 子问题目标函数的最小值之间存在依赖关系,将子问题的解记录

下来，以备后面求解时使用. 其中关于子问题目标函数的最小值之间的依赖关系是最关键的. 这一条称为**优化原则**或**最优子结构性质**. 更清晰的表述如下.

优化原则：一个最优决策序列的任何子序列本身一定是相对于子序列的初始和结束状态的最优的决策序列.

正是由于优化原则，在考虑后面较大子问题的最优解时，只需考虑它的子问题的最优解所延伸的结果. 需要注意的是：在实践中并不是所有的组合优化问题都适用于优化原则，有时对某个子问题的解不一定达到最优，但是当把它延伸成整个问题的解时反而成了最优解. 如果对这样的问题使用动态规划技术，就可能出错. 下面是一个简单的例子.

例 3.2 求总长模 10 的最短路径.

如图 3.3 所示，S 是起点，T 是终点，边上的数字代表道路的长度. 与例 3.1 类似，我们需要找出从 S 到 T 的模 10 意义下的最短路径. 这里的模 10 指的是除以 10 得到的余数.

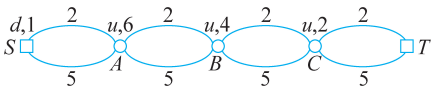

图 3.3 一个不满足优化原则的反例

采用和例 3.1 一样的动态规划算法. 首先计算从 C 到 T 的模 10 最短路径：

$$F(C)=\min\{2 \bmod 10,5 \bmod 10\}=\min\{2,5\}=2$$

在 C 的上方标注“u，2”. 第二步计算从 B 到 T 的模 10 最短路径：

$$F(B)=\min\{(2+F(C)) \bmod 10,(5+F(C)) \bmod 10\}=\min\{4,7\}=4$$

在 B 的上方标注“u，4”. 第三步计算从 A 到 T 的模 10 最短路径：

$$F(A)=\min\{(2+F(B)) \bmod 10,(5+F(B)) \bmod 10\}=\min\{6,9\}=6$$

在 A 的上方标注“u，6”. 最后计算从 S 到 T 的模 10 最短路径：

$$F(S)=\min\{(2+F(A)) \bmod 10,(5+F(A)) \bmod 10\}=\min\{8,1\}=1$$

于是得到 S 上方的标记“d，1”. 从而找到该实例的一个解，即沿“下、上、上、上”的方向从 S 走到 T，路径模 10 的长度为 1. 简单地观察就可以发现，这不是最优解. 如果选择方向为“下、下、下、下”的路径，模 10 以后的路径长度为 $(5+5+5+5) \bmod 10=0$.

动态规划算法不适于这样的问题，原因在于这个问题不满足优化原则. 方向为“下、下、下、下”的路径是一条从 S 到 T 的最优路径，但是它的某些子路径，比如从 C 向下到 T 的路径并不是从 C 到 T 的最优路径.

这说明在使用动态规划设计技术之前，首先要搞清楚所求解的问题是不是满足优化原则，这是使用动态规划技术的必要条件.

3.2 动态规划算法的设计要素

这里用一个矩阵链的乘法问题为例来说明动态规划算法的设计要素.

例 3.3 设 $\boldsymbol{A}_1,\boldsymbol{A}_2,\cdots,\boldsymbol{A}_n$ 为 n 个矩阵的序列，其中 $\boldsymbol{A}_i$ 为 $P_{i-1}\times P_i$ 阶矩阵，$i=1,2,\cdots,n$. 这个矩阵链的输入用向量 $\boldsymbol{P}=<P_0,P_1,\cdots,P_n>$ 给出，其中 P_0 是矩阵 $\boldsymbol{A}_1$ 的行数，$P_i(i=1,2,\cdots,n-1)$ 既是矩阵 $\boldsymbol{A}_i$ 的列数也是矩阵 $\boldsymbol{A}_{i+1}$ 的行数，最后的 P_n 是矩阵 $\boldsymbol{A}_n$ 的列数. 计算这个矩阵链需要做 $n-1$ 次两个矩阵的相乘运算，可以用 $n-1$ 对括号表示运算的顺序. 因为矩阵乘法满足结合律，无论采用哪种顺序，最后的结果都是一样的. 但是，采用不同的顺序计算的工作量却不一样. 怎样定义两个矩阵相乘的工作量呢？假设矩阵 $\boldsymbol{A}_1$ 与

$\boldsymbol{A}_2$ 相乘,其中 $\boldsymbol{A}_1$ 是 i 行 j 列的矩阵,$\boldsymbol{A}_2$ 是 j 行 k 列的矩阵,那么它们的乘积 $\boldsymbol{A}_1\boldsymbol{A}_2$ 是 i 行 k 列矩阵,含有 ik 个元素. 以元素相乘作为基本运算,乘积中每个元素的计算都需要做 j 次乘法,于是计算 $\boldsymbol{A}_1\boldsymbol{A}_2$ 总共需要 ijk 次乘法. 请看下面的具体实例:

假设输入的是 $\boldsymbol{P}=<10,100,5,50>$,这说明有 3 个矩阵相乘,其中

$$\boldsymbol{A}_1:10\times 100,\quad \boldsymbol{A}_2:100\times 5,\quad \boldsymbol{A}_3:5\times 50$$

有两种乘法次序,即 $(\boldsymbol{A}_1\boldsymbol{A}_2)\boldsymbol{A}_3$ 和 $\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3)$. 如果采用第一种次序,执行的基本运算次数是:

$$10\times 100\times 5+10\times 5\times 50=7500$$

而采用第二种次序,执行的基本运算次数是:

$$10\times 100\times 50+100\times 5\times 50=75\,000$$

工作量相差达到 10 倍之多.

我们的问题是:给定向量 $\boldsymbol{P}$,确定一种乘法次序,使得基本运算的总次数达到最少.

一般的蛮力算法就是枚举所有可能的乘法次序,针对每种次序计算基本运算的次数,从中找出具有最小运算次数的乘法次序. 每一种乘法次序对应了一种在 n 个项中加 $n-1$ 对括号的方法. 根据组合数学的知识不难知道,加 n 对括号的方法数是一个 Catalan 数,它的值是 $\dfrac{1}{n+1}\binom{2n}{n}$. 根据 Stirling 公式,即使对每种次序的计算工作量为常数,对规模为 $n+1$ 的输入使用蛮力算法的时间将是:

$$W(n)=\Omega\left(\frac{1}{n+1}\frac{(2n)!}{n!\ n!}\right)=\Omega\left(\frac{1}{n+1}\frac{\sqrt{2\pi 2n}\left(\frac{2n}{\mathrm{e}}\right)^{2n}}{\sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^{n}\sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^{n}}\right)$$

$$=\Omega\left(\frac{1}{n+1}\frac{n^{\frac{1}{2}}2^{2n}n^{2n}\mathrm{e}^{n}\mathrm{e}^{n}}{\mathrm{e}^{2n}n^{\frac{1}{2}}n^{n}n^{\frac{1}{2}}n^{n}}\right)=\Omega(2^{2n}/n^{\frac{3}{2}})$$

这是一个指数时间的算法. 下面尝试动态规划的算法. 我们将从子问题的划分、递推方程的确定、递归和迭代的实现方法、复杂度分析等方面介绍动态规划算法的设计特征.

