数学建模期末报告-背包问题

吴天阳 2204210460 强基数学 002

1 背包问题

背包问题（Knapsack problem）是一种组合优化的 NP 完全问题. 问题基本表述为：给定一组物品，每种物品具有自己的体积与价值，在有限的体积内，如何选择，使得物品的总价值达到最大.

该问题在实际中具有非常广泛的应用，例如如何打包行李使得最大化行李价值且不超载、资源分配问题：从多个项目中具有时间或预算的限制，在相同时间或预算内达到最大的价值. 所以研究背包问题十分有价值，这里以经典的《背包九讲》对基础的三种背包算法进行学习.

注：下文中小数默认**向下取整**.

1.1 01 背包问题

问题 总共有 N 件物品和一个容量大小为 V 的背包，其中第 i 件物品的容量为 $c[i]$，价值为 $w[i]$. 求解在不超过背包容量的前提下最大化物品价值.

分析 该问题是最基础的背包问题，由于每个物品只能选择装与不装，即对应 0 与 1 的状态，所以也称为 01 背包问题.

定义状态数组：$dp[i][j]$ 表示前 i 件物品放入容量为 j 的背包可获得的最大价值. 则其状态转移方程为

$$dp[i][j] = \max\{dp[i-1][j], dp[i-1][j-c[i]]+w[i]\}$$

动态规划本质是考虑当前状态与之前状态的关系，通过之前已有的状态通过转移方程得到当前状态的值. 我们考虑当前第 i 个物品是否放入背包：如果不放，则问题转化为前 $i-1$ 个物品放入大小为 j 的背包可获得的最大价值；如果放，则问题转化为前 $i-1$ 个物品放入大小为 $j-c[i]$ 大小的背包可获得的最大价值加上当前物品的价值 $w[i]$. 于是，再对这两项取 max 即可得到上述状态转移方程.

优化 上述算法的时间复杂度与空间复杂度均为 $\mathcal{O}(VN)$，但是空间复杂度可以优化到 $\mathcal{O}(N)$. 首先分析上述算法的具体实现方法

```
for i=1 to N do
    for j=c[i] to V do
        dp[i][j] = max(dp[i-1][j], dp[i-1][j-c[i]]+w[i])
```

如果我们考虑反向推导第二维循环，那么我们就可以通过一维数组完成上述操作

```
for i=1 to N do
    for j=V to c[i] do
        dp[j] = max(dp[j], dp[j-c[i]]+w[i])
```

因为如果反向枚举，我们仍可以保证 $dp[j-c[i]]$ 就是原来的 $dp[i-1][j-c[i]]$，即前 $i-1$ 个物品的对应的状态.

由于 01 背包用途广泛，所以我们引入如下函数专门用于处理一件 01 背包中的物品

```
def ZeroOnePack(cost, weight)  # cost 为物品大小, weight 为物品的价值
    for v=V to cost do
        dp[v] = max(dp[v], dp[v-cost]+weight)
```

有了这个过程后，01 背包可以简写成如下形式

```
for i=1 to N do
    ZeroOnePack(c[i], w[i])
```

1.2 完全背包问题

问题 总共有 N 件物品和一个容量大小为 V 的背包，每种物品都可以无限次使用. 其中第 i 件物品的容量为 $c[i]$，价值为 $w[i]$. 求解在不超过背包容量的前提下最大化物品价值.

分析 该问题与 01 背包问题非常类似. 根据 01 背包的状态转移方程，我们可以类似定义状态数组：$dp[i][j]$ 表示前 i 件物品放入容量为 j 的背包可获得的最大价值，则其状态转移方程为

$$dp[i][j] = \max\{dp[i-1][j], dp[i-1][j-k\cdot c[i]] + k\cdot w[i] : k\cdot c[i] \leqslant j\} \tag{1.1}$$

当前状态 $dp[i][j]$ 可以通过选择 k 个第 i 种物品，于是可以从 $dp[i-1][j-k\cdot c[i]]$ 处进行转移. 这根据 01 背包的问题相同，类似求解思路可以得到时间复杂度为 $\mathcal{O}(V\sum\limits_{i=1}^{N}\dfrac{V}{c[i]})$.

优化 由于每件物品数量具有无穷多个，所以如果我们考虑第 i 件物品不从第 $i-1$ 个物品的状态进行转移，而是直接通过当前物品的上一个状态进行转移，即以下转移方程

$$dp[i][j] = \max\{dp[i-1][j], dp[i][j-c[i]] + w[i]\} \tag{1.2}$$

首先，理性分析下上述转移方程的含义，由于每个物品能够选择无限多次. 所以，当前第 $dp[i][j]$ 个物品的状态可以直接通过当前物品的 $j-c[i]$ 的容量大小进行转移，这就相当于可以重复选择当前物品多次.

