

**实 验 报 告**

（ 2021/2022 学年 第 2 学期）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 算法分析与设计 | | | | | |
| 实验名称 | 回溯法 | | | | | |
| 实验时间 | 2022 | 年 | 3 | 月 | 28 | 日 |
| 指导单位 | 南京邮电大学计算机学院、软件学院 | | | | | |
| 指导教师 | 陈春玲 | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 学生姓名 | 张宸冉 | 学 号 | B20060527 |
| 学院(系) | 计算机学院 | 专 业 | 计算机科学与技术 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | | 回溯法 | | | | 指导教师 | | | 陈春玲 |
| 实验类型 | | 验证 | 实验学时 | | 2 | 实验时间 | | | 2022.5.7 |
| 1. **实验任务和目的** 2. 目的：   学习编程实现深度优先搜索状态空间树求解实际问题的方法，着重体会求解第一个可行解和求解所有可行解之间的差别。加深理解回溯法通过搜索状态空间树、同时用约束函数剪去不含答案状态子树的算法思想，会用蒙特卡罗方法估计算法实际生成的状态空间树的结点数。   1. 任务： 2. 要求用回溯法求解**8-皇后问题**，使放置在8\*8棋盘上的8个皇后彼此不受攻击，即：任何两个皇后都不在同一行、同一列或同一斜线上。请输出8皇后问题的**所有可行解**。 3. 用回溯法编写一个**递归程序**解决如下**装载问题**：有n个集装箱要装上2艘载重分别为和的轮船，其中集装箱i的重量为（1≤ i ≤ n），且≤+。问是否有一个合理的装载方案可以将这n个集装箱装上这2艘轮船？如果有，请给出装载方案。 | | | | | | | | | |
| 1. **实验环境**   硬件：AMD Ryzen 7 4800H with Radeon Graphics  软件：Windows操作系统，Visual C++ 6.0。 | | | | | | | | | |
| **三、核心算法设计及分析**  **1、核心算法设计**  **1）8-皇后问题**  N皇后问题的**解**可以写成n元组的形式（x1,x2,…，xn-1），对于每一个xi表示第i行皇后的列号  对于**约束条件**，有两种方案。1、显式约束为：。对应的隐式约束就是。2、显式约束为：，隐式约束就是。  对于第一种显式约束，解空间的大小是，对于第二种显式约束，解空间的大小是n！。  算法在程序中有两个函数，一个是NQueens(int k, int n, int\* x)递归版本，一个是Place(int k, int i, int\* x)。  Place是**约束函数**用于判断，下标为k的行上，皇后放在下标为i的列是否可行，NQueens函数首先调用place函数进行判断，如果判断成功，就在解向量下标为k的元素上输入i。如果状态空间树已经搜索到了解状态，就会输出，没有搜索到解状态就会将k+1进行下一次递归调用。    **思考题**  **1）**输出12个独立解，本质是对输出的92个可行解进行后处理，根据皇后问题的性质，将输出的解，旋转90°，180°，270°，上下对称，左右对称，沿着主对角线对称，沿着副对角线对称，可以得到7个不独立的解。  具体的转换公式如下：  Y表示当前的解数组，x是存放独立解的矩阵，n是皇后问题皇后的个数，num是solutions的横坐标（当前独立解的个数）  90°： y[ x[num][j] ] = n - 1 - j;  180°： y[n-1-j] = n -1 - x[num][j];  270°： y[n - 1 - x[num][j]] = j;  上下对称： y[n - 1 - j] =x[num][j];  左右对称： y[j] = n - 1 - x[num][j];  主对角线对称： y[x[num][j]] = j ;  副对角线对称： y[n - 1 - x[num][j]] = n - 1 - j;  **2）**在输出一个可行解之后停止函数，只要增加一个flag标志就可以实现。Flag标记用指针保存，在for循环的条件中增加flag判断逻辑。输出语句一旦达到符合输出的条件，就可以将flag标志取反，使得后续NQueens不会进行。  **2）装载问题**  装载问题可以看成是一种特殊的0-1背包问题，当第一艘船对于所有集装箱尽最大可能的装满之后，如果剩余的集装箱的重量小于第二艘船，就得到了可行解。  所以装载问题就转化成了目标函数是的0-1背包问题。  第一艘船的**解结构**可以写成n元组的形式（x1,x2,…，xn-1），其中的取值只有0和1，当取0的时候，表示当前编号的集装箱不会装入第一艘船，取1的时候，表示当前编号的集装箱会装入第一艘船。  对于**约束条件**，显式约束就是每一个的取值只能是0和1。对于隐式约束，有。  用cw表示考虑到第k个集装箱的时候，当前第一艘船的载重量，r表示考虑到第k个集装箱的时候，剩余集装箱的重量。Bestw表示考虑完一种对于第一艘船的装载组合后迄今为止最好的解值。  因此，**约束函数**可以这样定义。如果cw+w[i] ≤c，即加入当前考虑的集装箱不会超过第一艘船的承载能力，就可以生成左子树。