3.2.1 子问题的划分和递推方程

我们的优化目标是基本运算次数的最小化. 如何界定子问题的边界? 令 $\boldsymbol{A}_{1..n}$ 表示输入的矩阵链. 如果从前向后划分,所产生的子问题只有后边界,是 $\boldsymbol{A}_{1..i}$ 的形式,$i=1,2,\cdots,n$. 但是在计算子问题 $\boldsymbol{A}_{1..j}$,$j>i$ 时,我们不仅需要子问题 $\boldsymbol{A}_{1..i}$ 的信息,也需要子问题 $\boldsymbol{A}_{(i+1)..j}$ 的信息. 这说明子问题的划分需要前后两个边界. 用 $\boldsymbol{A}_{i..j}$ 定义矩阵链 $\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_j$ 相乘的子问题,$m[i,j]$表示得到乘积 $\boldsymbol{A}_{i..j}$ 所用的最少基本运算次数. 假定其最后一次相乘发生在矩阵链 $\boldsymbol{A}_{i..k}$ 和 $\boldsymbol{A}_{k+1..j}$ 之间,即

$$\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_j=(\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_k)(\boldsymbol{A}_{k+1}\boldsymbol{A}_{k+2}\cdots\boldsymbol{A}_j)\quad k=i,i+1,\cdots,j-1$$

那么子问题 $\boldsymbol{A}_{i..j}$ 的计算依赖于子问题 $\boldsymbol{A}_{i..k}$ 和 $\boldsymbol{A}_{k+1..j}$ 的计算结果. 换句话说,$m[i,j]$的值依赖于 $m[i,k]$和 $m[k+1,j]$的值,具体的依赖关系可以表示为

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\leqslant k<j}\{m[i,k]+m[k+1,j]+P_{i-1}P_kP_j\} & i<j\end{cases}$$

上式中的 k 代表子问题的划分位置，应该考虑所有可能的划分，即 $i \leqslant k < j$，从中比较出最小的值；$P_{i-1}P_kP_j$ 是最后把两个子矩阵链 $\boldsymbol{A}_{i..k}$ 和 $\boldsymbol{A}_{k+1..j}$ 的结果矩阵相乘所需要的基本运算次数. 当 $i=j$ 时，矩阵链只有一个矩阵 $\boldsymbol{A}_i$，这时乘法次数是 0，对应了递推式的初值.

不难看到，这个问题是满足优化原则的. 这意味着当 $m[i,j]$ 达到最小值时，子问题的优化函数值 $m[i,k]$ 和 $m[k+1,j]$ 也是最小的. 因为如果对于子问题存在更好的优化函数值，比如存在更小的 $m'[i,k]$，那么用这个更小的值替换原来的 $m[i,k]$，就可以得到更小的 $m'[i,j]$，这将与 $m[i,j]$ 的最优性矛盾.

3.2.2 动态规划算法的递归实现

下面考虑算法的实现. 为了确定每一次相乘时加括号的位置，需要设计表 $s[i,j]$ 来记录 $m[i,j]$ 达到最小值时 k 的划分位置. 根据上面的递推公式，可以有两种实现方法：递归实现和迭代实现.

先考虑递归实现方法.

算法 3.1 RecurMatrixChain($\boldsymbol{P}$,i,j)

输入：矩阵链 $\boldsymbol{A}_{i..j}$ 的输入为向量 $\boldsymbol{P}=<P_{i-1},P_i,\cdots,P_j>$，其中 $1 \leqslant i \leqslant j \leqslant n$

输出：计算 $\boldsymbol{A}_{i..j}$ 的所需最小乘法运算次数 $m[i,j]$ 和最后一次运算的位置 $s[i,j]$

```
1. if i=j
2. then m[i,j]←0; s[i,j]←i; return m[i,j]
3. m[i,j]←∞
4. s[i,j]←i
5. for k←i to j−1 do            //考虑所有可能的划分位置
6.     q←RecurMatrixChain(P,i,k)+RecurMatrixChain(P,k+1,j)+P_{i-1}P_kP_j
7.     if q<m[i,j]
8.     then m[i,j]←q            //用找到的更好优化函数值替换原值,并记录划分位置
9.          s[i,j]←k
10. return m[i,j]
```

n 个矩阵相乘，只需令 $i=1,j=n$，调用算法 3.1 即可. 下面分析这个递归算法的效率. 考虑输入规模为 n 的矩阵链相乘，即 $i=1,j=n$ 的情况. 算法在第 5 行执行 for 循环，k 从 1 到 $n-1$. 每次进入循环体都在第 6 行进行两个子问题的递归求解，其中一个规模为 k，另一个为 $n-k$. 其余工作都是常数时间. 因此该算法的时间复杂度函数满足递推关系：

$$T(n) \geqslant \begin{cases} 0 & n=1 \\ \sum\limits_{k=1}^{n-1}(T(k)+T(n-k)+O(1)) & n>1 \end{cases}$$

经过化简得

$$T(n) \geqslant \Theta(n)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k)=\Theta(n)+2\sum_{k=1}^{n-1}T(k)$$

定理 3.1 当 $n>1$ 时，$T(n)=\Omega(2^{n-1})$.

证 $n=2$，$T(2) \geqslant c=c_1 2^{2-1}$，$c_1=c/2$ 为某个正数.

假设对于任何小于 n 大于或等于 2 的 k，$T(k) \geqslant c_1 2^{k-1}$，则存在某个常数 c'，使得

$$T(n) \geqslant c'n + 2\sum_{k=1}^{n-1} T(k) \geqslant c'n + 2\sum_{k=2}^{n-1} c_1 2^{k-1}$$

$$= c'n + c_1(2^n - 4) \geqslant c_1 2^{n-1}$$

可以看到,通过使用动态规划的设计思想,该算法的时间复杂度比蛮力算法有所改进,但是并没有得到多项式时间的高效算法. 时间复杂度高的原因在于:在递归调用中同一个子问题被多次重复计算. 以矩阵链 $\mathbf{A}_{1..5}$ 的计算为例. 图 3.4 用一棵横放的树给出了递归计算的过程. 每个问题对应一个结点,树根是原始问题,记作"1..5". 第 1 层的结点对应于原问题在 $k=1,2,3,4$ 的不同位置进行划分所得到的 8 个子问题,每个子问题表示成原问题的儿子. 第 2 层对应这 8 个子问题进一步递归求解得到的新的子问题,有24 个. 类似地,第 3 层有 32 个新的子问题,第4 层有 16 个新的子问题. 在整个递归计算中总计产生了

$$1 + 8 + 24 + 32 + 16 = 81$$

个子问题. 但是,根据边界考查不同的子问题个数:规模为 1 的子问题有 5 个,规模为 2 的有 4 个,规模为 3 的有 3 个,规模为 4 的有 2 个,规模为 5 的有 1 个,总计

$$5 + 4 + 3 + 2 + 1 = 15$$

个不同的子问题. 这说明算法计算的 81 个子问题中有许多是重复的,这就是递归实现动态规划算法时间复杂度高的原因.

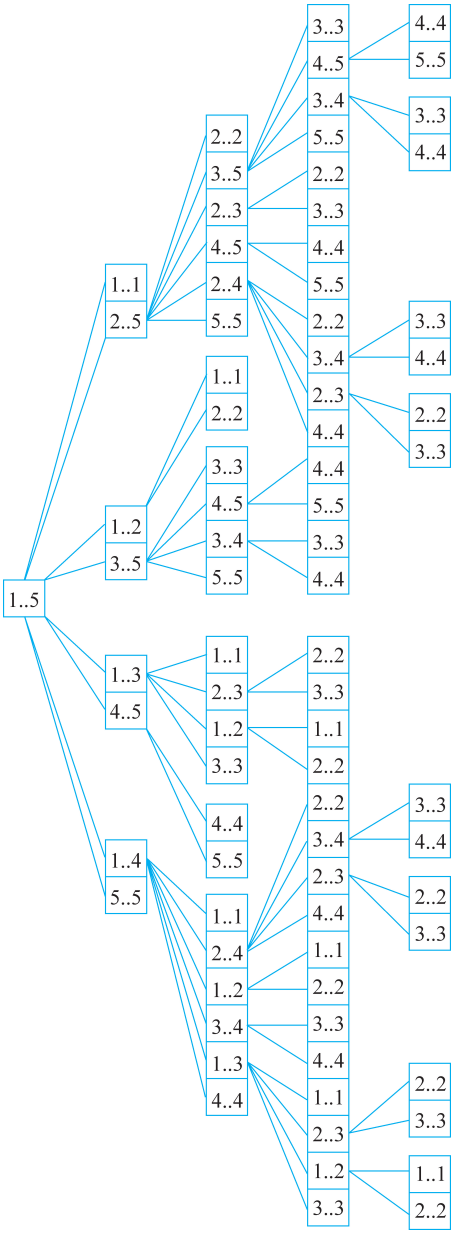

图 3.4　子问题

3.2.3　动态规划算法的迭代实现

如果在计算中对每个子问题只计算一次,并且把结果保存起来. 等到后面计算其他子问题需要这个值时,直接把它代入,这样就能够改善算法的时间复杂度. 这就是迭代实现动态规划算法的思想. 为了实现这个想法,需要解决以下两个问题:

(1) 开辟一个存储空间,用表格的方式来存储子问题的优化函数值和划分边界,通常把这些表格称为"备忘录". 这说明提高效率需要付出空间的代价. 对于规模大的实例,过大的空间需求往往成为不能使用动态规划算法的主要因素.

