实验5、《算法综合实验》

**一、实验目的**

1. 理解和复习所学各种算法的概念

2. 掌握和复习所学各种算法的基本要素

3. 掌握各种算法的优点和区别

4. 通过应用范例掌握选择最佳算法的设计技巧与策略

**二、实验内容**

1. 使用贪心算法、回溯法、分支限界法解决0-1背包问题；

2. 通过上机实验进行算法实现；

3. 保存和打印出程序的运行结果，并结合程序进行分析，上交实验报告。

**三、算法思想分析**

1.贪心算法理论上只能解决满足贪心选择性质的问题，而0-1背包并不满足该性质，所以并不能保证能够找到最优解法，只能找到最接近的解，当然如果运气好，也是可以找到最优解的。利用按重量从小到大、按价值从大到小、按价值/重量从大到小三种方式通过贪心算法求得每种方式的最终结果，并比较三种方式的最大价值取最大的那个，即为贪心算法获得的最优解。

2.回溯法解决0-1背包问题的解空间为子集树，利用回溯法的基本代码模版即可，其中左子树为约束条件，即背包能否装下该物品，右子树为限界条件，即当前物品不放入背包，剩余物品是否有可能创造比当前最大价值更大的价值，如果可以则进入右子树，反之，则直接剪去右子树。

3.0-1背包的解空间为子集树，分支界限法是采用广度优先搜索，每次选取队列的最前面的结点为活结点。

1）算法从根结点A即标记结点开始，初始时活结点队列为空，A入队列。

2）A为活结点，A的儿子结点B、C为可行结点。将B、C加入队列，舍弃A。此时队列元素为C-B；

3）B为活结点，B的儿子结点D、E,而D为不可行结点。将E入队列，舍弃B。此时队列元素为E-C；

4)循环以上步骤

按照以上方式扩展到叶节点。

**四、实验过程分析**

1.贪心算法的思路很简单即为一直循环下去，直至不满足指定条件。用于解决0-1背包问题时需要考虑多种放入方式，因为不管哪种方式都不能百分百会得到最优解，只能取多种放入方式中的最优解作为问题的最优解。

这道题目的收获在于贪心算法对于不能保证获得最优解的情况下，如何获得最接近的解，比如0-1背包问题则是采用多种放入方式再进行比较取最优解。

2.回溯法解决0-1背包问题的收获在于限界条件的运用，对于解空间为子集树的问题也有了进一步的了解。

3.从分支界限法解决0-1背包问题中，我更加懂得了队列的运用及标记结点的优势，本题中标记结点不仅代表着深度，而且保证队列不为空，当为空时循环结束。另外此题中的限界函数只有在最后的深度才发挥它的作用，在此之前一直没有到达底层， bestValue的数据一直为初始化数据0。

不过，还有一点不足就是没能实现背包获得最优价值时，所有物品的放入情况。

**五、算法源代码及用户屏幕**

1.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

贪心算法解决0-1背包问题

codeblocks C++

2018.11.6

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

//\*

//测试数据一，最大价值50,放置方式0,1，1

const int n = 3; //物品数量

int c = 30; //背包大小

int w[n] = {16, 15, 15}; //背包物品的重量

int v[n] = {45, 25, 25}; //背包物品的价值

//\*/

/\*

//测试数据二，最大价值15，放置方式1,1,0,0,1

const int n = 5; //物品数量

int c = 10; //背包大小

int w[n] = {2, 2, 4, 6, 5}; //背包物品的重量

int v[n] = {6, 3, 6, 5, 4}; //背包物品的价值

\*/

int value[3][n+1] = {0};

//二维数组记录三种放入方式的总价值和是否放入

//第一列为总价值，其余各列依次表示该物品是否放入

void weightG(); //按重量从轻到重重新定义物品放入顺序并求解

void valueG(); //按价值从大到小重新定义物品放入顺序并求解

void valueWeightG(); //按价值/重量重大到小重新定义物品放入顺序并求解

//贪心算法放入物品,number表示是哪种放入方式，indexArray[]表示加入物品的先后顺序

void greedyArrange(int number, int indexArray[]);

int main()

