第 5 章　动态规划法

5.1　动态规划概述

假设某公司有一条数据需从设备 A 传到设备 E，又需经过 B、C、D 3 种设备的确认和处理，其中 B_1、B_2 和 B_3 为 B 类设备，C_1、C_2 和 C_3 为 C 类设备，D_1 和 D_2 为 D 类设备，边上权重代表设备之间传递并处理数据需花费的时间，如图 5.1 所示。一条消息需在最短时间内从设备 A 传递到设备 E，那么应该如何规划数据传输路径？

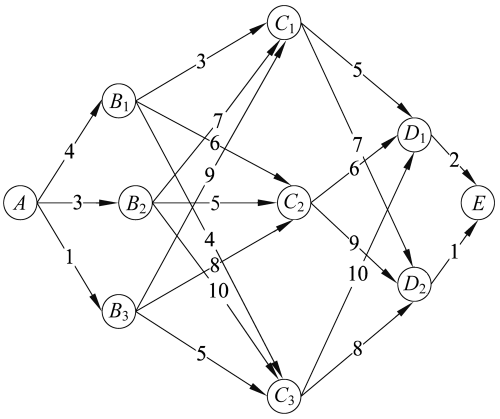

图 5.1　数据传输处理时间图

消息要从 A 到达 E，可分为 AB、BC、CD 和 DE 4 个阶段，要想获得最短花费时间，可从最后一段开始计算其最短耗时。

(1) DE 阶段：从 D_1 到 E 的最小耗时 $T(D_1E)=2$；从 D_2 到 E 的最小耗时 $T(D_2E)=1$。

(2) CE 阶段：从 C_1 到 E 的最小耗时 $T(C_1E)=\min\{T(C_1D_1)+T(D_1E),T(C_1D_2)+T(D_2E)\}=7$；从 C_2 到 E 的最小耗时 $T(C_2E)=8$；从 C_3 到 E 的最小耗时 $T(C_3E)=9$。

(3) BE 阶段：从 B_1 到 E 的最小耗时 $T(B_1E)=\min\{T(B_1C_1)+T(C_1E),T(B_1C_2)+T(C_2E),T(B_1C_3)+T(C_3E)\}=10$；从 B_2 到 E 的最小耗时 $T(B_2E)=13$；从 B_3 到 E 的最小耗时 $T(B_3E)=14$。

(4) AE 阶段：从 A 到 E 的最小耗时 $T(AE)=\min\{T(AB_1)+T(B_1E),T(AB_2)+T(B_2E),T(AB_3)+T(B_3E)\}=14$。

从上述计算过程可看出，以 $AB_1C_1D_1E$ 的路径传递消息，花费最少耗时为 14，该方法通过将原问题拆分为重叠子问题，然后组合这些重叠子问题的解来获得原问题的解，体现了经典的动态规划思想。

动态规划法(Dynamic Programming)通常用来解决最优化问题，即找出问题的最优解。在最优化问题的求解过程中，可能会得到多个可行解，希望从这些解中可找到具有最优值的解。一般地，只要实际问题可划分为规模更小的子问题，且原问题的最优解中包含了子问题的最优解，则可考虑用动态规划法求解。动态规划法的基本思想是：将原问题分解成多个子问题(子问题之间并不独立)，先计算出子问题的解，再结合这些子问题的解得到原问题的解。动态规划算法包括 4 个基本步骤：刻画最优解的结构特征，递归定义最优解的值，自底向上计算最优解的值，基于最优值构造最优解。

使用动态规划法求解最优化问题，应满足以下两个性质：最优子结构性质；重叠子问题性质。

(1) 最优子结构性质，是指原问题的最优解包含其子问题的最优解。动态规划法通常自底向上地使用最优子结构性质，首先将原问题划分成若干子问题，求得子问题的最优解，从而逐步构造出原问题的最优解。