(2) 子问题的计算需要从底向上进行. 每个子问题只计算一次. 这就是说,从最小规模的子问题开始,比如规模为 1 的所有子问题,然后是规模为 2 的所有子问题,接着是规模为

3 的所有子问题……直到规模为 n 的子问题(原始问题)为止. 这样才能在前面的计算中准备好后面计算要查找的数据.

下面的伪码给出了矩阵链问题动态规划算法的迭代实现方法.

算法 3.2 MatrixChain($\boldsymbol{P}$,n)
输入:矩阵链 $\boldsymbol{A}_{1..n}$ 的输入为向量 $\boldsymbol{P}=<P_0,P_1,\cdots,P_n>$
输出:计算 $\boldsymbol{A}_{i..j}$ 的所需最小乘法运算次数 $m[i,j]$和最后一次运算的位置 $s[i,j]$,$1\leqslant i\leqslant j\leqslant n$

```
1. 令所有的 m[i,i]初值为 0,s[i,j]初值为 i,1≤i≤j≤n
2. for r←2 to n do                          //r 为当前计算的链长(子问题规模)
3.     for i←1 to n−r+1 do                  //n−r+1 为最后一个 r 链的前边界
4.         j←i+r−1                          //计算前边界为 i,长为 r 链的后边界 j
5.         m[i,j]←m[i+1,j]+P_{i−1} * P_i * P_j   //划分为 A_i(A_{i+1}…A_j),* 为普通乘法
6.         s[i,j]←i                         //记录分割位置
7.         for k←i+1 to j−1 do
8.             t←m[i,k]+m[k+1,j]+ P_{i−1} * P_k * P_j   //划分位置是(A_i…A_k)(A_{k+1}…A_j)
9.             if t<m[i,j]                  //用更好的值替换
10.            then m[i,j]←t
11.                 s[i,j]←k
```

下面分析这个算法的时间复杂度. 在算法的第 2 行、第 3 行和第 7 行的规模都不超过 $O(n)$,总计嵌套循环执行 $O(n^3)$次,循环体内的计算为 2 次加法,2 次乘法,是常数时间,因此,算法的时间复杂度是 $W(n)=O(n^3)$,比起递归实现的指数时间确实有了明显的改进.

3.2.4 一个简单实例的计算过程

下面给出一个计算实例,进一步说明备忘录的结构.

设输入是 $\boldsymbol{P}=<30,35,15,5,10,20>$,$n=5$,相应的矩阵链是:$\boldsymbol{A}_1,\boldsymbol{A}_2,\boldsymbol{A}_3,\boldsymbol{A}_4,\boldsymbol{A}_5$,其中,

$$\boldsymbol{A}_1:30\times35,\quad \boldsymbol{A}_2:35\times15,\quad \boldsymbol{A}_3:15\times5,\quad \boldsymbol{A}_4:5\times10,\quad \boldsymbol{A}_5:10\times20$$

计算开始,当子问题规模 $r=1$ 时,所有的 $m[1,1],m[2,2],\cdots,m[5,5]$都等于 0. 然后 $r=2$,有 4 个子问题,计算结果是:

$$m[1,2]=30\times35\times15=15\,750$$
$$m[2,3]=35\times15\times5=2625$$
$$m[3,4]=15\times5\times10=750$$
$$m[4,5]=5\times10\times20=1000$$

接着,$r=3$,有 3 个子问题,计算结果是:

$$m[1,3]=\min\{m[1,2]+30\times15\times5,m[2,3]+30\times35\times5\}=7875,\quad s[1,3]=1$$
$$m[2,4]=\min\{m[2,3]+35\times5\times10,m[3,4]+35\times15\times10\}=4375,\quad s[2,4]=3$$
$$m[3,5]=\min\{m[3,4]+15\times10\times20,m[4,5]+15\times5\times20\}=2500,\quad s[3,5]=3$$

接着,$r=4$,有 2 个子问题,计算结果是:

$$m[1,4]=\min\{m[2,4]+30\times35\times10,m[1,2]+m[3,4]+30\times15\times10,$$
$$m[1,3]+30\times5\times10\}=9375$$
$$m[2,5]=\min\{m[3,5]+35\times15\times20,m[2,3]+m[4,5]+35\times5\times20,$$

$$m[2,4]+35\times10\times20\}=7125$$

对应的划分位置分别是 $s[1,4]=3$ 和 $s[2,5]=3$. 最后,$r=5$,就是原始问题,计算结果是:

$$m[1,5]=\min\{m[2,5]+30\times35\times20,m[1,2]+m[3,5]+30\times15\times20,$$
$$m[1,3]+m[4,5]+30\times5\times20,m[1,4]+30\times10\times20\}=11\,875$$
$$s[1,5]=3$$

存储上述计算结果的备忘录和标记函数分别如表 3.1 和表 3.2 所示.

表 3.1 优化函数值的备忘录

$r=1$	$m[1,1]=0$	$m[2,2]=0$	$m[3,3]=0$	$m[4,4]=0$	$m[5,5]=0$
$r=2$	$m[1,2]=15\,750$	$m[2,3]=2625$	$m[3,4]=750$	$m[4,5]=1000$	
$r=3$	$m[1,3]=7875$	$m[2,4]=4375$	$m[3,5]=2500$		
$r=4$	$m[1,4]=9375$	$m[2,5]=7125$			
$r=5$	$m[1,5]=11\,875$				

表 3.2 标记函数

$r=2$	$s[1,2]=1$	$s[2,3]=2$	$s[3,4]=3$	$s[4,5]=4$	
$r=3$	$s[1,3]=1$	$s[2,4]=3$	$s[3,5]=3$		
$r=4$	$s[1,4]=3$	$s[2,5]=3$			
$r=5$	$s[1,5]=3$				

根据备忘录可以知道最少运算次数是 11 875,根据 $s[1,5]=3$ 知道最后一次划分在第三个矩阵的后面,于是得到 $\boldsymbol{A}_{1..5}=\boldsymbol{A}_{1..3}\boldsymbol{A}_{4..5}$. 接着查找 $s[1,3]=1$,于是得到 $\boldsymbol{A}_{1..3}=\boldsymbol{A}_{1..1}\boldsymbol{A}_{2..3}$,从而得到最佳的运算次序是:

$$\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4\boldsymbol{A}_5=(\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))(\boldsymbol{A}_4\boldsymbol{A}_5)$$

通过矩阵链相乘的例子,我们可以总结动态规划算法的设计要素:

(1) 划分子问题,用参数表达子问题的边界,将问题求解转变成多步判断的过程.

(2) 确定优化函数,以该函数的极大(或极小)值作为判断的依据,确定是否满足优化原则.

(3) 列出关于优化函数的递推方程(或不等式)和边界条件.

(4) 考虑是否需要设立标记函数,以记录划分位置.

(5) 自底向上计算,以备忘录方法(表格)存储中间结果.

(6) 根据备忘录(和标记函数)通过追溯给出最优解.

3.3 动态规划算法的典型应用

本节介绍动态规划算法的一些典型应用.

