**多维背包问题算法设计与分析**

**问题描述**

MKP和单约束背包问题不同在于每个物品消耗的资源并不是单一的。在单约束背包问题中，物品只消耗一个资源属性，即重量，物品的装入只受重量的约束，而在多约束背包问题中，物品有多个消耗属性，比如重量，体积等等，物品的装入就变得更加复杂。现要把n个物品装进一个大小为C的背包，每个物品的价值是Vi ,它们有多种属性，这个对每种属性都会有约束。那么这个背包可以装载物品的最大价值是多少？

可以把它形式化描述为：

Max ,

∈{0 , 1} , i=0 , 1 , …, n -1

,

在上式中，=0表示不装入背包，=1表示装入背包，表示物品属性需求

**算法设计与分析**

要使用回溯算法解决MKP，要明确地定义好问题的解空间，问题的解空间至少应该包含问题的一个最优解。对于有n种可选择物品的多维背包问题，它的解空间由长度为n的多维向量组成。回溯就相当于穷举加上剪枝，以深度优先方式搜索解空间，从根结点出发，让根结点成为活结点，同时成为当前的扩展结点。在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为新的活结点，并成为当前扩展结点。如果在当前的扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点。此时，应往回移动(回溯)至最近的一个活结点处，并使这个活结点成为当前的扩展结点。回溯法以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已无活结点时为止。

**算法复杂性**

在解决多维背包问题时算法复杂性是O（nm）, 时间复杂度为O（nm)

**算法描述（C语言）**

核心代码如下：

int BackTrack(int a) //回溯函数

{

int i;

//到达叶子节点

if (a >= n) {

if (V2 >= V1) {

V1 = V2;

printf("V1=%0.2f\n",V1); //输出当前的最优价值

for (i = 0; i < n; i++) {

printf("%d ", X[i]); //输出当前的最优选择

}

printf("\n");

for (i = 0; i < n; i++) { //保存当前路径的物品选取情况

X1[i] = X[i];

}

}

return 0;

}

//进入左子树

if (prun\_function(a) ) { //该物品可以装入当前的背包

for (i = 0; i < m; i++) {

r1[i] =r1[i] + r[i][a]; //把该物品的属性项加到原本背包的属性中

}

V2 =V2 + V[a]; //价值也要加到背包的物品总价值中

X[a] = 1; //1表示装入背包

BackTrack(a + 1); // 回溯

for (i = 0; i < m; i++) { //走不通则把属性还原

r1[i] =r1[i] - r[i][a];

}

V2 =V2 - V[a]; //价值也要还原

X[a] = 0; //0表示不装入背包

}

//进入右子树

BackTrack(a + 1);

return 0;

}

int prun\_function(int i){ //声明一个约束函数

int z;

for (z = 0; z < m; z++) {

if (C[z] >= r1[z] + r[z][i])

continue;

else

return 0; //该物品已经无法装入当前的背包

}

return 1;

}

**执行代码**

**输入：**

1、物品个数n，属性m，最优价值V

2、每个物品的价值

3、每个物品的各个属性

4、背包对每种属性的约束

**输出：**

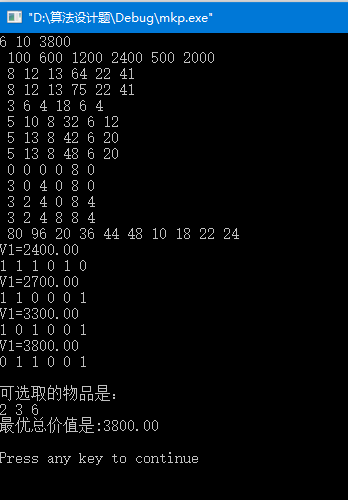
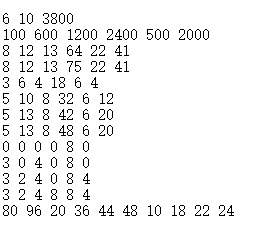
背包的物品选取情况

背包装入物品的最优总价值

例如：（这里只测试了三组数据）

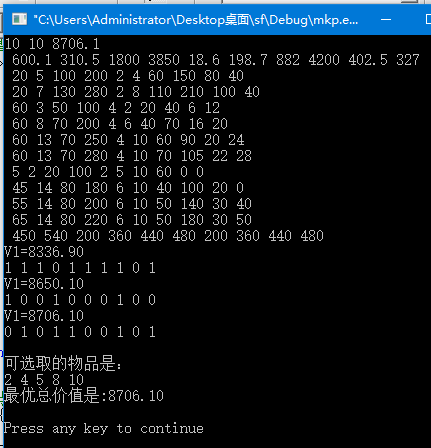
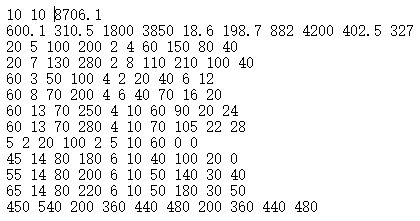
第一组数据

输入： 输出：



第二组数据

输入： 输出：



第三组数据

输入： 输出：