(2) 重叠子问题性质，是指在自顶向下求解问题时，每次产生的子问题并不总是新问题，某些子问题会被重复计算多次。动态规划算法利用重叠子问题性质，对每一个子问题只计算一次，其计算结果保存在一个表格中，当再次需要计算已计算过的子问题时，只需要在表格中简单地查找结果，从而避免每次求解子问题时的重复计算，保证算法的高效性。

本章以 0-1 背包问题为代表，介绍动态规划算法的基本思想和主要步骤，并分析算法的时间复杂度。

5.2 0-1 背包问题

给定 n 种物品和一个背包，物品 i 的质量为 w_i，物品对应的价值为 v_i，背包的总容量为 c。应如何选择物品装入背包，使得装入背包中物品的总价值达到最大？

在选择装入背包的物品时，对每种物品 i 只有两种选择方式，即装入背包或不装入背包。规定每个物品 i 不能装入背包多次，也不能将物品 i 切割后装入背包中。因此，此类问题被称为 0-1 背包问题(1 和 0 分别代表物品装入背包和不装入背包)。

假设给定 $c>0, w_i>0, v_i>0 (1\leqslant i\leqslant n)$，要求找出一个 n 元 0-1 向量 $(x_1, x_2, \cdots, x_n)$，$x_i\in\{0,1\} (1\leqslant i\leqslant n)$，使得 $\sum_{i=1}^{n} w_i x_i \leqslant c$，且 $\sum_{i=1}^{n} v_i x_i$ 达到最大。因此，0-1 背包问题可看作一个特殊的整数规划问题，其形式化描述如下。

$$\max \sum_{i=1}^{n} v_i x_i, \quad \text{s.t.} \sum_{i=1}^{n} w_i x_i \leqslant c \tag{5-1}$$

其中，$x_i\in\{0,1\}, 1\leqslant i\leqslant n$。

5.2.1 最优子结构性质

下面证明 0-1 背包问题具有最优子结构性质。

设$(y_1,y_2,\cdots,y_n)$是 0-1 背包问题求得的一个最优解，则$(y_2,y_3,\cdots,y_n)$是其中相应子问题的一个最优解，于是

$$\max\sum_{i=2}^{n}v_ix_i,\quad \text{s.t.}\sum_{i=2}^{n}w_ix_i\leqslant c-w_1y_1 \tag{5-2}$$

其中，$x_i\in\{0,1\}$，$2\leqslant i\leqslant n$。否则，设$(z_2,z_3,\cdots,z_n)$是上述子问题的一个最优解，而$(y_2,y_3,\cdots,y_n)$不是其最优解，则有$\sum_{i=2}^{n}v_iz_i>\sum_{i=2}^{n}v_iy_i$，且$w_1y_1+\sum_{i=2}^{n}w_iz_i\leqslant c$，因此

$$v_1y_1+\sum_{i=2}^{n}v_iz_i>\sum_{i=1}^{n}v_iy_i \tag{5-3}$$

$$w_1y_1+\sum_{i=2}^{n}w_iz_i\leqslant c \tag{5-4}$$

那么，$(y_1,y_2,\cdots,y_n)$是所给 0-1 背包问题的一个更优解，从而$(y_2,y_3,\cdots,y_n)$不是所给 0-1 背包问题的最优解，矛盾。

5.2.2　递推式

设 0-1 背包问题的子问题的最优值为$b(i,j)$，表示前 i 个物品在背包容量为 j 时获得的最大价值。由 0-1 背包问题的最优子结构性质，可建立如下动态规划的递推式$b(i,j)$。

$$b(i,j)=\begin{cases}\max\{b(i-1,j),b(i-1,j-w_i)+v_i\}, & j\geqslant w_i\\ b(i-1,j), & 0\leqslant j<w_i\end{cases} \tag{5-5}$$