3.3.1 投资问题

例 3.4 设有 m 元,n 项投资,函数 $f_i(x)$ 表示将 x 元投入第 i 项项目所产生的效益,

$i=1,2,\cdots,n$. 问：如何分配这 m 元，使得投资的总效益最高？

假设钱数的分配都是非负整数，分配给第 i 个项目的钱数是 x_i，那么该问题可以描述为

$$\text{目标函数}\quad \max\{f_1(x_1)+f_2(x_2)+\cdots+f_n(x_n)\}$$

$$\text{约束条件}\quad x_1+x_2+\cdots+x_n=m,\quad x_i\in\mathbf{N}$$

一个简单的实例是：有 5 万元，4 个项目，效益函数（以万元为单位）如表 3.3 所示.

表 3.3　效益函数

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$	x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0	3	13	10	30	22
1	11	0	2	20	4	14	15	32	23
2	12	5	10	21	5	15	20	40	24

使用动态规划算法首先需要界定子问题的边界. 可以是按项目划分成 n 个阶段，比如先考虑对第 1 个项目的分配，接着考虑对前 2 个项目的分配，……，直到考虑对 n 个项目的分配. 每阶段的问题都构成后面阶段的子问题. 与前面的最短路径和矩阵链乘法的不同点在于：这里每个阶段的子问题还存在另一个参数：投资的钱数. 于是每个阶段的子问题还可以按照钱数进行更细的划分. 这里使用的是具有两个不同参数的优化函数. 设 $F_k(x)$ 表示 x 万元投给前 k 个项目的最大效益，其中 $k=1,2,\cdots,n$；$x=1,2,\cdots,m$. 在第 k 步，钱数为 x 万元时需要做的是：假设知道 p 万元（$p\leqslant x$）投给前 $k-1$ 个项目的最大效益，如何对前 k 个项目分配 x 元以取得最大的效益. 首先从 x 万元中分配 x_k（$0\leqslant x_k\leqslant x$）万元给第 k 个项目，那么剩下的 $x-x_k$ 万元将分配给前 $k-1$ 个项目，最佳分配方案已经在第 $k-1$ 步计算过. 于是得到如下递推方程和边界条件：

$$F_k(x)=\max_{0\leqslant x_k\leqslant x}\{f_k(x_k)+F_{k-1}(x-x_k)\},\quad k=2,3,\cdots,n$$

$$F_1(x)=f_1(x),\quad F_k(0)=0,\quad k=1,2,\cdots,n$$

这里需要设立标记函数 $x_k(x)$，它表示当 $F_k(x)$ 取得最大值时应该分配给第 k 个项目的钱数.

不难验证这个问题满足优化原则. 算法的伪码描述请读者自己给出. 下面以上述实例看看算法的运行过程.

首先，计算 $F_1(x)$. x 万元仅分配给第一个项目，这对应于递推关系的初值，可以直接从效益函数表中查到，于是有

$$F_1(1)=f_1(1)=11,\quad F_1(2)=f_1(2)=12,\quad F_1(3)=f_1(3)=13,$$

$$F_1(4)=f_1(4)=14,\quad F_1(5)=f_1(5)=15,\quad x_1(x)=x,\quad x=1,2,3,4,5$$

这些值存在备忘录的第一列，见表 3.4. 下面计算 $F_2(x)$. 首先计算 $F_2(1)$，总钱数 1 万元，分配给第二个项目可能有两种结果：0 万元或 1 万元. 于是递推关系是：

$$F_2(1)=\max\{f_2(0)+F_1(1),\quad f_2(1)+F_1(0)\}=\max\{0+11,0+0\}=11,$$

$$x_2(1)=0$$

其中，$x_2(1)=0$ 表示取得最大效益 11 万元时分配给第二个项目的钱数是 0. 类似地可以计算 $F_2(2),F_2(3),F_2(4),F_2(5)$：

$$F_2(2)=\max\{f_2(0)+F_1(2),f_2(1)+F_1(1),f_2(2)+F_1(0)\}=12,\quad x_2(2)=0$$

$$F_2(3)=\max\{f_2(0)+F_1(3),f_2(1)+F_1(2),f_2(2)+F_1(1),$$
$$f_2(3)+F_1(0)\}=16,\quad x_2(3)=2$$

$$F_2(4)=\max\{f_2(0)+F_1(4),f_2(1)+F_1(3),f_2(2)+F_1(2),f_2(3)+F_1(1),$$
$$f_2(4)+F_1(0)\}=21,\quad x_2(4)=3$$

$$F_2(5)=\max\{f_2(0)+F_1(5),f_2(1)+F_1(4),f_2(2)+F_1(3),f_2(3)+F_1(2),$$
$$f_2(4)+F_1(1),f_2(5)+F_1(0)\}=26,\quad x_2(5)=4$$

这些值和相应的标记函数 $x_2(x)$存在备忘录的第2列. 照此下去,继续计算 $F_3(x)$和 $F_4(x)$,$x=1,2,3,4,5$. 从而完成整个备忘录. 从表中 $F_4(5)=61$ 可知,投资5万元可以产生的最大效益是61万元. 那么能够达到这个效益的分配方案是什么呢? 这需要从表上的标记函数从后向前追溯. 首先查到 $x_4(5)=1$ 知道1万元分配给第4个项目. 于是分配给前三个项目的钱数为 $5-1=4$ 万元. 再查 $x_3(4)=3$,从而知道在4万元中又有3万元分配给第三个项目. 那么还剩下 $4-3=1$ 万元分配给前两个项目. 最后查 $x_2(1)=0$,说明在这1万元中分配给第二个项目的钱数是0,只能全部分配给第一个项目. 经过这样的追溯,可以得到问题的解,即

分配方案是:$x_1=1,x_2=0,x_3=3,x_4=1$

最大效益是:$F_4(5)=61$

对于解的追溯过程可以在表3.4的灰色方格中看到.

表3.4 解的追溯

x	$F_1(x)$	$x_1(x)$	$F_2(x)$	$x_2(x)$	$F_3(x)$	$x_3(x)$	$F_4(x)$	$x_4(x)$
1	11	1	11	0	11	0	20	1
2	12	2	12	0	13	1	31	1
3	13	3	16	2	30	3	33	1
4	14	4	21	3	41	3	50	1
5	15	5	26	4	43	4	61	1

最后,让我们看看这个算法的效率. 假设给定的输入实例含有 n 个项目,钱数为 m,那么备忘录中有 m 行 n 列,共计 mn 项. 根据递推关系

$$F_k(x)=\max_{0\leqslant x_k\leqslant x}\{f_k(x_k)+F_{k-1}(x-x_k)\}$$

$$F_1(x)=f_1(x)$$

除了 $k=1$ 的初值以外,对项 $F_k(x)(2\leqslant k\leqslant n,1\leqslant x\leqslant m)$的计算需要 $x+1$ 次加法和 x 次比较. 对 k 求和,于是算法执行的加法次数满足:

$$\sum_{k=2}^{n}\sum_{x=1}^{m}(x+1)=\frac{1}{2}(n-1)m(m+3)$$

比较次数满足:

$$\sum_{k=2}^{n}\sum_{x=1}^{m}x=\frac{1}{2}(n-1)m(m+1)$$

于是该算法的时间复杂度是 $W(n,m)=O(nm^2)$.

3.3.2 背包问题

背包问题(Knapsack Problem) 是一个著名的 NP 难问题.

例 3.5 一个旅行者准备随身携带一个背包. 可以放入背包的物品有 n 种,物品 j 的重量和价值分别为 w_j 和 v_j,$j=1,2,\cdots,n$. 如果背包的最大重量限制是 b,怎样选择放入背包的物品以使得背包的价值最大?

