**算法分析与设计实验报告**

**采用动态规划方法实现0-1背包问题**

姓 名： 于 欣 悦

专 业： 软件工程

学 号： 1120191350

实 验 日 期：2019年12月19日

1. **实验目的**

1、编程实现采用动态规划方法求解0-1背包问题的算法。

2、问题描述：给定*n*种物品和一个容量为*M*的背包，第*i*个物品的重量为*wi*，效益值为*pi*，假设该问题的解向量为(*x*1, *x*2, … , *xn*)，其中，若*xi*=1表示第*i*个物品放在包中；若*xi*=0表示第*i*个物品没有被选中，*i*=1, 2, … , *n*。要求依据动态规划思想构造一个装包算法，使得背包的总效益值达到最大，并加以实现。

极大化函数：达到极大。

约束条件：≤ *M*。*xi*=0 或*xi*=1。

3、假设*n*=20，M小于20个物品的总重量。随机生成(*wi*, *pi*)，每个*wi*和*pi*均为正整数，*i*=1, 2, … , *n*。输入数据用文件保存，以便打印输出。

4、输出生成的输入数据及根据输入数据所得到的解，并说明所得到的解是否为最优解。

5、根据实验结果，撰写实验报告。

1. **实验内容**

动态规划是用空间换时间的一种方法的抽象。其关键是发现子问题和记录其结果。然后利用这些结果减轻运算量。

1、基本思路  
 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牵扯前i-1件物品的问题。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]。

2、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。

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）能够得到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的顺序推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：

for i=1..N

for v=V..0

f[v]=max{f[v],f[v-c[i]]+w[i]};

其中的f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}，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]]推知。

过程ZeroOnePack，表示处理一件01背包中的物品，两个参数cost、weight分别表明这件物品的费用和价值。

procedure ZeroOnePack(cost,weight)

for v=V..cost

f[v]=max{f[v],f[v-cost]+weight}

注意这个过程里的处理与前面给出的伪代码有所不同。前面的示例程序写成v=V..0是为了在程序中体现每个状态都按照方程求解了，避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了，就可以加入优化。费用为cost的物品不会影响状态f[0..cost-1]，这是显然的。

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..N

ZeroOnePack(c[i],w[i]);

3、常数优化

前面的伪代码中有 for v=V..1，可以将这个循环的下限进行改进。

由于只需要最后f[v]的值，倒推前一个物品，其实只要知道f[v-w[n]]即可。以此类推，对以第j个背包，其实只需要知道到f[v-sum{w[j..n]}]即可，即代码中的

for i=1..N

for v=V..0

可以改成

for i=1..n

bound=max{V-sum{w[i..n]},c[i]}

for v=V..bound

这对于V比较大时是有用的。

1. **程序代码**

#include<stdio.h>

#include <stdlib.h>

int c[10][100];

int knapsack(int m,int n)

{

int i,j,w[10],p[10];

for(i=1;i<n+1;i++)

scanf("\n%d,%d",&w[i],&p[i]);

for(i=0;i<10;i++)

for(j=0;j<100;j++)

c[i][j]=0;

for(i=1;i<n+1;i++)

for(j=1;j<m+1;j++)

{

if(w[i]<=j)

{

if(p[i]+c[i-1][j-w[i]]>c[i-1][j])

c[i][j]=p[i]+c[i-1][j-w[i]];

else

c[i][j]=c[i-1][j];

}

else c[i][j]=c[i-1][j];

}

return(c[n][m]);

}

int main()

{

int m,n;int i,j;

scanf("%d,%d",&m,&n);

printf("Input each one:\n");

printf("%d",knapsack(m,n));

printf("\n");

for(i=0;i<10;i++)

for(j=0;j<15;j++)

{

printf("%d ",c[i][j]);

if(j==14)printf("\n");

}

system("pause");

}

1. **实验结果**
2. 测试数据如下：

10,3

3,4

4,5

5,6

1. 程序结果输出图如图4.1所示：

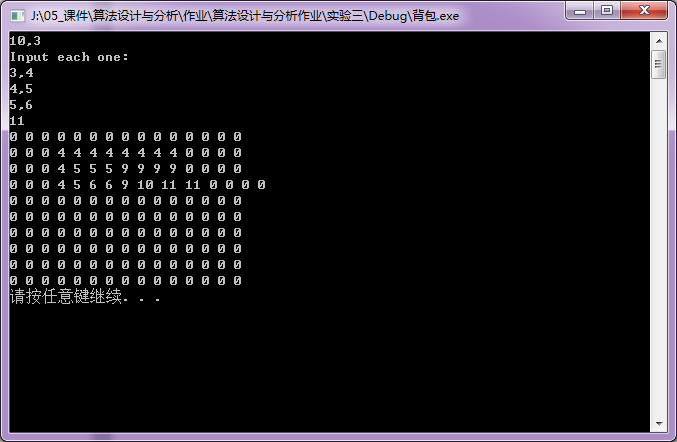


图4.1程序结果输出图

1. **实验分析及总结**

首先从最优化理论分析可知，该算法存在更优解，因此该算法不能得出最优解。

0-1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0-1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，以及最后怎样优化的空间复杂度。