(1) 若装不下当前物品 w_i，那么装入前 i 个物品策略的最优值为装入前 $i-1$ 个物品策略的最优值。

(2) 若装得下当前物品 w_i，那么：

① 装入当前物品 w_i，且需给当前物品预留足够的空间装入，那么装入前$(i-1)$个物品策略的最优值加上当前物品的价值即为装入前 i 个物品策略的最优值。

② 不装入当前物品 w_i，则装入前 i 个物品策略的最优值与装入前$(i-1)$个物品策略的最优值相同。

③ 最后选取①和②中的最大值，作为当前策略的最大价值。

5.2.3　算法步骤

(1) 0-1 背包问题的动态规划算法。

算法 5.1 给出求解 0-1 背包问题的动态规划算法。

算法 5.1　0-1 Knapsak //0-1 背包的动态规划算法

输入：

$\{w_1,w_2,\cdots,w_n\}$：n 个物品的质量；$\{v_1,v_2,\cdots,v_n\}$：n 个物品的价值；c：背包容量

输出：

$\boldsymbol{b}$：价值矩阵　//$\boldsymbol{b}$ 中的元素$b(i,j)$表示前面 i 个物品在 j 容量时的最大价值

步骤：

```
1.  初始化矩阵 b                                    //b 的大小为(n+1)×(c+1)
2.  b(0,n)←0;b(n,0)←0                               //第一行第一列置空
3.  For j=1 To c Do
4.      For i=1 To n Do
5.          If j≥w_i Then
6.              Value0←b(i-1,j)                     //Value0 为不装入物品 i 时背包的价值
7.              Value1←b(i-1,j-w_i)+v_i             //Value1 为装入物品 i 时背包的价值
8.              b(i,j)←max{Value0,Value1}
9.          Else
10.             b(i,j)←b(i-1,j)                     //不装入物品 i
11.         End If
12.     End For
13. End For
14. Return b
```

例 5.1 假设有 5 个物品，质量分别为 2、2、6、5、4，其价值分别为 6、3、5、4、6，背包容量为 10，表 5.1 展示了使用算法 5.1 求解该实例的过程。

表 5.1 背包容量价值表

物品	容量										
	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14
5	0	0	6	6	9	9	12	12	15	15	15

① 第一行的值为 0，因为物品标号为 0（考虑前 0 个物品的最优值）的情况下（即没有物品）最大价值都是 0。

② 第一列的值为 0，因为在当前背包容量为 0 的情况下最大价值都是 0。

③ 当 $i=2$、$j=4$ 时，$w_2=2$，由递推式 $j>w_2$，因此，物品 2 可装入背包，也可不装入背包，此时比较装入或不装物品 2 时的最大价值：当物品 2 不装入背包时的价值为 $b(2,4)=6$；当物品 2 装入背包时的价值为 $b(2,4)=b(2-1,4-2)+3=6+3=9$；取两者中的最大值 $\max\{6,9\}=9$，即此时背包最大价值为 9。

④ 当 $i=3$、$j=4$ 时，$w_3=6$，由递推式 $j<w_3$，因此，物品 3 不可装入背包，此时背包最大价值取 $b(3,4)=9$。

⑤ 继续迭代，直至填满表格。

⑥ 回溯：自顶向下往前推，当 $i=5$、$j=10$ 时，$b(5,10)=15$ 表示当背包容量为 10 时，考虑前 5 个物品的装入策略，所能装入的最优值为 15。根据递推式判断是否应该装入编号为 5 的物品，由于 $b(5-1,10)\neq b(5,10)$，此时判断物品 5 装入。装入物品 5 后，需给当前物品

预留相应的背包容量，回到没有装入物品 5 之前，即 $b(5-1,10-4)=b(4,6)$。接着，继续判断物品 4 是否装入。

因此，从表的右下方开始回溯，如果发现前 n 个物品的最优策略的价值和前 $n-1$ 个物品最优策略的价值相同，就说明第 n 个物品未装入背包。否则，第 n 个物品装入背包。