如果cw + r > bestw，即当前考虑的集装箱不加入第一艘船的话，后面所有没有考虑的集装箱全部加入第一艘船可以比迄今最多的装载量大（如果小，就算是后面的集装箱全部加入也没有意义）就可以生成右子树。  在算法中，如果已经考虑完所有集装箱，比较当前cw和bestw，如果cw大于了bestw就更新bestw，并且将当前解的路径使用bestx数组保存，用于最后的输出。  Show函数用于后处理，先是判断剩余的集装箱可以不可以装入第二艘船，如果装不进去，就输出失败。装的进去，第一艘船没有装入的集装箱，就会装入第二艘船。    **思考题**  **1）**  **1、**第一种策略是先运行计算最优解值的算法，求得bestw，然后再创建一个新的类将bestw直接作为初始化参数传递给对象。这样bestx数组只会改变一次，其他所有方案都无法通过限界函数的判断。  **2、**另一种策略是在算法中动态的更新bestx，在第i层将当前最优解x[i]写入bestx[i]中。由于限界函数的存在，在第一次到达叶结点后，如果其他解向量无法超过当前最优解是无法达到叶结点的，index的值不会改变，所以无法向bestx最优解存储数组内更新数据，从而达到了减少时间复杂度的效果。  **2）**使用迭代的方法实现回溯法，主要是需要用一个变量来控制迭代的深度。  在算法中具体就是搜索左子树或者是右子树之后，深度变量i需要加1。当i到达最深，也就是搜索到达了叶结点，就会向bestx数组输出当前的一个结果，由于在搜索右子树的时候已经将不能实现最大值的情况舍去，所以到达叶结点的都是当前最优解。  在满足了不能生成右子树的条件时，就会向上回溯，这时只要让i减1就会向上回溯。  **2、核心算法分析**  **1）8-皇后问题**  运用蒙特卡罗方法可以估计实际生成的结点数。蒙特卡罗方法是在状态空间树中随机选择一条路径，x是这个路径上的一个结点，x不受限制的孩子结点的个数是m，则认为整个状态空间树的这一层不受限制的孩子结点的个数都是m。  随机选取5条路径（1，3，0，2，4）、（3，5，2，4，6，0）、（0，7，5，2，6，1，3）、（0，2，4，1，3）、（2，5，1，6，0，3，7，4）。参考第一个路径，第一个结点是1，所以第二层不受限制的结点个数是8-3=5，同理可以计算出第三行……不受限制结点的个数。  所以选择这条路径，状态空间树上世纪生成的问题状态数是：1649，对随机生成的5个路径的总状态空间树个数求平均值可以得到最终是1625个。所以8皇后问题的状态空间树的结点总数是109601。  **2）装载问题**  对于n个集装箱，长为n的解向量最坏可能更新2^n次，更新每一次需要考查n个结点，所以更新一次的时间复杂度是O（n），整个算法时间复杂度最坏是O（n\*2^n） | | | | | | | | | |
| **四、实验及结果分析**  **1、实验**  **1）8-皇后问题**  **输出所有解**    **输出一个解**    **输出12个独立解**    **2）装载问题**  输入**22 35 24 19 4：（迭代法和递归法输出一样）**    输入**22 35 24 15 4：**    输入**22 35 24 15 3：**    **思考题的两种修改策略，都可以达到最优解和最优解值：**    **2、实验结果分析**  **1）8-皇后问题**  通过对8皇后问题的代码进行单步调试，可以看出算法按照深度优先的顺序对状态空间树进行动态的创建和搜索。约束函数，对于皇后互相攻击的情况进行了剪枝，让会出现冲突的分支直接不会生成。  通过对称变化的关系，可以画图证明对称变化公式的正确性，并且对称变化得到的解答也是符合n皇后问题的规则的。  使用flag标记对n皇后问题的整个函数第一个循环进行逻辑限制，单步调试可以看到，当已经在输出端口输出了一个可行解之后，flag标记取反，在下一次递归，for循环条件不满足，直接走到n皇后问题函数的底端，函数执行结束。Flag标记很好的实现了对一个可行解的输出  **2）装载问题**  从第三个测试用例可以看出，第一艘船在最后一个集装箱的重量是3的时候刚好装满，如果最后一个集装箱是4，则第一个集装箱无法装满，从而导致最后一个集装箱无法装下。很显然，第一个测试用例总重是104，已经超过了两艘船的总重量。第二个测试用例总重量刚好是100，但是由于两个船载重的组合，使得第二个用例无法得出可行解。  通过单步测试，也可以很清楚的看出递归函数的递归调用过程，可以按照递归的顺序深度优先生成状态空间树，并且在不符合条件的情况下进行剪枝，没有递归调用，递归调用结束之后的恢复语句，也可以再单步调试中清楚的看清他们的逻辑功能。 | | | | | | | | | |
| **五、实验小结（包括问题和解决办法、心得体会、建议和意见等）**  通过本次实验，对回溯法通过递归调用的方法，按照深度优先原则动态生成状态空间树，并且通过约束函数进行剪枝，减少状态空间树搜索的个数有了深入了解。对皇后问题的约束条件有了更深的理解。学会了使用变量控制输出个数。  了解了n皇后问题解的对称性，会用对称性对解答进行后处理得到独立解。  通过装载问题，对0-1背包问题有了更深理解，对限界函数的作用有了更深的认识，知道了怎么通过建立深度控制变量，把递归函数转换成非递归。 | | | | | | | | | | |
| **六、指导教师评语** | | | | | | | | | | |
| 成 绩 |  | | 批阅人 |  | | | 日 期 |  | | |