这是一个组合优化问题,设 x_j 表示装入背包的第 j 种物品的数量,那么目标函数和约束条件是:

$$\text{目标函数}\quad \max\sum_{j=1}^{n} v_j x_j$$

$$\text{约束条件}\quad \begin{cases}\sum_{j=1}^{n} w_j x_j \leqslant b\\ x_j \in \mathbf{N}\end{cases}$$

如果组合优化问题的目标函数和约束条件都是线性函数,称为**线性规划问题**,如果线性规划问题的变量 x_j 都是非负整数,则称为**整数规划问题**. 背包问题就是整数规划问题. 限制所有的 $x_j=0$ 或 $x_j=1$ 时的背包问题称为 **0-1 背包问题**.

不难验证背包问题也满足优化原则,可以使用动态规划设计技术. 与投资问题类似,背包问题也有两个参数,物品种类和背包重量的限制. 可以根据物品种类和背包的重量限制进行子问题划分. 设 $F_k(y)$表示只允许装前 k 种物品,背包总重不超过 y 时背包的最大价值,分两种情况考虑:不装第 k 种物品或者至少装 1 件第 k 种物品. 如果不装第 k 种物品,那么只能用前 $k-1$ 种物品装入背包,而背包的重量限制仍旧是 y,在这种情况下,背包最大价值是 $F_{k-1}(y)$.如果第 k 种物品装了一件,那么背包的价值是 v_k,且重量是 w_k,剩下的物品仍旧需要在前 k 种物品中选择(因为最优的装法可能包含多个第 k 种物品). 于是问题归约为在背包重量限制为 $y-w_k$ 的情况下如何用前 k 种物品装入背包以取得最大价值 $F_k(y-w_k)$. 我们需要从这两种情况中取价值较大的作为最优选择. 于是,得到递推关系与边界条件:

$$F_k(y)=\max\{F_{k-1}(y),F_k(y-w_k)+v_k\}$$

$$F_0(y)=0 \qquad 0\leqslant y\leqslant b$$

$$F_k(0)=0 \qquad 0\leqslant k\leqslant n$$

$$F_1(y)=\left\lfloor\frac{y}{w_1}\right\rfloor v_1$$

$$F_k(y)=-\infty \quad y<0$$

上面公式的初值比较多,$F_0(y)$是背包不装物品的价值,显然等于 0;$F_k(0)$是背包重量限制为 0 时的最大价值,当然也等于 0;$F_1(y)$是只能用第一种物品背包重量限制为 y 时的最大价值,为了保证背包不超重,第一种物品至多能装$\lfloor y/w_1\rfloor$ 个,因此背包价值是$\lfloor y/w_1\rfloor v_1$. 为什么还需要设定当 $y<0$ 时的初值呢? 因为在递推式中有 $F_k(y-w_k)$,在递推过程中,某些 $y-w_k$ 可能得到负值,这意味着背包此刻能够承受的重量已经小于第 k 种物品的重量,实际上是不能装的. 通过令 $F_k(y)=-\infty$(可以选机器中的最小数),使得这个值在与另一个优化函数值比较时被淘汰,从而使得这种装法被排除.

与投资问题相比较,背包问题的递推关系也可以使用与投资问题类似的表述,请读者给出相关的表达式和初值.

考虑下面的实例,其中

$$v_1=1,\quad v_2=3,\quad v_3=5,\quad v_4=9$$
$$w_1=2,\quad w_2=3,\quad w_3=4,\quad w_4=7$$
$$b=10$$

则 $F_k(y)$的计算表如表 3.5 所示,其中省略了所有 $F_k(0)$的值.

表 3.5 优化函数 $F_k(y)$

k	y									
	1	2	3	4	5	6	7	8	9	10
1	0	1	1	2	2	3	3	4	4	5
2	0	1	3	3	4	6	6	7	9	9
3	0	1	3	5	5	6	8	10	10	11
4	0	1	3	5	5	6	9	10	10	12

表 3.5 只是计算了各个子问题的优化函数值,最后一项 $F_4(10)=12$,这就是背包的最大价值. 怎样选择物品可以得到这个价值呢? 可以像投资问题一样设立标记函数,记录得到这个值时所装入的最大标号物品的个数. 这里采用另一种标记方法. 令 $i_k(y)$表示在计算优化函数值 $F_k(y)$时所用到物品的最大标号. 在计算 $F_k(y)$时,如果 $F_{k-1}(y)$比 $F_k(y-w_k)+v_k$ 大,那么此刻装入背包物品的最大标号与计算 $F_{k-1}(y)$所用物品的最大标号一致;反之,选择标号为 k 的物品装入背包才能使得背包价值达到最大,即 $i_k(y)=k$. 于是 $i_k(y)$满足下述递推关系:

$$i_k(y)=\begin{cases} i_{k-1}(y) & F_{k-1}(y)>F_k(y-w_k)+v_k \\ k & F_{k-1}(y)\leqslant F_k(y-w_k)+v_k \end{cases}$$

不难看出,标记函数 $i_k(y)$不需要额外的计算,只需计算 $F_k(y)$时把相关的 $i_k(y)$值填入一个标记表就行了. 标记表的格式如表 3.6 所示,其中省略了所有 $i_k(0)=0$ 的初值.

表 3.6 标记函数 $i_k(y)$

k	y									
	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2	2
3	0	1	2	3	3	3	3	3	3	3
4	0	1	2	3	3	3	4	3	4	4

根据表 3.6 可以向前追踪,找到问题的解. 具体追踪过程是:由 $i_4(10)=4$ 可知 4 号物品至少用 1 个,占用背包重量 $w_4=7$,背包剩余重量是 $10-7=3$,于是继续检查 $i_4(3)$. 由于 $i_4(3)=2$,即剩余物品的最大标号是 2,这说明 2 号物品至少装 1 个,而不再装入第2 个 4 号

物品和 3 号物品. 装入这个 2 号物品后,背包重量又增加了 $w_2=3$,剩余背包重量是 0,于是不能装任何物品了. 用公式表示追踪解的过程是:

$$i_4(10)=4 \Rightarrow x_4 \geqslant 1$$
$$i_4(10-w_4)=i_4(3)=2 \Rightarrow x_2 \geqslant 1, x_4=1, x_3=0$$
$$i_2(3-w_2)=i_2(0)=0 \Rightarrow x_2=1, x_1=0$$

最终得到该实例的解是:$x_1=0, x_2=1, x_3=0, x_4=1$. 上述追踪过程也可以用伪码描述.

算法 3.3 TrackSolution

输入:$i_k(y)$表,其中 $k=1,2,\cdots,n$;$y=1,2,\cdots,b$

输出:$x_1, x_2, \cdots, x_n$,n 种物品的装入量

```
1. for j←1 to n do
2.       x_j←0
3. y←b, j←n
4. j←i_j(y)
5. x_j←1
6. y←y−w_j
7. while i_j(y)=j do
8.     y←y−w_j
9.     x_j←x_j+1
10. if i_j(y)≠0 then goto 4
```

最后考虑算法的时间复杂度. 与投资问题类似,该算法的时间复杂度为 $O(nb)$. 表面上看,这是一个 n 和 b 为变量的多项式,但这个算法并不是多项式时间的算法. 因为正整数 b 的输入规模是 b 在机器中占用的存储空间大小,因此问题的输入规模实际上是 n 和 $\log b$ ($\log b$ 是 b 的二进制表示的位数). 显然 $O(nb)$不是 n 和 $\log b$ 的多项式函数,我们把这样的算法称为**伪多项式时间**的算法. 关于这个问题将在 9.3 节进一步讨论.

背包问题具有广泛的应用背景. 许多任务调度、资源分配、轮船装载等问题都可以用背包问题建模. 背包问题也有许多变种,比如在原有背包问题的基础上增加体积约束条件,即物品 i 不但有重量 w_i、价值 v_i,还有体积 d_i,背包本身也有体积 D 的限制. 作为典型的 NP 难问题,许多人对背包问题进行了深入的研究,特别对 0-1 背包问题已经得到了有效的近似算法. 这些将在第 10 章介绍.

3.3.3 最长公共子序列 LCS

在实际问题中经常需要对两个对象进行类比分析,找出它们之间的公共部分. 最长公共子序列问题就是这类问题的一种简单的抽象模型.