下面我们来证明该转移方程的结果与 (1.1) 的结果相同，假设 (1.1) 式中取到 k 时有最大值，也即是

$$\begin{aligned}
&dp[i-1][j-k\cdot c[i]] > dp[i-1][j-(k+1)\cdot c[i]] + w[i] > dp[i][j-(k+1)j\cdot c[i]] \\
\Rightarrow\ &dp[i][j-k\cdot c[i]] = dp[i-1][j-k\cdot c[i]]
\end{aligned}$$

所以存在一种转移链使得

$$\begin{aligned}
dp[i][j] &= dp[i][j-c[i]] + w[i] = dp[i][j-2c[i]] + 2w[i] = \cdots \\
&= dp[i][j-k\cdot c[i]] + k\cdot w[i] = dp[i-1][j-k\cdot c[i]] + k\cdot w[i]
\end{aligned}$$

于是转移方程 (1.1) 与 (1.2) 完全等价，且复杂度为 $\mathcal{O}(VN)$，和 01 背包相同. 代码实现如下

```
for i=1 to N do
    for j=c[i] to V do
        dp[i][j] = max(dp[i-1][j], dp[i][j-c[i]]+w[i])
```

类似地可以编写专门处理一件完全背包中的物品

```
def CompletePack(i, cost=c[i], weight=w[i])  # 当前处理第 i 件物品
    for j=1 to V do
        dp[i][v] = max(dp[i-1][v], dp[i][v-c[i]]+w[i])
```

对于与 01 背包的区别，我们可以观察到仅有第二层循环的遍历顺序不同，转移方程上有微小变化.

其他优化技巧 一个直观的优化方法，假设物品 i 的容量大且价值低与另一个物品 j，也即 $c[i]\geqslant c[j]$ 且 $w[i]\leqslant w[j]$，则物品 i 一定不会被选中. 因为假如能够选中物品 i，则一定可以通过选物品 j 代替物品 i 使得总价值达到更大.

第二种优化方法，虽然不如第一种优化方法，但仍具有思考价值. 如果我们将一种物品 i，打包的思路合成一个物品，则一共能打包为 $\lfloor V/c[i]\rfloor$ 个物品，设 $k=1,2,\cdots,\lfloor V/c[i]\rfloor$，则第 k 个物品的容量为 $k\cdot c[i]$，价值为 $k\cdot w[i]$，则打包后的物品总数目为 $\sum\limits_{i=1}^N\dfrac{V}{c[i]}$. 于是可以将问题转化为求解 01 背包，但是这样的时间复杂度并没有优化，仍和转移方程 (1.1) 的时间复

进一步思考，如果一件物品的打包不是一个一个的打包，而是以 2 进制进行打包，也即 $k=1,2,\cdots,\lfloor\log V/c[i]\rfloor$，第 k 个的容量为 $2^k\cdot c[i]$，价值为 $2^k\cdot w[i]$，这样就打包出 $\sum\limits_{i=1}^N\log\dfrac{V}{c[i]}$ 个物品，因为可以通过是否选择一个二进制数来得到所有数字，所以和第一种方式结果相同. 再使用 01 背包方式求解，总复杂度为 $\mathcal{O}(V\cdot\sum\limits_{i=1}^N\log\dfrac{V}{c[i]})$.

1.3 多重背包问题

问题 总共有 N 件物品和一个容量大小为 V 的背包. 第 i 种物品最多有 $a[i]$ 件可以选取，第 i 种物品的容量均为 $c[i]$，价值为 $w[i]$. 求解在不超过背包容量的前提下最大化物品价值.

分析 该问题与完全背包问题非常类似. 根据完全背包的思路 k 会被两个上界所限制住

$$dp[i][j]=\max\left\{dp[i-1][j],dp[i-1][j-k\cdot c[i]]+k\cdot w[i]:k\leqslant\min\left\{a[i],\frac{V}{c[i]}\right\}\right\} \tag{1.3}$$

总时间复杂度为 $\mathcal{O}(V\cdot\sum\limits_{i=1}^N a[i])$.