**附件：**

**8-皇后问题以及输出一个可行解：**

#include<iostream>

using namespace std;

bool Place(int k, int i, int\* x) {

//判断两个皇后是否在同一列或者是同一斜线上

//k是当前皇后的个数 i是k+1个皇后能否放在第i列

for (int j = 0; j < k; j++) {

if ((x[j] == i) || (abs(x[j] - i) == abs(j - k))) {

return false;

}

}

return true;

}

void NQueens(int k, int n, int\* x) {

for (int i = 0; i < n; i++) { //显式约束 当前递归调用循环n个列还是没有找到结果就会通过递归调用回溯

if (Place(k, i, x)) { //可以放置皇后： （约束函数）

x[k] = i;

if (k == n - 1) {

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

cout << x[j] << " ";

}

cout << endl;

}

else

{

NQueens(k + 1, n, x);

}

}

}

}

void NQueens(int n, int\* x) {

//递归入口

NQueens(0, n, x);

}

void NQueens1(int k, int n, int\* x, int\* flag) {

//输出一个解答之后就不输出其他的解答了 在顶层调用的时候传递一个flag变量（指针） 一旦输出了一个解答

for (int i = 0; i < n && !\*flag; i++) { //显式约束 当前递归调用循环n个列还是没有找到结果就会通过递归调用回溯

if (Place(k, i, x)) { //可以放置皇后： （约束函数）

x[k] = i;

if (k == n - 1) {

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

cout << x[j] << " ";

}

cout << endl;

\*flag = 1;

}

else {

NQueens1(k + 1, n, x, flag);

//当输出一个解之后 会回到上一个递归调用点 直接return掉

}

}

}

}

void NQueens1(int n, int\* x) {

//递归入口

int flag = 0;

NQueens1(0, n, x, &flag);

}

int main() {

int n = 5;

int\* x = new int[n];

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

x[i] = -1;

}

//NQueens(n, x);

NQueens1(n, x);

//NQueens2(n, x);