算法 5.1 的时间复杂度为 $O(nc)$，在实际应用问题中往往并不适用，局限在于无法处理非整数的物品价值或背包容量，且当背包容量 $c>2^n$ 时，算法 5.1 的时间复杂度为 $O(n2^n)$，呈指数增长。通过观察算法 5.1 构建的二维表，发现只有部分“跳跃间断点”会对结果产生影响。因此，对算法 5.1 进行改进，得到求解 0-1 背包问题的跳跃间断点算法。

(2) 0-1 背包问题的跳跃间断点算法。

事实上，观察递推式中 $b(i,j)$ 的表达式可知，$b(i,j)\geqslant b(i,j-1)$，当 i 值确定时，$b(i,j)$ 的值是关于背包容量 j 的单调非递减函数，将 j 的取值扩展到实数域时仍成立，此时 $b(i,j)$ 的值是关于背包容量 j 的阶梯状单调非递减函数，存在决定 $b(i,j)$ 值的跳跃间断点，如图 5.2 所示。

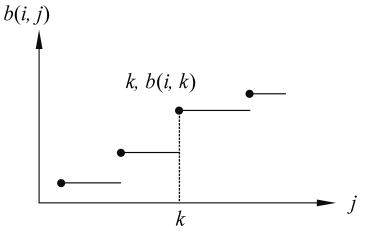

图 5.2　$b(i,j)$ 函数

对于 $b(i,j)$ 的跳跃间断点，由于 $b(i,j)=\max\{b(i-1,j),b(i-1,j-w_i)+v_i\}$，那么 $b(i,j)$ 的跳跃间断点包含于 $b(i-1,j)$ 的跳跃间断点集或 $b(i-1,j-w_i)+v_i$ 的跳跃间断点集中。跳跃间断点集 $p(i-1)$ 由函数 $b(i-1,j)$ 确定，对应不装物品 i 的所有情况；跳跃间断点集 $q(i-1)$ 由函数 $b(i-1,j-w_i)+v_i$ 确定，对应装入物品 i 的所有情况，且有 $q(i-1)=\{(j+w_i,b(i-1,j)+v_i)\mid(j,b(i-1,j))\in p(i-1),j+w_i\leqslant c\}$，即装入物品 i 时，对跳跃断点集 $p(i-1)$ 中每个元素加上 (w_i,v_i)。假设 (c_1,v_1) 和 (c_2,v_2) 为 $p(i-1)\cup q(i-1)$ 中的两个跳跃间断点，则当 $c_2\geqslant c_1$ 或 $v_2<v_1$ 时，(c_2,v_2) 受控于 (c_1,v_1)，从而 (c_2,v_2) 不是 $p(i)$ 中的跳跃间断点。除受控跳跃间断点外，$p(i-1)\cup q(i-1)$ 中的其他跳跃间断点都是 $p(i)$ 中的跳跃间断点。那么，跳跃间断点集 $p(i)$ 可由跳跃间断点集 $p(i-1)$ 递归地构造，即首先依据跳跃间断点集 $p(i-1)$ 计算得到跳跃间断点集 $q(i-1)$，再合并 $p(i-1)$ 和 $q(i-1)$ 两个跳跃间断点集，并去除其中的受控跳跃点，即可得到跳跃间断点集 $p(i)$，最终查询跳跃间断点集 $p(i)$ 得到背包放入物品的最优规划。算法 5.2 给出了解决 0-1 背包问题的跳跃间断点算法。

