背包问题九讲2.0

崔添翼 (Tianyi Cui, a.k.a. dd_engi)

September 15, 2011

Contents

1 01背包问题

1.1 题目

有N件物品和一个容量为V的背包。放入第i件物品耗费的空间是C_i，得到的价值是W_i。求解将哪些物品装入背包可使价值总和最大。

1.2 基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即$F[i,v]$表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。则其状态转移方程便是：

$$F[i,v]=\max\{F[i-1,v],F[i-1,v-C_i]+W_i\}$$

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只和前$i-1$件物品相关的问题。如果不放第i件物品，那么问题就转化

为“前$i-1$件物品放入容量为v的背包中”，价值为$F[i-1,v]$；如果放第i件物品，那么问题就转化为“前$i-1$件物品放入剩下的容量为$v-C_i$的背包中”，此时能获得的最大价值就是$F[i-1,v-C_i]$再加上通过放入第i件物品获得的价值W_i。

伪代码如下：

$F[0,0..V]=0$
for i = 1 to N
  for v = C_i to V
    $F[i,v]$ = max$\{F[i-1,v],F[i-1,v-C_i]+W_i\}$

1.3 优化空间复杂度

以上方法的时间和空间复杂度均为$O(VN)$，其中时间复杂度应该已经不能再优化了，但空间复杂度却可以优化到$O(V)$。

先考虑上面讲的基本思路如何实现，肯定是有一个主循环$i=1..N$，每次算出来二维数组$F[i,0..V]$的所有值。那么，如果只用一个数组$F[0..V]$，能不能保证第i次循环结束后$F[v]$中表示的就是我们定义的状态$F[i,v]$呢？$F[i,v]$是由$F[i-1,v]$和$F[i-1,v-C_i]$两个子问题递推而来，能否保证在推$F[i,v]$时（也即在第i次主循环中推$F[v]$时）能够取用$F[i-1,v]$和$F[i-1,v-C_i]$的值呢？事实上，这要求在每次主循环中我们以$v=V..0$的递减顺序计算$F[v]$，这样才能保证推$F[v]$时$F[v-C_i]$保存的是状态$F[i-1,v-C_i]$的值。伪代码如下：

$F[0..V]=0$
for i = 1 to N
  for v = V to C_i
    $F[v]$ = max$\{F[v],F[v-C_i]+W_i\}$

其中的$F[v]=\max\{F[v],F[v-C_i]+W_i\}$一句，恰就对应于我们原来的转移方程，因为现在的$F[v-C_i]$就相当于原来的$F[i-1,v-C_i]$。如果将$v$的循环顺序从上面的逆序改成顺序的话，那么则成了$F[i,v]$由$F[i,v-C_i]$推导得到，与本题意不符。

事实上，使用一维数组解01背包的程序在后面会被多次用到，所以这里抽象出一个处理一件01背包中的物品过程，以后的代码中直接调用不加说明。

def ZeroOnePack(F,C,W)
  for v = V to C
    $F[v]$ = max$\{F[v],f[v-C]+W\}$

有了这个过程以后，01背包问题的伪代码就可以这样写：

for i = 1 to N
  ZeroOnePack(F,C_i,W_i)

1.4 初始化的细节问题

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了$F[0]$为0，其它$F[1..V]$均设为$-\infty$，这样就可以保证最终得到的$F[V]$是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将$F[0..V]$全部设为0。

这是为什么呢？可以这样理解：初始化的F数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为0的背包可以在什么也不装且价值为0的情况下被“恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，应该被赋值为-∞了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为0，所以初始时状态的值也就全部为0了。

这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

1.5 一个常数优化

上面伪代码中的

```
for i = 1 to N
  for v = V to C_i
```

中第二重循环的下限可以改进。它可以被优化为

```
for i = 1 to N
  for v = V to max{V − Σ_i^N W_i, C_i}
```

这个优化之所以成立的原因请读者自己思考。（提示：使用二维的转移方程思考较易。）

1.6 小结

01背包问题是最基本的背包问题，它包含了背包问题中设计状态、方程的最基本思想。另外，别的类型的背包问题往往也可以转换成01背包问题求解。故一定要仔细体会上面基本思路的得出方法，状态转移方程的意义，以及空间复杂度怎样被优化。