}

**8-皇后12个独立解;**

#include <iostream>

#include <math.h>

#include <iomanip>

using namespace std;

class nqueen{

public:

nqueen(int N); //构造函数

~nqueen(); //析构函数

void nqueens();

void Out\_Solutions(); //输出Solutions数组中的12个独立解

private:

bool Place(int k, int i); //约束函数, 判断两个皇后是否在同一列或同一斜线，不冲突返回true

void nqueens(int k); //皇后问题主函数

bool Constructing(int num); //判断是否独立解，是则返回true。

bool Test(int num);

int\* y; //临时保存变换后的可行解

int\*\* solutions; //存12个独立解。由于递归，solutions数组不能只开12行，否则输出结果出错！

int number; //独立解的个数

int n; //n-皇后问题

int\* x; //存放一个解

};

nqueen::nqueen(int N) //构造函数

{

n = N; number = 0;

x = new int[n];

solutions = new int\* [20];

for (int i = 0; i < 20; i++) {

solutions[i] = new int[n];

}

y = new int[n];

}

nqueen::~nqueen() //析构函数

{

delete[]x;

for (int i = 0; i < 20; i++) {

delete[]solutions[i];

}

delete[]solutions;

delete[]y;

}

//输出solutions数组中的12个独立解

void nqueen::Out\_Solutions()

{

int i, j;

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

{

cout << setw(4) << i + 1 << " (";

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

cout << solutions[i][j] << ",";

}

cout << solutions[i][j] << ")" << endl;

}

}

//约束函数, 判断两个皇后是否在同一列或同一斜线，不冲突返回true

bool nqueen::Place(int k, int i)

{

for (int j = 0; j < k; j++)

if ((x[j] == i) || (abs(x[j] - i) == abs(j - k)))

return false; //在同一列或同一斜线，返回false

return true;

}

void nqueen::nqueens()

{

nqueens(0);

}

void nqueen::nqueens(int k)

{

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

if (Place(k, i)){ //判断第k个皇后能否放在下标i列

x[k] = i;

if (k == n - 1){

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

solutions[number][i] = x[i]; //第x[i]列元素放在第number个解的第i个位置，即将一个解x[]放到solutions数组中的number行。

}

if (Constructing(number)) { //判断是否独立解 判断通过给number+1 number记录了独立解的个数

number++;

}

}

else {

nqueens(k + 1);

}

}

}

}

bool nqueen::Constructing(int num){ //把新得到的解x[]做7种变换后，得到的结果保存到y[]中,和之前的独立解比较，相同返回false，不同返回true

int j, k;

//把新得到的解x[]旋转90°保存到y[]中

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

y[ solutions[num][j] ] = n - 1 - j;

}

for (k = 0; k < num; k++) {

if (Test(k)) { //test判断是否有相同 只有返回相同 才会走到return语句 直接让Constructing返回否定 这个解是重复解

return false;

}

}

//把新得到的解x[]旋转180°保存到y[]中

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

y[n-1-j] = n -1 - solutions[num][j];

}

for (k = 0; k < num; k++) {

if (Test(k)) {

return false;

}

}

//把新得到的解x[]旋转270°保存到y[]中

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

y[n - 1 - solutions[num][j]] = j;

}

for (k = 0; k < num; k++) {

if (Test(k)) {

return false;

}

}

//把新得到的解x[]上下对称变换后保存到y[]中

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

y[n - 1 - j] =solutions[num][j];

}

for (k = 0; k < num; k++) {

if (Test(k)) {

return false;

}

}

//把新得到的解x[]左右对称变换后保存在y[]中

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

y[j] = n - 1 - solutions[num][j];

}

for (k = 0; k < num; k++) {

if (Test(k)) {

return false;

}

}

//把新得到的解x[]关于y=x对角线对称变换后保存到y[]中

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

y[solutions[num][j]] = j ;

}

for (k = 0; k < num; k++) {

if (Test(k)) {

return false;

}

}

//把新得到的解x[]关于y=-x对角线对称变换后保存到y[]中

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

y[n - 1 - solutions[num][j]] = n - 1 - j;

}

for (k = 0; k < num; k++) {

if (Test(k)) {

return false;

}

}

return true; //与之前得到的独立解不同，返回true

}

//判断变换后得到的解y[]与之前得到的独立解solutions[num][]是否相同

bool nqueen::Test(int num){

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

if (y[i] != solutions[num][i]) {

return false; //不相同，返回false，该解为独立解

}

}

return true; //相等，返回true，该解为重复解

}

void main()