算法 5.2　0-1 背包跳跃间断点算法

输入：

$\{w_1,w_2,\cdots,w_n\}$：n 个物品的质量；$\{v_1,v_2,\cdots,v_n\}$：n 个物品的价值；c：背包容量

输出：

$P=\{p(i)\mid 0\leqslant i\leqslant n\}$：背包价值全部跳跃间断点集合

//$p(i)$ 中的每个元素 (c_k,v_k) 为一个跳跃间断点的二元组，表示装入前 i 个物品时容量为 c_k 时背包获得的最大价值 v_k

步骤：

```
1.  p(0)←{(0,0)}
2.  P←p(0)                                  //初始化跳跃间断点集
```

```
3.  For i=1 To n Do
4.    q(i-1)←{(0,0)}
5.    For Each(c_k,v_k)∈p(i-1) Do
6.      If c_k+w_i≤c Then
7.        q(i-1)←q(i-1)∪{(c_k+w_i,v_k+v_i)}    //得到新的跳跃点集 q(i-1)
8.      End If
9.    End For
10.   p(i)←p(i-1)∪q(i-1)
11.   删除集合 p(i)中的受控跳跃点
12.   P←P∪p(i)                                //将当前跳跃间断点集加入背包价值的全部跳跃间断点集
13. End For
14. Return P
```

由于计算 $q(i-1)$ 和 $p(i)$ 的时间复杂度均为 $O(|p(i-1)|)$，而跳跃间断点的个数 $|p(i-1)|$ 不会超过 2^{i-1}，且有 $1 \leqslant i \leqslant n$，因此，计算跳跃间断点集的时间复杂度为 $O\left(\sum_{i=1}^{n} 2^{i-1}\right)=O(2^n)$，即算法 5.2 的时间复杂度为 $O(2^n)$。

对于例 5.1 中的 0-1 背包问题实例，使用算法 5.2 可得到：

① 初始 $p(0)=\{(0,0)\}$，$(w_1,v_1)=(2,6)$，那么 $q(0)=\{(2,6)\}$，$p(1)=\{(0,0),(2,6)\}$；

② $(w_2,v_2)=(2,3)$，那么 $q(1)=\{(2,3),(4,9)\}$，$p(2)=\{(0,0),(2,6),(4,9)\}$；

③ $(w_3,v_3)=(6,5)$，那么 $q(2)=\{(6,5),(8,11),(10,14)\}$，$p(3)=\{(0,0),(2,6),(4,9),(8,11),(10,14)\}$；

④ $(w_4,v_4)=(5,4)$，那么 $q(3)=\{(5,4),(7,10),(9,13)\}$，$p(4)=\{(0,0),(2,6),(4,9),(7,10),(8,11),(9,13),(10,14)\}$；

⑤ $(w_5,v_5)=(4,6)$，那么 $q(4)=\{(4,6),(6,12),(8,15)\}$，$p(5)=\{(0,0),(2,6),(4,9),(6,12),(8,15)\}$。

回溯跳跃间断点集，即可得到物品装备背包的最优策略。

然而，对于实际应用来说，动态规划算法的时间复杂度仍然过高，因此，可采用第 4 章的贪心法对其进行近似求解。首先计算每种物品的单位质量价值 v_i/w_i，然后按照单位质量价值对物品进行非升序排序，根据贪心选择策略优先选择当前单位质量价值最大的物品，直到背包不能再装入任何物品。使用贪心法近似求解 0-1 背包问题时，计算物品单位质量价值并排序的时间复杂度为 $O(n\log n)$，通过贪心选择将物品加入背包的时间复杂度为 $O(n)$，因此，算法的时间复杂度为 $O(n\log n)$，此时得到的解不一定为最优解。

针对前述例子，使用贪心法首先计算各个物品的单位质量价值{3，1.5，0.83，0.8，1.5}，按照单位质量价值对物品进行排序得到{物品 1，物品 2，物品 5，物品 3，物品 4}，首先装入物品 1，背包价值为 6，剩余空间为 8；接着装入物品 2，背包价值为 9，剩余空间为 6；接着装入物品 5，背包价值为 15，剩余空间为 2，此时无法再装入任何物品，总价值达到最大。

