**算法设计与分析-报告书**

**0/1背包问题的实现与性能分析**

**姓名：董浩 学号：2018230408 班级：计数182**

目的：深刻掌握用蛮力法、回溯法、动态规划法、分支限界法的设计思想

内容包含：

1、用蛮力法、回溯法、动态规划法、分支限界法设计0/1背包问题的实现方法

2、分析算法随n和C变化的时间性能

3、比较几种算法在0/1背包问题求解上的特点

蛮力法解决背包问题

#include <iostream>

#include <vector>

#define PAC\_MAX\_VOL 10

using namespace std;

void dfs(const vector<int> weights, const vector<int> vals, bool visit[], int currWeight, vector<int> seq);

void outputResult(vector<int> weights, vector<int> vals, bool visit[], int currWeight, vector<int> seq);

void dfs(const vector<int> weights, const vector<int> vals, bool visit[], int currWeight, vector<int> seq) {

for (int i = 0; i < 4; ++i) {

if (!visit[i]) {

if (currWeight + weights[i] > PAC\_MAX\_VOL)

continue;

visit[i] = true;

seq.push\_back(i);

dfs(weights, vals, visit, (currWeight + weights[i]), seq); seq.erase(find(seq.begin(), seq.end(), i));

visit[i] = false;

}

}

outputResult(weights, vals, visit, currWeight, seq);

return;

}

void outputResult(vector<int> weights, vector<int> vals, bool visit[], int currWeight,vector<int> seq) {

cout << "当前背包的重量是：" << currWeight << ", ";

int sumVal = 0;

for (int i = 0; i < 4; ++i)

if (visit[i])

sumVal += vals[i];

cout << "当前背包的总价值为：" << sumVal << endl;

cout << "当前结果集(编号)是：";

for (auto ele : seq) {

cout << ele << " ";

}

cout << "\n-----------------------------" << endl;

}

int main()

{

const vector<int> weights = { 7,3,4,5 };

const vector<int> vals = { 42,12,40,25 };

vector<int> seq;

bool visit[4] = {false};

dfs(weights, vals, visit, 0, seq);

return 0;

}

回溯法

#include <stdio.h>  
#include <conio.h>  
  
int n;

double c;  
double v[100];double w[100];  
double cw = 0.0;  
double cp = 0.0;  
double bestp = 0.0;  
double perp[100];  
int order[100];  
int put[100];  
  
  
void knapsack()  
{  
    int i,j;  
    int temporder = 0;  
    double temp = 0.0;  
  
   for(i=1;i<=n;i++)  
       perp[i]=v[i]/w[i];  
   for(i=1;i<=n-1;i++)  
    {  
       for(j=i+1;j<=n;j++)  
           if(perp[i]<perp[j])perp[],order[],sortv[],sortw[]  
       {  
           temp = perp[i];  
           perp[i]=perp[i];  
           perp[j]=temp;  
  
           temporder=order[i];  
           order[i]=order[j];  
           order[j]=temporder;  
           temp = v[i];  
           v[i]=v[j];  
           v[j]=temp;  
  
           temp=w[i];  
           w[i]=w[j];  
           w[j]=temp;  
       }  
    }  
}  
  
void backtrack(int i)  
{  
    double bound(int i);  
   if(i>n)  
    {  
       bestp = cp;  
       return;  
    }  
   if(cw+w[i]<=c)  
    {  
       cw+=w[i];  
       cp+=v[i];  
       put[i]=1;  
       backtrack(i+1);  
       cw-=w[i];  
       cp-=v[i];  
    }  
   if(bound(i+1)>bestp)       backtrack(i+1);  
}  
  
double bound(int i)  
{  
    double leftw= c-cw;  
    double b = cp;  
   while(i<=n&&w[i]<=leftw)  
    {  
       leftw-=w[i];  
       b+=v[i];  
       i++;  
    }  
   if(i<=n)  
       b+=v[i]/w[i]\*leftw;  
    return b;  
  
}  
  
  
int main()  
{  
    int i;  
   printf("请输入物品的数量和容量：");  
    scanf("%d %lf",&n,&c);  
   printf("请输入物品的重量和价值：");  
   for(i=1;i<=n;i++)  
    {  
       printf("第%d个物品的重量：",i);  
       scanf("%lf",&w[i]);  
       printf("价值是：");  
       scanf("%lf",&v[i]);  
       order[i]=i;  
    }  
   knapsack();  
   backtrack(1);  
   printf("最有价值为：%lf\n",bestp);  
   printf("需要装入的物品编号是：");  
   for(i=1;i<=n;i++)  
    {  
       if(put[i]==1)  
           printf("%d ",order[i]);  
    }  
    return 0;  
}