{

int n = 8;

nqueen solution(n);

cout << "8-皇后问题的12个独立解(solutions数组)：" << endl;

solution.nqueens();

solution.Out\_Solutions();

}

**装载问题，包含思考题**

#include<iostream>

#include <iomanip>

using namespace std;

template <class T>

class Loading{

private:

int n, //集装箱数

\* x, //当前解 长度为n 的0-1数组

\* bestx; //当前第一艘船的最优解 长度为n 的0-1数组

int index; //当前最优解的下标变量，为实现改进策略(2)增设的变量

T c1, //第一艘轮船的核定载重量

c2, //第二艘轮船的核定载重量

\* w, //集装箱重量数组

total=0, //所有集装箱重量之和

cw, //当前第一艘船的载重量

bestw, //当前第一艘船的最优载重量 最优解值

r; //剩余集装箱总重量

void Backtrack(int i); //找到最接近第一艘轮船载重c1的最佳装载方案，最优载重值bestw，最优解数组bestx。

void BacktrackImprove2(int i); //动态生成bestx 防止不必要的覆盖

void BacktrackImprove1(int i, int\* bestw); //运行最优解值计算，求得最优解值bestw 然后直接通过最优解值作为条件

public:

Loading(int n,T c1,T c2,T\* w ){

this->n = n;

this->x = new int[n];

this->bestx = new int[n];

this->c1 = c1;

this->c2 = c2;

this->w = w;

this->index = 0;

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

this->total += w[i];

}

cw = 0;

bestw = 0;

r = total;

}

~Loading(){

delete bestx;

delete x;

}

void Backtrack() {

//对外接口

Backtrack(0);

}

void Show();//输出整个装载方案

void MaxLoading(); //非递归算法

void BacktrackImprove2() {

BacktrackImprove2(0);

}

void BacktrackImprove1() {

int bestw = 0;

BacktrackImprove1(0,&bestw);

cout << bestw;

}

};

template <class T>

void Loading<T>::Backtrack(int i){ //搜索第i层结点

if (i == n){//到达叶节点

if (cw > bestw){ //只有在当前解大于迄今为止的最优解的时候，才会把最优解和最优解值进行更新

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

bestx[j] = x[j];

}

bestw = cw;

}

return;

}

//搜索子树

r -= w[i]; //不论一个结点是否加入 剩余的重量都会减少

/\*搜索左子树\*/

if (cw + w[i] <= c1){ //x[i]=1时的可行解约束条件

x[i] = 1;

cw += w[i];

Backtrack(i + 1);

cw -= w[i]; // 这是左子树递归的返回点 当前载重量在回到父节点的时候需要将当前装载进去的取出 修改cw值

}

/\*搜索右子树\*/

if (cw + r > bestw) { //x[i]=0时增加的约束函数，剪去不含最优解的分枝

x[i] = 0;

Backtrack(i + 1);

}

r += w[i]; //这是递归的返回点【左子树右子树都会经过这个语句】 向父节点返回的时候 记录剩余重量的值需要增加相应的值

}

template<class T>

void Loading<T>::MaxLoading() {

int i = 0; //使用一个变量标记递归深度

while (true){

//搜索左子树

while (i<n&& cw + w[i] <= c1){

r -= w[i];

x[i] = 1;

cw += w[i];

i++;

}

//到达叶结点

if (i == n) {

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

bestx[j] = x[j];

}

bestw = cw;

}

//左子树没有一直搜索到叶结点，就进入右子树

else{

r -= w[i]; //这里只是进入右子树一次

x[i] = 0;

i++;

}

//如果满足了不能生成右子树的条件 就会向上回溯

while (cw + r <= bestw) {

i--; //先向上一层

while (i > 0 && !x[i]) { //除非到顶或者已经搜索过右子树（搜索过右子树这个结点就不用再看了）

r += w[i];

i--;

}

if (i == 0) {

return;

}

//如果这个顶点只搜索了左子树 还有右子树没有搜索 就会进入右子树

x[i] = 0;

cw -= w[i];

i++;

}

}

}

template <class T>

void Loading<T>::Show(){

//后处理函数

//对于整个类 通过Backtrack函数已经得到了 最优解向量bestx 最优解值bestw

int totalW = this->total;

int c1W = 0; //第一艘船总载重

int i;