2 完全背包问题

2.1 题目

有N种物品和一个容量为V的背包，每种物品都有无限件可用。放入第i种物品的耗费的空间是C_i，得到的价值是W_i。求解：将哪些物品装入背包，可使这些物品的耗费的空间总和不超过背包容量，且价值总和最大。

2.2 基本思路

这个问题非常类似于01背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取0件、取1件、取2件……直至取$\lfloor V/C_i \rfloor$件等很多种。

如果仍然按照解01背包时的思路，令$F[i,v]$表示前i种物品恰放入一个容量为v的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：

$$F[i,v] = \max\{F[i-1, v-kC_i] + kW_i \mid 0 \le kC_i \le v\}$$

这跟01背包问题一样有$O(VN)$个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态$F[i,v]$的时间是$O(\frac{v}{C_i})$，总的复杂度可以认为是$O(NV\Sigma\frac{V}{C_i})$，是比较大的。

将01背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明01背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是要试图改进这个复杂度。

2.3 一个简单有效的优化

完全背包问题有一个很简单有效的优化，是这样的：若两件物品i、j满足$C_i \leq C_j$且$W_i \geq W_j$，则将可以将物品j直接去掉，不用考虑。

这个优化的正确性是显然的：任何情况下都可将价值小耗费高的j换成物美价廉的i，得到的方案至少不会更差。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

这个优化可以简单的$O(N^2)$地实现，一般都可以承受。另外，针对背包问题而言，比较不错的一种方法是：首先将费用大于V的物品去掉，然后使用类似计数排序的做法，计算出费用相同的物品中价值最高的是哪个，可以$O(V+N)$地完成这个优化。这个不太重要的过程就不给出伪代码了，希望你能独立思考写出伪代码或程序。

2.4 转化为01背包问题求解

01背包问题是最基本的背包问题，我们可以考虑把完全背包问题转化为01背包问题来解。

最简单的想法是，考虑到第i种物品最多选$\left\lfloor\frac{V}{C_i}\right\rfloor$件，于是可以把第i种物品转化为$\left\lfloor\frac{V}{C_i}\right\rfloor$件费用及价值均不变的物品，然后求解这个01背包问题。这样的做法完全没有改进时间复杂度，但这种方法也指明了将完全背包问题转化为01背包问题的思路：将一种物品拆成多件只能选0件或1件的01背包中的物品。

更高效的转化方法是：把第i种物品拆成费用为$C_i 2^k$、价值为$W_i 2^k$的若干件物品，其中k取遍满足$C_i 2^k \leq V$的非负整数。

这是二进制的思想。因为，不管最优策略选几件第i种物品，其件数写成二进制后，总可以表示成若干个2^k件物品的和。这样一来就把每种物品拆成$O(\log\frac{V}{C_i})$件物品，是一个很大的改进。

2.5 $O(VN)$的算法

这个算法使用一维数组，先看伪代码：

```
F[0..V] = 0
for i = 1 to N
  for v = C_i to V
    F[v] = max{F[v], F[v - C_i] + W_i}
```

你会发现，这个伪代码与01背包问题的伪代码只有v的循环次序不同而已。

为什么这个算法就可行呢？首先想想为什么01背包中要按照v递减的次序来循环。让v递减是为了保证第i次循环中的状态$F[i,v]$是由状态$F[i-1,v-C_i]$递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果$F[i-$

$1, v - C_i]$。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果$F[i, v - C_i]$，所以就可以并且必须采用v递增的顺序循环。这就是这个简单的程序为何成立的道理。

值得一提的是，上面的伪代码中两层for循环的次序可以颠倒。这个结论有可能会带来算法时间常数上的优化。

这个算法也可以由另外的思路得出。例如，将基本思路中求解$F[i, v - C_i]$的状态转移方程显式地写出来，代入原方程中，会发现该方程可以等价地变形成这种形式：

$$F[i, v] = \max(F[i - 1, v], F[i, v - C_i] + W_i)$$

将这个方程用一维数组实现，便得到了上面的伪代码。

最后抽象出处理一件完全背包类物品的过程伪代码：

```
def CompletePack(F, C, W)
  for v = C to V
    F[v] = max{F[v], f[v − C] + W}
```

2.6 总结

完全背包问题也是一个相当基础的背包问题，它有两个状态转移方程。希望你能够对这两个状态转移方程都仔细地体会，不仅记住，也要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。

事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。