定义 3.1 设 X 和 Z 是两个序列,其中

$$X=<x_1, x_2, \cdots, x_m>$$
$$Z=<z_1, z_2, \cdots, z_k>$$

如果存在 X 的元素构成的按下标严格递增序列$<x_{i_1}, x_{i_2}, \cdots, x_{i_k}>$,使得 $x_{i_j}=z_j$,$j=1,2,\cdots,k$,那么称 Z 是 X 的**子序列**. Z 含有的元素个数,称为**子序列的长度**.

定义 3.2 设 X 和 Y 是两个序列,如果 Z 既是 X 的子序列,也是 Y 的子序列,则称

Z 是 X 与 Y 的**公共子序列**.

例 3.6 最长公共子序列问题.

给定序列

$$X=<x_1,x_2,\cdots,x_m>,\quad Y=<y_1,y_2,\cdots,y_n>$$

求 X 和 Y 的最长公共子序列.

例如,$X=<A,B,C,B,D,A,B>$,$Y=<B,D,C,A,B,A>$,它们的一个最长的公共子序列是$<B,C,B,A>$,长度是 4. 当然还有另一个最长公共子序列,即$<B,C,A,B>$. 我们的要求是求出一个最长的公共子序列. 当最长公共子序列不唯一时,不同的算法的求解结果可能不一样,但是它们的长度都相等.

假设 $m\leqslant n$. 蛮力算法可以这样做:找出 X 的每个子序列 X',并且把 X'和 Y 进行比较,看看 X'是否也是 Y 的子序列. 如果是,那么 X'就是 X 与 Y 的公共子序列. 当所有的比较完成后就找到了 X 与 Y 的最长公共子序列. 在选择 X'时,每个 X 的元素有两种可能的选择:属于 X'还是不属于 X',于是 X 有 2^m 个子序列. 如果检查 X'与 Y 需要 $O(n)$时间,那么整个算法需要 $O(n2^m)$时间,这是一个指数时间的算法.

考虑动态规划算法,首先需要思考如何界定子问题的边界. 假设子问题的 X 和 Y 的起始位置都从第一个元素开始,X 的终止位置是第 i 个元素,Y 的终止位置是第 j 个元素,那么

$$X_i=<x_1,x_2,\cdots,x_i>,\quad Y_j=<y_1,y_2,\cdots,y_j>$$

就代表了由 i 和 j 共同界定的子问题的输入.

下面分析子问题之间的依赖性质,这是设计递推关系的基础. 设

$$X_i=<x_1,x_2,\cdots,x_i>,\quad Y_j=<y_1,y_2,\cdots,y_j>,\quad Z_k=<z_1,z_2,\cdots,z_k>$$

如果 Z_k 为 X_i 和 Y_j 的最长公共子序列,那么

(1) 若 $x_i=y_j$,则 $z_k=x_i=y_j$,且 Z_{k-1}是 X_{i-1}与 Y_{j-1}的最长公共子序列. 如若不然,一定存在 X_{i-1}和 Y_{j-1}的最长公共子序列 Z',$|Z'|>|Z_{k-1}|$. 在 Z'后面加上 z_k 就得到一个 X_i 与 Y_j 的公共子序列,且它的长度为$|Z'|+1>|Z_{k-1}|+1=|Z_k|$,与 Z_k 是 X_i 与 Y_j 的最长公共子序列矛盾.

如果 x_i 与 y_j 不等,或 x_i 不是 z_k,或 y_j 不是 z_k. 下面的(2)和(3)分别对应于这两种情况.

(2) 若 $x_i\neq y_j$,$z_k\neq x_i$,则 Z_k 是 X_{i-1}与 Y_j 的最长公共子序列.

(3) 若 $x_i\neq y_j$,$z_k\neq y_j$,则 Z_k 是 X_i 与 Y_{j-1}的最长公共子序列.

设 $C[i,j]$表示 X_i 与 Y_j 的最长公共子序列的长度. 根据上述分析,得到优化函数的递推关系如下:

$$C[i,j]=\begin{cases}C[i-1,j-1]+1 & \text{如果 } i,j>0,x_i=y_j\\ \max\{C[i,j-1],C[i-1,j]\} & \text{如果 } i,j>0,x_i\neq y_j\end{cases}$$

$$C[0,j]=C[i,0]=0\quad \text{如果 } 1\leqslant i\leqslant m,1\leqslant j\leqslant n$$

上述的递推公式恰好反映了前面的三种情况. $C[i,j]=C[i-1,j-1]+1$ 对应于情况(1);$C[i-1,j]$与 $C[i,j-1]$则分别对应于情况(2)和(3). 当 $i=0$ 或者 $j=0$ 时,两个序列中的一个序列是空序列,那么公共子序列只能是空序列,因此 $C[0,j]=C[i,0]=0$,这是递推关系的初值.

根据这个递推关系和初值,不难给出算法迭代实现的伪码描述.

算法 3.4 LCS(X,Y,m,n)

输入：序列 X,Y,其中 $X[1..m]$,$Y[1..n]$

输出：最长公共子序列长度 $C[i,j]$,标记 $B[i,j]$,$1\leqslant i\leqslant m$,$1\leqslant j\leqslant n$

```
1. for i←1 to m do                          //第 1～4 行处理初值
2.     C[i,0]←0
3. for i←1 to n do
4.     C[0,i]←0
5. for i←1 to m do
6.     for j←1 to n do
7.         if X[i]=Y[j]                     //X[i]与 Y[j]被选入公共子序列
8.         then C[i,j]←C[i-1,j-1]+1
9.              B[i,j]←"↖"
10.        else if C[i-1,j]≥C[i,j-1]
11.             then C[i,j]←C[i-1,j]
12.                  B[i,j]←"↑"
13.             else C[i,j]←C[i,j-1]
14.                  B[i,j]←"←"
```

上述算法计算了所有子问题的最长公共子序列的长度 $C[i,j]$. $B[i,j]$是标记函数，记录 X_i 与 Y_j 的最长公共子序列的元素是怎样选取的. 有三种标记：①用“↖”表示序列 X_i 与 Y_j 的最后项 $x_i=y_j$，已被选入最长公共子序列；②用“↑”表示在 X_i 与 Y_j 的最长公共子序列选择时不考虑 x_i，它们的最长公共子序列就是 X_{i-1} 与 Y_j 的最长公共子序列；③用“←”表示在 X_i 与 Y_j 的最长公共子序列选择时不考虑 y_j，它们的最长公共子序列就是 X_i 与 Y_{j-1} 的最长公共子序列. 设立这些标记是为了追踪解. 可以从后向前追踪这些标记，遇到“↖”标记时，就把 x_i 加入最长公共子序列中. 如果遇到其他两种标记，表示没有元素加入最大公共子序列，这时仅需按照标记的指示向前继续追踪. 追踪解的算法在下面给出.

算法 3.5 Structure Sequence(B,i,j)

输入：$B[i,j]$

输出：X 与 Y 的最长公共子序列

```
1. if i=0 or j=0 then return                //一个序列为空
2. if B[i,j]="↖"
3. then 输出 X[i]
4.      Structure Sequence(B,i-1,j-1)
5. else if B[i,j]="↑" then Structure Sequence(B,i-1,j)
6.      else Structure Sequence (B,i,j-1)
```

回顾这个问题开始时所给出的实例：

$$X=<A,B,C,B,D,A,B>,\quad Y=<B,D,C,A,B,A>$$

其中，$m=7$,$n=6$. 算法 LCS 计算的 $C[i,j]$和 $B[i,j]$分别如表 3.7 和表 3.8 所示. 在表 3.8 中用连续的灰色表格表示追踪过程：

$$B[7,6]\to B[6,6]\to B[5,5]\to B[4,5]\to B[3,4]\to$$
$$B[3,3]\to B[2,2]\to B[2,1]\to j=0$$

其中，$B[6,6]=B[4,5]=B[3,3]=B[2,1]=$“↖”，这就得到解 $Z=<x_2,x_3,x_4,x_6>=$

$<B,C,B,A>$.