{

//按重量从轻到重定义物品放入顺序并求解

weightG();

//按价值从大到小定义物品放入顺序并求解

valueG();

//按价值/重量从大到小定义物品放入顺序并求解

valueWeightG();

//输出原始数据

cout<<"Greedy algorithm"<<endl;

cout<<"Things number : "<<n<<endl;

cout<<"Bag size : "<<c<<endl;

cout<<"Things weight : ";

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

cout<<w[i]<<" ";

}

cout<<endl;

cout<<"Things value : ";

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

cout<<v[i]<<" ";

}

cout<<endl<<endl;

//输出各种放入方式结果

//按重量从小到大

cout<<"Value(weight) : "<<value[0][0]<<endl;

cout<<"Arrange : ";

for(int j=1; j<n+1; j++){

cout<<value[0][j]<<" ";

}

cout<<endl;

//按价值从大到小

cout<<"Value(value) : "<<value[1][0]<<endl;

cout<<"Arrange : ";

for(int j=1; j<n+1; j++){

cout<<value[1][j]<<" ";

}

cout<<endl;

//按价值/重量从大到小

cout<<"Value(valueWeight) : "<<value[2][0]<<endl;

cout<<"Arrange : ";

for(int j=1; j<n+1; j++){

cout<<value[2][j]<<" ";

}

cout<<endl<<endl;

//确定最大价值的放入方式

int maxValueNumber = 0; //数字记录哪种放入方式价值最大

int maxValue = value[0][0];

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

if(maxValue < value[i][0]){

maxValueNumber = i;

maxValue = value[i][0];

}

}

//输出最大价值放入方式及价值

cout<<"MaxValue : "<<value[maxValueNumber][0]<<endl;

cout<<"Best arrange : ";

for(int j=1; j<n+1; j++){

cout<<value[maxValueNumber][j]<<" ";

}

cout<<endl;

return 0;

}

//按重量从轻到重重新定义物品放入顺序

void weightG(){

int tempSort[n]; //临时保存重量

int a[n]; //记录下标

//初始化数组

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

tempSort[i] = w[i];

a[i] = i;

}

//冒泡排序，按重量从小到大排序

for(int i=0; i<n-1; i++){

for(int j=0; j<n-1; j++){

if(tempSort[j] > tempSort[j+1]){

int temp = tempSort[j];

tempSort[j] = tempSort[j+1];

tempSort[j+1] = temp;

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

greedyArrange(0, a);

}

//按价值从大到小重新定义物品放入顺序并求解

void valueG(){

int tempSort[n]; //临时保存价值

int a[n]; //记录下标

//初始化数组

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

tempSort[i] = v[i];

a[i] = i;

}

//冒泡排序，按价值从大到小排序

for(int i=0; i<n-1; i++){

for(int j=0; j<n-1; j++){

if(tempSort[j] < tempSort[j+1]){

int temp = tempSort[j];

tempSort[j] = tempSort[j+1];

tempSort[j+1] = temp;

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

greedyArrange(1, a);

}

//按价值/重量重大到小重新定义物品放入顺序并求解

void valueWeightG(){

double tempSort[n]; //临时保存价值/重量

int a[n]; //记录下标

//初始化数组

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

tempSort[i] = (double)v[i]/(double)w[i];

a[i] = i;

}

//冒泡排序，按价值/重量从大到小排序

for(int i=0; i<n-1; i++){

for(int j=0; j<n-1; j++){

if(tempSort[j] < tempSort[j+1]){

int temp = tempSort[j];

tempSort[j] = tempSort[j+1];

tempSort[j+1] = temp;

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

greedyArrange(2, a);

}

//贪心算法放入物品

void greedyArrange(int number, int indexArray[]){

int cBag = c;

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

if(cBag >= w[indexArray[i]]){

value[number][indexArray[i]+1] = 1;

cBag -= w[indexArray[i]];