动态规划

#include<stdio.h>  
#include<stdlib.h>  
int c[50][50];   
int w[10],v[10];   
int x[10];   
int n;   
void KNAPSACK\_DP(int n,int W);  
void OUTPUT\_SACK(int c[50][50],int k) ;  
void KNAPSACK\_DP(int n,int W)   
{   
int i,k;  
for(k=0;k<=W;k++)   
c[0][k]=0;   
for(i=1;i<=n;i++)   
{   
c[i][0]=0;   
for(k=1;k<=W;k++)   
{   
if(w[i]<=k)   
{   
if(v[i]+c[i-1][k-w[i]]>c[i-1][k])   
c[i][k]=v[i]+c[i-1][k-w[i]];   
else   
c[i][k]=c[i-1][k];   
}   
else   
c[i][k]=c[i-1][k];   
}   
}   
}   
void OUTPUT\_SACK(int c[50][50],int k)   
{   
int i;  
for(i=n;i>=2;i--)   
{   
if(c[i][k]==c[i-1][k])   
x[i]=0;   
else   
{   
x[i]=1;   
k=k-w[i];   
}   
}   
x[1]=(c[1][k]?1:0);   
for(i=1;i<=n;i++)   
printf("%4d",x[i]);  
}   
void main()   
{   
int m;   
int i,j,k;  
printf("输入物品个数:");  
scanf("%d",&n);  
printf("依次输入物品的重量:\n");  
for(i=1;i<=n;i++)   
scanf("%d",&w[i]);  
printf("依次输入物品的价值:\n");  
for(i=1;i<=n;i++)   
scanf("%d",&v[i]);  
printf("输入背包最大容量:\n");  
scanf("%d",&m);   
for(i=1;i<=m;i++)   
printf("%4d",i);   
printf("\n");   
KNAPSACK\_DP(n,m);   
printf("构造最优解过程如下:\n");  
for(j=1;j<=5;j++)   
{   
for(k=1;k<=m;k++)   
printf("%4d",c[j][k]);   
printf("\n");  
}   
printf("最优解为:\n");  
OUTPUT\_SACK(c,m);   
system("pause");  
}

分支限界

#include <iostream>

#include <vector>

#include <stack>

#include <algorithm>

#include <list>

using namespace std;

const int NKNAPSACKCAP = 10;

class Goods {

public:

int weight;

int value;

friend ostream& operator<<(ostream &os, const Goods &out);

};

ostream& operator<<(ostream &os, const Goods &out)

{

os << "重量：" << out.weight << " 价值： " << out.value;

return os;

}

typedef vector<Goods> AllGoods;//定义所有货物数据类型

class Node

{

public:

Node \*parent;

int nWeight;

int nValue;

int id;

bool leftChild;

Node(Node \*\_parent, bool \_left, int weight, int value, int \_id) :parent(\_parent), leftChild(\_left), nWeight(weight), nValue(value), id(\_id)

{}

~Node()

{

}

};

class Knapsack

{

private:

int capacity;//背包容量

int nGoodsNum;//物品数

vector<Goods> goods;

int nMaxValue;

int nCurrentWeight;

int nCurrentValue;

vector<bool> bestResult;

int bound(int i)

{

int nLeftCapacity = capacity - nCurrentWeight;

int tempMaxValue = nCurrentValue;

while (i < nGoodsNum && goods[i].weight <= nLeftCapacity)

{

nLeftCapacity -= goods[i].weight;

tempMaxValue += goods[i].value;

}

if (i < nGoodsNum)

{

tempMaxValue += (float)(goods[i].value) / goods[i].weight \* nLeftCapacity;

}

return tempMaxValue;

}

public:

Knapsack(AllGoods &AllGoods, int nKnapsackCap)

{

nGoodsNum = AllGoods.size();

capacity = nKnapsackCap;

nCurrentWeight = 0;

nCurrentValue = 0;

nMaxValue = 0;

for (int i = 0; i < nGoodsNum; ++i)

{

goods.push\_back(AllGoods[i]);

bestResult.push\_back(false);

}

}

void sortByUintValue()

{

stable\_sort(goods.begin(), goods.end(), [](const Goods& left, const Goods& right)

{return (left.value \* right.weight > left.weight \* right.value); });