5.3　Python 程序示例

本节给出 Python 程序示例，实现 0-1 背包问题的动态规划算法(算法 5.1)。

程序示例 5.1

```
#计算价值矩阵 b
def dynamic_plan(n, c, w, v):
    b = [[-1 for j in range(c + 1)] for i in range(n + 1)]
    for j in range(c + 1):
        b[0][j] = 0
    for i in range(1, n + 1):
        for j in range(1, c + 1):
            b[i][j] = b[i - 1][j]
            if j >= w[i - 1] and b[i][j] < b[i - 1][j - w[i - 1]] + v[i - 1]:
                b[i][j] = b[i - 1][j - w[i - 1]] + v[i - 1]
    return b


#输出选择的物品
def need_goods(n, c, w, b):
    print('最大价值为:', b[n][c])
    x = [False for i in range(n)]
    j = c
    for i in range(1, n + 1):
        if b[i][j] > b[i - 1][j]:
            x[i - 1] = True
            j -= w[i - 1]
    print('选择的物品为:')
    for i in range(n):
        if x[i]:
            print(f'第 {i} 个,', end='')
    print('')


if __name__ == '__main__':
    n = 5
    c = 10
    w = [2, 2, 6, 5, 4]
    v = [6, 3, 5, 4, 6]
    b = dynamic_plan(n, c, w, v)
    print(b)
    need_goods(n, c, w, b)
```

运行结果：

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [-1, 0, 6, 6, 6, 6, 6, 6, 6, 6, 6], [-1, 0, 6, 6, 9, 9,
9, 9, 9, 9, 9], [-1, 0, 6, 6, 9, 9, 9, 9, 11, 11, 14], [-1, 0, 6, 6, 9, 9, 9, 10, 11, 13,
14], [-1, 0, 6, 6, 9, 9, 12, 12, 15, 15, 15]]
最大价值为：15
选择的物品为：
第 0 个,第 1 个,第 4 个
```

5.4 小　　结

动态规划法与贪心法都是将原问题分解为更小的子问题，并试图通过求解子问题产生一个全局最优解。不同的是，贪心法选择当前最优解，做出的每步贪心选择都无法改变，因为贪心策略并不保留上一步之前的最优解；动态规划的全局最优解中一定包含某个局部最优解，但不一定包含前一个局部最优解，因此需记录之前的所有最优解。

动态规划法将原问题分解为多个重叠子问题，先求得子问题的解，再合并这些子问题的解得到原问题的解，其优点在于能保存已解决子问题的解，计算过程中再找出需要的、已求出的子问题答案，这样可避免大量的重复计算。若采用递归算法求解这样的问题，子问题经过多次重复计算，计算次数呈指数性增长。动态规划的本质是以空间换时间，子问题只计算一次，相比递归算法能有效降低时间复杂度。针对具体问题，普通动态规划的空间复杂度也会发生维度爆炸问题，这时可使用贪心法等策略近似求解。

习　　题

1. 简述动态规划算法的基本思想、使用动态规划算法求解问题的基本步骤。举例说明动态规划算法适用的条件或场景。

2. 动态规划法和分治法有什么共同特点？这两种技术主要存在哪些不同点？

3. 对于如下 0-1 背包问题的实例，背包容量 $W=6$。

物　品	质　量	价　值
1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

(1) 给出使用算法 5.1 求解以上 0-1 背包问题的过程。

(2) 上面(1)中的实例有多少个不同的最优子集？

（3）一般来说，如何从算法 5.1 所生成的表中判断 0-1 背包问题的实例是否具有多个最优子集？

4. 找零问题以待找零金额 n 和各种面额 $d_1,d_2,\cdots,d_m$ 的数量无限的硬币，使找零硬币数最小。设计一个动态规划算法求解找零问题，得到金额 n 的找零硬币的最少个数或指出该问题无解。

5. 某国际航空公司在世界范围有 n 个国际机场。第 i 个国际机场到中心机场的距离为 $d_i(i=1,2,\cdots,n)$。从国际机场 j 到国际机场 i 的飞行费用为 $w(i,j)=s+(d_i-d_j)^2$，s 为地面加油费用。从任何国际机场飞往中心机场的飞机可以在任一国际机场加油后继续飞行。飞机加油问题要求确定从距中心机场最远的国际机场飞到中心机场的最少费用。设计一个动态规划算法求解飞机加油问题，并分析其时间复杂度。