最后我们分析这个算法的时间复杂度. 算法 LCS 的第 1 行和第 2 行是 $O(m)$时间,第 3 行和第 4 行是 $O(n)$时间,第 5 行和第 6 行的循环执行 $O(mn)$次,循环内部的运算是 $O(1)$时间,于是算法的总时间是 $O(mn)$. 此外,追踪解的 Structure Sequence 算法从 $i=m,j=n$ 开始,每找到一个标记 $B[i,j]$,算法执行常数操作,然后或者 i 和 j 同时减 1,或者 i 减 1,或者 j 减 1,总之$i+j$ 至少减 1. 由于初始 $i+j=m+n$,因此至多 $m+n$ 次在标记上的操作,算法将结束,于是算法的时间是 $O(m+n)$. 我们看到这个算法把蛮力算法的 $O(n2^m)$时间降低到 $O(mn)$时间.

表 3.7 优化函数 $C[i,j]$

	0	1	2	3	4	5	6
0	$C[0,0]=0$	$C[0,1]=0$	$C[0,2]=0$	$C[0,3]=0$	$C[0,4]=0$	$C[0,5]=0$	$C[0,6]=0$
1	$C[1,0]=0$	$C[1,1]=0$	$C[1,2]=0$	$C[1,3]=0$	$C[1,4]=1$	$C[1,5]=1$	$C[1,6]=1$
2	$C[2,0]=0$	$C[2,1]=1$	$C[2,2]=1$	$C[2,3]=1$	$C[2,4]=1$	$C[2,5]=2$	$C[2,6]=2$
3	$C[3,0]=0$	$C[3,1]=1$	$C[3,2]=1$	$C[3,3]=2$	$C[3,4]=2$	$C[3,5]=2$	$C[3,6]=2$
4	$C[4,0]=0$	$C[4,1]=1$	$C[4,2]=1$	$C[4,3]=2$	$C[4,4]=2$	$C[4,5]=3$	$C[4,6]=3$
5	$C[5,0]=0$	$C[5,1]=1$	$C[5,2]=2$	$C[5,3]=2$	$C[5,4]=2$	$C[5,5]=3$	$C[5,6]=3$
6	$C[6,0]=0$	$C[6,1]=1$	$C[6,2]=2$	$C[6,3]=2$	$C[6,4]=3$	$C[6,5]=3$	$C[6,6]=4$
7	$C[7,0]=0$	$C[7,1]=1$	$C[7,2]=2$	$C[7,3]=2$	$C[7,4]=3$	$C[7,5]=4$	$C[7,6]=4$

表 3.8 标记函数 $B[i,j]$

	1	2	3	4	5	6
1	$B[1,1]=\uparrow$	$B[1,2]=\uparrow$	$B[1,3]=\uparrow$	$B[1,4]=\nwarrow$	$B[1,5]=\leftarrow$	$B[1,6]=\nwarrow$
2	$B[2,1]=\nwarrow$	$B[2,2]=\leftarrow$	$B[2,3]=\leftarrow$	$B[2,4]=\uparrow$	$B[2,5]=\nwarrow$	$B[2,6]=\leftarrow$
3	$B[3,1]=\uparrow$	$B[3,2]=\uparrow$	$B[3,3]=\nwarrow$	$B[3,4]=\leftarrow$	$B[3,5]=\uparrow$	$B[3,6]=\uparrow$
4	$B[4,1]=\nwarrow$	$B[4,2]=\uparrow$	$B[4,3]=\uparrow$	$B[4,4]=\uparrow$	$B[4,5]=\nwarrow$	$B[4,6]=\leftarrow$
5	$B[5,1]=\uparrow$	$B[5,2]=\nwarrow$	$B[5,3]=\uparrow$	$B[5,4]=\uparrow$	$B[5,5]=\uparrow$	$B[5,6]=\uparrow$
6	$B[6,1]=\uparrow$	$B[6,2]=\uparrow$	$B[6,3]=\uparrow$	$B[6,4]=\nwarrow$	$B[6,5]=\uparrow$	$B[6,6]=\nwarrow$
7	$B[7,1]=\nwarrow$	$B[7,2]=\uparrow$	$B[7,3]=\uparrow$	$B[7,4]=\uparrow$	$B[7,5]=\nwarrow$	$B[7,6]=\uparrow$

3.3.4 图像压缩

计算机中的图像由一系列像点构成,每个像点称为一个像素,图像分辨率越高,使用的像素就越多. 例如,Windows 桌面的图片经常使用的像素设置是 1024×768,大概达到 10^6 量级. 图像传输和视频处理有时在 1 秒内要处理几十帧图片,这些图片的像素就很可观了,因此,图像处理常常需要大量的存储空间和高的处理速度,图像压缩问题就成了计算机科学技术中的重要研究课题之一.

以黑白图像的处理来说明图像压缩中的问题. 每幅黑白图像由像点构成,每个像点具有灰度值,用 0～255 的整数表示. 如果每个整数都用相同的二进制位来表示,那么需要用 8 个二进制位. 假设一幅图像有 n 个像素,那么这 n 个像素的灰度值构成一个整数序列

$$P=<p_1,p_2,\cdots,p_n>$$

其中,p_i 表示第 i 个像素的灰度. 存储这幅图片时,可以像数组一样连续把这些整数存起来,共需要 $8n$ 个二进制位.

下面考虑一种图像压缩方法. 一般来说,在一幅图片中许多连续区域中像点的灰度值是接近的. 例如,有些交通标志图片,大片的区域是白的,可能少量区域有颜色,而且是比较单调的颜色. 对这样的图片是否可以采用分段存储的方法:对灰度值较小的段的像素采用比较少的位数,如 2 位;对灰度值较大的段的像素采用较多的位数,如 8 位,这样就可能减少空间的占用. 这就是变位压缩技术的基本想法. 这种技术节省了空间,但在读取图像时带来了新的问题. 在每个像素 8 位的存储方法中,读取图像时每 8 位就是一个像素的灰度值,不会出错. 但是对于分段压缩的图像,看起来就是一个长长的 0-1 序列. 当读取这个序列时,怎么知道每段的划分位置及每段像素占用的二进制位数呢? 这里需要对段的划分和段中像素使用的二进制位数(要求同一段内不同像素用的存储位数都一样)给出明确的信息. 为此,我们对每个段给出两个整数值,一个表示该段含有的像素个数,一个表示每个像素所占用的二进制位数. 比如第 i 段,有 l_i 个像素,每个像素用 b_i 位. 由于某些技术要求,规定每段像素总数不超过 256,即 $l_i\leqslant 256$. 于是,可以用 8 位来表示 l_i(8 位二进制数恰好有 256 个值). 此外,由于每个灰度值为 0～255,表达每个灰度值所用二进制数的位数 b_i 不超过 8,于是记录 b_i 还需要三个二进制位. 对每段来说,这额外的 11 位作为段头信息. 分段越多,每段内部像素所占用的位数会减少,但过多的段头会消耗较多的二进制位;相反,分段越少,段内像素的空间消耗会增加,但是段头消耗少.

请看下面的例子. 设输入的灰度值序列是:

$$P=<10,12,15,255,1,2,1,1,2,2,1,1>$$

考虑下面三种分段方法:

分法 1　$S_1=<10,12,15>$,　$S_2=<255>$,　$S_3=<1,2,1,1,2,2,1,1>$

分法 2　$S_1=<10,12,15,255,1,2,1,1,2,2,1,1>$

分法 3　$S_1=<10>$,　$S_2=<12>$,　$S_3=<15>$,　$S_4=<255>$,　$S_5=<1>$,
$S_6=<2>$,　$S_7=<1>$,　$S_8=<1>$,　$S_9=<2>$,　$S_{10}=<2>$,
$S_{11}=<1>$,　$S_{12}=<1>$

分法 1 有 3 段,第 1 段 3 个像素,每个像素用 4 位;第 2 段 1 个像素,每个像素用 8 位;第 3 段 8 个像素,每个像素用 2 位;加上 3 个段头,每个段头 11 位,总计位数是:

$$4\times 3+8\times 1+2\times 8+3\times 11=69$$

分法 2 有 1 段,12 个像素,每个像素用 8 位,段头 11 位,总计位数是:

$$8\times 12+11=107$$

分法 3 有 12 段,前 3 段的像素用 4 位,第 4 段像素用 8 位,后面有 5 段像素用 1 位,3 段像素用 2 位,还有 12 个段头,每个 11 位,总计位数是:

$$4\times 3+8\times 1+1\times 5+2\times 3+11\times 12=163$$

看起来分法 1 占用的位数最少. 我们的问题是寻找存储位数最少的分段方法.