}

void printGoods()

{

for (size\_t i = 0; i < goods.size(); ++i)

{

cout << goods[i] << endl;

}

}

void printResult()

{

cout << "MAX VALUE: " << nMaxValue << endl;

for (int i = 0; i < nGoodsNum; ++i)

{

if (bestResult[i])

{

cout << goods[i] << endl;

}

}

}

void knapsack\_0\_1\_branchAndBound()

{

list<Node\*> activeNodes;

list<Node\*> diedNodes;

sortByUintValue();

if (goods[0].weight < capacity)

{

activeNodes.push\_back(new Node(nullptr, true, goods[0].weight, goods[0].value,0));

}

activeNodes.push\_back(new Node(nullptr, false, 0,0,0));

Node \*curNode = nullptr;

Node \*preNode = nullptr;

int curId;

while (!activeNodes.empty())

{

curNode = activeNodes.front();

activeNodes.pop\_front();

diedNodes.push\_back(curNode);

if (curNode->id + 1 == nGoodsNum)

{

continue;

}

if (nMaxValue < curNode->nValue)

nMaxValue = curNode->nValue;

preNode = curNode;

while (nullptr != preNode)

{

bestResult[preNode->id] = preNode->leftChild;

preNode = preNode->parent;

}

}

nCurrentValue = curNode->nValue;

nCurrentWeight = curNode->nWeight;

curId = curNode->id;

if (nMaxValue >= bound(curId + 1))//剪枝

{

continue;

}

if (nCurrentWeight + goods[curId + 1].weight <= capacity)

{

activeNodes.push\_back(new Node(curNode, true, nCurrentWeight + goods[curId + 1].weight, nCurrentValue + goods[curId + 1].value, curId + 1));

}

activeNodes.push\_back(new Node(curNode, false, nCurrentWeight,nCurrentValue,curId + 1));

}

while (!diedNodes.empty())

{

curNode = diedNodes.front();

delete curNode;

diedNodes.pop\_front();

}

}

};

void GetAllGoods(AllGoods &allGoods)

{

Goods goods;

goods.weight = 2;

goods.value = 6;

allGoods.push\_back(goods);

goods.weight = 2;

goods.value = 3;

allGoods.push\_back(goods);

goods.weight = 2;

goods.value = 8;

allGoods.push\_back(goods);

goods.weight = 6;

goods.value = 5;

allGoods.push\_back(goods);

goods.weight = 4;

goods.value = 6;

allGoods.push\_back(goods);

goods.weight = 5;

goods.value = 4;

allGoods.push\_back(goods);

}

int main()

{

AllGoods allGoods;

GetAllGoods(allGoods);

Knapsack knap(allGoods,NKNAPSACKCAP);

knap.printGoods();

knap.knapsack\_0\_1\_branchAndBound();

knap.printResult();

return 0;

}

蛮力法：

对于有n种可选物品的0/1背包问题，其解空间由长度为n的0-1向量组成,可用子集数表示。在搜索解空间树时，深度优先遍历，搜索每一个结点，无论是否可能产生最优解，都遍历至叶子结点，记录每次得到的装入总价值，然后记录遍历过的最大价值。

蛮力法求解0/1背包问题的时间复杂度为：T(n) O(2n)。

动态规划法：

第一阶段，只装入前1个物品，确定在各种情况下的背包能够得到的最大价值；第二阶段，只装入前2个物品，确定在各种情况下的背包能够得到的最大价值；以此类推，直到第n个阶段。最后，V(n,C)便是在容量为C的背包中装入n个物品时取得的最大价值

时间复杂度为：T(n) O(n C)。

回溯法：

对于有n种可选物品的0/1背包问题，其解空间由长度为n的0-1向量组成,可用子集数表示。在搜索解空间树时，只要其左儿子结点是一个可行结点，搜索就进入左子树。当右子树中有可能包含最优解时就进入右子树搜索。

时间复杂度为：T(n) O(2n)。

分支限界：节点的优先级由已装袋的物品价值加上剩下的最大单位重量价值的物品装满剩余容量的价值和。算法首先检查当展结点的左儿子结点的可行性。如果该左儿子结点是可行结点，则将它加入到子集树和活结点优先队列中。当前扩展结点的右儿子结点一定是可行结点，仅当右儿子结点满足上界约束时才将它加入子集树和活结点优先队列。当扩展到叶节点时为问题的最优值。

时间复杂度为：T(n) O(2n)。