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

{

if (bestx[i] == 1) { //根据解向量 计算第一艘船的总载重量

c1W += w[i];

}

}

if (totalW - c1W > c2) //如果第一艘船剩下的重量无法全部装在第二艘船上 方案失败

{

cout << " 没有合理的装载方案!" << endl;

return; //直接返回 没有后续输出

}

//下面是第一艘船尽可能大的装载 第二艘可以装下剩余的货物的情况 就是方案成功

cout << " 最优解(第一艘船)：(";

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

cout << bestx[i] << ",";

}

cout << bestx[i] << ")" << endl;

cout << " 最优解(第二艘船)：(";

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

cout << !bestx[i] << ","; //第二艘船的装载情况就是第一艘船取反

}

cout << !bestx[i] << ")" << endl;

//一些额外输出

cout << " 装载方案如下: " << endl;

cout << " 第一艘船装载:" << endl;

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

if (bestx[i] == 1) {

cout << " 集装箱" << i << ": " << setw(2) << w[i] << endl;

}

}

cout << " 总载重: " << c1W;

if (c1 - c1W == 0) {

cout << "，装满！" << endl;

}

else {

cout << "，剩余载重量" << c1 - c1W << "！" << endl;

}

cout << " 第二艘船装载:" << endl;

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

if (!bestx[i] == 1) {

cout << " 集装箱" << i << ": " << setw(2) << w[i] << endl;

}

}

cout << " 总载重: " << totalW - c1W;

if (c2 - (totalW - c1W) == 0) {

cout << "，装满！" << endl;

}

else {

cout << "，剩余载重量" << c2 - (totalW - c1W) << "！" << endl;

}

}

template<class T>

void Loading<T>::BacktrackImprove1(int i,int\* bestw) {

if (i == n) {//到达叶节点

if (cw > \*bestw) { //只有在当前解大于迄今为止的最优解的时候，才会把最优解和最优解值进行更新

\*bestw = cw;

}

return;

}

//搜索子树

r -= w[i]; //不论一个结点是否加入 剩余的重量都会减少

/\*搜索左子树\*/

if (cw + w[i] <= c1) { //x[i]=1时的可行解约束条件

x[i] = 1;

cw += w[i];

BacktrackImprove1(i + 1,bestw);

cw -= w[i]; // 这是左子树递归的返回点 当前载重量在回到父节点的时候需要将当前装载进去的取出 修改cw值

}

/\*搜索右子树\*/

if (cw + r > \*bestw) { //x[i]=0时增加的约束函数，剪去不含最优解的分枝

x[i] = 0;

BacktrackImprove1(i + 1,bestw);

}

r += w[i]; //这是递归的返回点【左子树右子树都会经过这个语句】 向父节点返回的时候 记录剩余重量的值需要增加相应的值

}

template<class T>

void Loading<T>::BacktrackImprove2(int i) {

//搜索第i层结点

if (i == n){ //到达叶节点

index = n-1; //到达叶节点，置index=n。该语句是改进策略(2)增加的

bestw = cw;

cout << setw(45) << "修正bestw=" << bestw << endl;

return;

}

//搜索子树

r -= w[i];

if (cw + w[i] <= c1){

//搜索左子树

x[i] = 1;

cw += w[i];

BacktrackImprove2(i + 1);

if (index == i){ //回溯上一层时，若index等于i，

bestx[i] = x[i]; //则将x[i]存入bestx[i]

index--;

}

cw -= w[i];

}

if (cw + r > bestw){

//搜索右子树

x[i] = 0;

BacktrackImprove2(i + 1);

if (index == i){ //回溯上一层时，若index等于i，

bestx[i] = x[i]; //则将x[i]存入bestx[i]

index--;

}

}

r += w[i];

}

int main(){

int w[5] = { 22,35,24,15,3 };

Loading<int> ld1(5,60,40,w);

Loading<int> ld2(5, 60, 40, w);

Loading<int> ld3(5, 60, 40, w);

Loading<int> ld4(5, 60, 40, w);

ld1.Backtrack();

ld1.Show();

cout << endl;

ld2.MaxLoading();

ld2.Show();

cout << endl;

ld3.BacktrackImprove2();

ld3.Show();

cout << endl;

ld4.BacktrackImprove1();

return 0;

}