优化 我们可以通过类似完全背包第二种优化方法，将物品进行二进制合并，然后通过二进制数合并得到 $1,\cdots,a[i]$ 每个数的组合. 假设打包为 $k+1$ 组，则前 $k-1$ 个包大小分别为 $1,2^1,2^2,\cdots,2^{k-1}$，则总计为 $\sum\limits_{i=0}^{k-1}2^i=2^k-1$，于是最后一个包大小是 $a[i]-2^k+1$，且满足 k 是使得 $a[i]-2^k+1>0$ 的最大值，即 $2^k\leqslant a[i]\Rightarrow k=\log a[i]$. 进而转化为 01 背包问题，从而总时间复杂度变为 $\mathcal{O}(V\cdot\sum\limits_{i=1}^{N}\log a[i])$. 多重背包问题中处理一个物品的代码如下：

```
def MultiplePack(i, cost=c[i], weight=w[i], amount=a[i])  # 当前处理第 i 件物品
    if cost * amount >= V do
        CompletePack(i, cost, weight)  # 当 c[i]*a[i] 大于当前包容量，则等价于完全背包问题
    k = 1  # 当前打包的大小
    while k < amount do
        ZeroOnePack(k * cost, k * weight)
        amount = amount - k
        k = k * 2
    ZeroOnePack(amount * cost, amount * weight)
```

单调队列优化 利用单调队列优化，可以将时间复杂度降低到 $\mathcal{O}(VN)$. 此算法是基于动态规划 (1.3) 式的优化算法，我们观察对于容量为 j，则当前容量只会和 $j-c[i],j-2c[i],\cdots,j-a[i]\cdot c[i]$ 相关，也就是 $\bar{j}$ 在模 $c[i]$ 下的等价类，取最近的 $a[i]$ 项. 所以利用这一性质，我们可以将整个背包容量用模 $c[i]$ 的等价类进行划分，在每个等价类中分别计算转移方程的最大值.

考虑如何在 $\mathcal{O}(1)$ 的复杂度下获得当前等价类中的最大 dp 值. 在一个等价类中，我们从左至右滑动一个大小为 $a[i]$ 的滑动窗口，并用单增的单调队列维护当前滑动窗口的最大值，加入当前滑动窗口最右侧端点值为 j，则 $dp[j]$ 为当前单调队列队首对应的 dp 值加上当前物品的价值.

单调队列具体实现方法：考虑用单调队列维护一个对应状态值单调递增的**下标**队列，保证队首元素处于当前滑动窗口内，当前元素加入队列时，保证队尾的状态值大于当前加入的状态值. 具体实现请见代码：

```
def MultiplePackQueue(i, cost=c[i], weight=w[i], amount=a[i])  # 当前处理第 i 件物品
    initialize queue  # 创建一个空的队列
    for j=0 to cost do  # 枚举当前等价类的开头
        for k=j to V by cost do  # 枚举等价类中的元素 k
            value = dp[i-1][k] - (k-j) / cost * weight  # 定义当前所需维护的权重
            while (queue not empty) and (queue.front.position < k - amount * cost) do  # 若首元素已经超出了当前滑动窗口大小，弹出
```

```
                pop queue.front
            while (queue not empty) and (queue.back.value < value) do  # 若队尾元素的权重小于当前权重，弹出
                pop queue.back
            if queue not empty do
                dp[i][k] = max(dp[i-1][k], dp[i-1][queue.head.position] + (k-queue.head.position) / cost * weight)
            else do
                dp[i][k] = dp[i-1][k]
            push (position=k, value=value) into queue  # 将当前新节点加入单调队列
```

上述过程虽然是三层 for 循环，但是第二、三层 for 循环是对 $1, \cdots, N$ 的一个划分，所以总复杂度仍为 $\mathcal{O}(VN)$.

总结

通过这三种背包的学习，基本掌握了解决背包问题的基本思想，主要为通过设计动态规划数组与状态转移方程，通过数学建模的方法将原问题转化为数学中离散的最优化问题. 在 01 背包的基础上，通过可以多次选取物品，衍生出完全背包（可无限次选取）与多重背包（每种物品数目有限）两种背包问题，一种有效的通用方法为二进制打包，即将多个物品打包以 2 的幂次打包成为一个物品，再利用 01 背包算法求解. 而他们都分别具有更为高效的算法，完全背包通过调整转移方程进行优化，多重背包则是通过划分为子问题、通过滑动窗口结合单调队列维护最大值进行求解.

通过这三种问题的研究，对数学建模有了更深刻的认识，通过将实际问题转化为数学最优化问题，再从一个问题推广到求解其他类似问题，更进一步考虑不同优化方法对算法进行改进. 这提示我们要不断思考与改进现有的模型，并学会在已有的基础上进行创新，从而得到更高效的算法.