例 3.7 图像压缩问题.

给定非负整数序列 $P=<p_1,p_2,\cdots,p_n>$,其中 p_i 为第 i 个像素的灰度值,采用变位压缩存储格式,如何对 P 进行分段,以使得存储 P 占用的二进制位数达到最少?

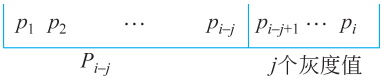

图 3.5 子问题的归约

使用动态规划算法. 先考虑子问题的划分边界. 这个问题比较简单,每个子问题的输入都从 p_1 开始,第 i 个子问题的输入 P_i 到第 i 个像素值 p_i 结束,即 $P_i=<p_1,p_2,\cdots,p_i>$. 如图 3.5 所示,如果最后一段的灰度值有 j 个,其中 $1\leqslant j\leqslant\min\{256,i\}$,那么剩余部分将归约为输入是 $P_{i-j}=<p_1,p_2,\cdots,p_{i-j}>$ 的子问题.

下面考虑子问题之间的依赖关系. 设 $S[i]$表示输入为 P_i 时按最优分段存储所使用的二进制位数. 假设最后一段是第 m 段,记作 S_m,且 S_m 含有 j 个灰度值,即 $S_m=<p_{i-j+1},\cdots,p_i>$. 存储 S_m 的每个灰度值需要的二进制位数记作 $b[i-j+1,i]$,它应该是用二进制表示 S_m 中的最大灰度值所占用的位数,即

$$b[i-j+1,i]=\left\lceil \log(\max_{p_k\in S_m} p_k+1)\right\rceil\leqslant 8$$

由于 S_m 含有 j 个灰度值,于是需要 $j\times b[i-j+1,i]$个二进制位. 剩下就是考虑前 $i-j$ 个灰度值的最优分段所占用的位数,这恰好是子问题 P_{i-j} 的最优解 $S[i-j]$. 于是在最后一段为 j 个灰度值时的最少的位数是:

$$S[i-j]+j\times b[i-j+1,i]+11$$

其中 11 是该段段头的空间消耗. 最后分段的灰度值个数 j 可以选择 $1,2,\cdots,\min\{256,i\}$中的每一个值,我们必须对所有可能的 j 值进行上述计算,并从中选出最小的位数值,这个最小值就是 $S[i]$. 根据上述分析,不难得到关于优化函数 $S[i]$的递推关系:

$$S[i]=\min_{1\leqslant j\leqslant\min\{i,256\}}\{S[i-j]+j\times b[i-j+1,i]+11\}$$

$$S[0]=0$$

算法实现采用迭代方法,子问题的输入从 $P_1,P_2,\cdots$,最后达到 P_n. 对给定的输入 P_i,算法从最后分段开始考虑,依次处理像素数为 $1,2,\cdots$,直到 $\min\{256,i\}$的分段结果. 用 $S[i]$保留对应于最优分段的最小位数,并用了 $l[i]$记录达到最少位数时最后一段的灰度值个数. 如果随着分段像素灰度值个数的增加,得到了比当前的位数更少的位数,那么就用新的位数替换原来的位数,同时更新最后段的像素个数 $l[i]$. 算法的伪码描述如下:

算法 3.6 Compress(P,n)

输入:数组 $P[1..n]$

输出:最小位数 $S[n]$,$l[1..n]$

```
1. Lmax←256; header←11; S[0]←0          //Lmax 为最大段长,header 为每个段头占用空间
2. for i←1 to n do
3.    b[i]←length(P[i])                  //表示第 i 个像素灰度的二进制位数
4.    bmax←b[i]                          //第 3~6 行分法的最后一段只有 P[i]一个像素
5.    S[i]←S[i-1]+bmax
6.    l[i]←1
7.    for j←2 to min{i,Lmax} do          //最后段含 j 个像素,j=2,…,min{i,256}
8.        if bmax<b[i-j+1]               //第 8~9 行统一段内表示像素的二进制位数
```

```
9.      then bmax←b[i−j+1]
10.     if S[i]>S[i−j]+ j * bmax          //找到更少的位数
11.     then S[i]←S[i−j]+j * bmax
12.          l[i]←j
13.  S[i]←S[i]+header
```

下面给出一个运行实例. 设输入 $P=<10,12,15,255,1,2>$. 假设 $S[1],S[2],\cdots,S[5]$已经计算完成,且

$$S[1]=15,\quad S[2]=19,\quad S[3]=23,\quad S[4]=42,\quad S[5]=50,$$
$$l[1]=1,\quad l[2]=2,\quad l[3]=3,\quad l[4]=1,\quad l[5]=2$$

算法迭代计算的最后一个子问题,即对 $i=6$ 的计算过程说明如下:

初始分段,最后段含有 1 个像素灰度(算法 3.6 的第 3 行到第 6 行),即 $j=1$,这时该像素灰度值是 2,占用位数bmax$=b[6,6]=2$,加上段头 11 位[①],于是得到

$$S[6]=S[5]+1\times 2+11=50+13=63,\quad l[6]=1$$

接着 $j=2$,最后段含有两个像素灰度值,这时 bmax$=b[5,6]=2$,于是最后一段占用位数等于 $2\times 2=4$ 位,这个位数计算出是 57,即

$$S[6]=S[4]+2\times 2+11=42+15=57,\quad l[6]=2$$

57 比 63 小,于是更新了 $S[6]$和 $l[6]$. 下一步 $j=3$,最后段含有 3 个像素灰度值,这时 bmax$=b[4,6]=8$,于是最后一段占用位数等于 $3\times 8=24$ 位,计算结果是:

$$S[3]+3\times 8+11=23+35=58$$

由于这个数比 57 大,于是不再更新 $S[6]$. 后面几步计算和这个过程类似,得到的最小位数分别是 62,66,59,它们都大于 57,因此最终保留的 $S[6]=57$,$l[6]=2$. 图 3.6 给出了整个计算步骤,灰色区域表示最后一段的范围.

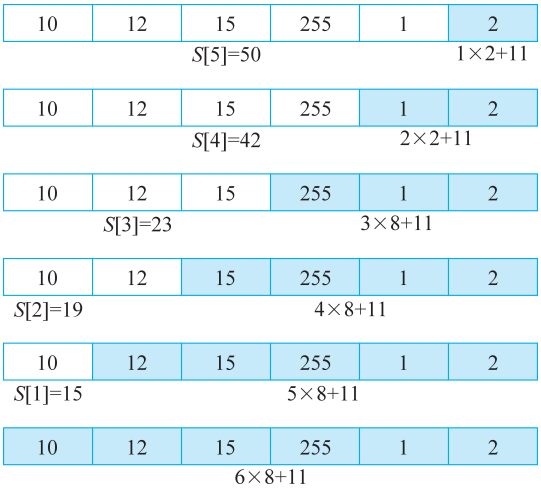

图 3.6　一个实例

① 把段头所占用的 11 位加上,是为了与图 3.6 中不同划分占用的位数一致. 在算法 3.6 的伪码中,不是每次划分后都做 1 次加法,而是找到最优划分、确定了最后分段的位置之后才加段头的 11 位. 两种方法的最优划分和占用位数是一样的,但这个过程比算法做的加法次数更多.