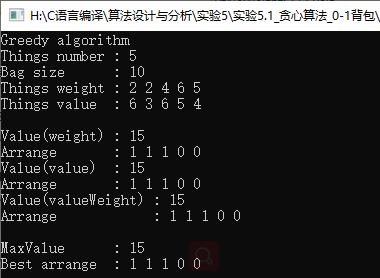
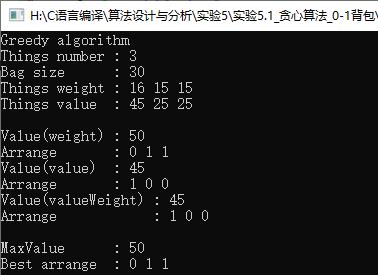
value[number][0] += v[indexArray[i]];

}

}

}

（2）用户屏幕



2.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

回溯法解决0-1背包问题

codeblocks C++

2018.11.5

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

void backTrack(int deep, int n, int c, int w[], int v[]); //deep为树的深度，n为物品数量，c为背包大小，w[]为物品重量，v[]为物品价值

int bound(int deep, int n, int c, int w[], int v[]); //限界函数

int arrangCurrent[100] = {0}; //当前放置方式

int bestArrange[100] = {0}; //最佳放置方式

int valueCurrent = 0;

int maxValue = 0;

int main()

{

//\*

//测试数据一，最大价值50,放置方式0,1，1

int n = 3; //物品数量

int c = 30; //背包大小

int w[n] = {16, 15, 15}; //背包物品的重量

int v[n] = {45, 25, 25}; //背包物品的价值

//\*/

/\*

//测试数据二，最大价值15，放置方式1,1,0,0,1

int n = 5; //物品数量

int c = 10; //背包大小

int w[n] = {2, 2, 4, 6, 5}; //背包物品的重量

int v[n] = {6, 3, 6, 5, 4}; //背包物品的价值

\*/

//初始化都不放置

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

arrangCurrent[i] = 0;

bestArrange[i] = 0;

}

backTrack(0, n, c, w, v);

cout<<"Things number : "<<n<<endl;

cout<<"Bag size : "<<c<<endl;

cout<<"Things weight : ";

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

cout<<w[i]<<" ";

}

cout<<endl;

cout<<"Things value : ";

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

cout<<v[i]<<" ";

}

cout<<endl<<endl;

cout<<"MaxValue : "<<maxValue<<endl;

cout<<"Best arrange : ";

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

cout<<bestArrange[i]<<" ";

}

cout<<endl;

return 0;

}

void backTrack(int deep, int n, int c, int w[], int v[]){

if(deep >= n){

if(valueCurrent > maxValue){

//当前价值大于最大价值，更新放置方式及最大价值

maxValue = valueCurrent;

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

bestArrange[i] = arrangCurrent[i];

}

}

}

else{

if(w[deep] <= c){ //搜索左子树，约束条件

arrangCurrent[deep] = 1;

c -= w[deep];

valueCurrent += v[deep];

backTrack(deep+1, n, c, w, v);

arrangCurrent[deep] = 0;

c += w[deep];

valueCurrent -= v[deep];

}

if(bound(deep+1, n, c, w, v) > maxValue){ //搜素右子树

arrangCurrent[deep] = 0;

backTrack(deep+1, n, c, w, v);

}

}

}

int bound(int deep, int n, int c, int w[], int v[]){

int left = c; //剩余重量

int valueTemp = valueCurrent; //当前价值

while(deep<n && w[deep]<=left){

left -= w[deep];

valueTemp += v[deep];

deep++;

}

if(deep<n){

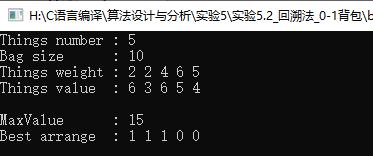
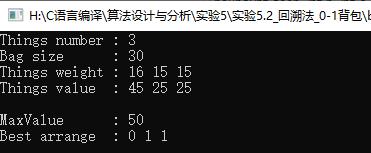
valueTemp += left\*(v[deep]/w[deep]);

}

return valueTemp;

}

（2）用户屏幕



3.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

队列式分支界限法解决0-1背包问题

codeblocks C++

2018.11.6

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <queue>

using namespace std;

//每个结点包含当前重量、价值、深度、是否放入

typedef struct treeNode{

int weight;

int value;

int treeDeep;

int flag;

}treeNode;

//创建队列，存放活结点数据

queue<struct treeNode> liveNodeQueue;

//\*

//测试数据一，最大价值50,放置方式0,1，1

const int n = 3; //物品数量

int c = 30; //背包大小

int w[n] = {16, 15, 15}; //背包物品的重量

int v[n] = {45, 25, 25}; //背包物品的价值

//\*/

/\*

//测试数据二，最大价值15，放置方式1,1,0,0,1

const int n = 5; //物品数量

int c = 10; //背包大小

int w[n] = {2, 2, 4, 6, 5}; //背包物品的重量

int v[n] = {6, 3, 6, 5, 4}; //背包物品的价值

//\*/

int bestValue = 0;

//活结点则加入到队列中

void joinQuenen(int weight, int value, int deep, int flag);

void knap(); //广度搜索

int bound(int deep, int cBag, int value); //限界条件

int main()

{

knap();

//输出背包物品相关信息

cout<<"Branch-bound Algorithm"<<endl;

cout<<"Things number : "<<n<<endl;

cout<<"Bag size : "<<c<<endl;

cout<<"Things weight : ";

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

cout<<w[i]<<" ";

}

cout<<endl;

cout<<"Things value : ";

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

cout<<v[i]<<" ";

}

cout<<endl<<endl;

//输出结果

cout<<"MaxValue : "<<bestValue<<endl;

return 0;

}

//活结点则加入到队列中

void joinQuenen(int weight, int value, int deep, int flag){

treeNode node;

node.weight = weight;

node.value = value;

node.treeDeep = deep;

node.flag = flag;

if(deep == n){

//到底最底层,不加入队列

if(node.value > bestValue){

bestValue = node.value;

}

}

else{

//不是最底层，加入队列

liveNodeQueue.push(node);

}

}

//广度搜索

void knap(){

int deep = 1; //表示广度搜索的深度

treeNode tag, liveNode; //tag结点表示为标记结点，表示是否在同一深度，livenode结点表示当前的扩展结点

//队列的起始结点

tag.weight = -1; //用于区别其他结点，判断是否进入下一深度

tag.value = 0;

tag.treeDeep = 0; //表示广度搜索的深度

tag.flag = 0;

//初始化活结点

liveNode.weight = 0;

liveNode.value = 0;

liveNode.treeDeep = 1;

liveNode.flag = 0;

liveNodeQueue.push(tag);

while(1){

//约束条件，判断左子树是否可行

if(liveNode.weight + w[deep-1] <= c){

joinQuenen(liveNode.weight+w[deep-1], liveNode.value+v[deep-1], deep, 1);

}

//限界条件，判断右子树是否可行

if(bound(deep, liveNode.weight, liveNode.value) > bestValue){

joinQuenen(liveNode.weight, liveNode.value, deep, 0);

}

//记录并去掉队列最前面的结点

liveNode = liveNodeQueue.front();

liveNodeQueue.pop();

//判断是否为同一深度

if(liveNode.weight == -1){

//活结点表为空

if(liveNodeQueue.empty()){

break;

}

//获取下一个活结点，并从队列中去掉

liveNode = liveNodeQueue.front();

liveNodeQueue.pop();

//加入标记结点，表明是同一深度

liveNodeQueue.push(tag);

//深度加1

deep++;

}

}

}

//限界条件,当做背包问题求最大价值

int bound(int deep, int cBag, int value){

int left = c - cBag; //剩余重量

int valueTemp = value; //当前价值

while(deep<n && w[deep]<=left){

left -= w[deep];

valueTemp += v[deep];

deep++;

}

if(deep<n){

valueTemp += left\*(v[deep]/w[deep]);

}

return valueTemp;

}

（2）用户屏幕

