**算法分析与设计实验报告**

**第 3 次实验**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 杨杰 | 学号 | 201908010705 | | 班级 | 计科1907 |
| 时间 | 5.20 | 地点 | 软件大楼 | | | |
| 实验名称 | 用回溯法求解0-1背包问题 | | | | | |
| 实验目的 | 通过上机实验，要求掌握用回溯法求解0-1背包问题描述、算法设计思想、程序设计。 | | | | | |
| 实验原理 | 0-1背包属于找最优解问题，用回溯法需要构造解的子集树。在搜索解空间树时，只要左子节点是一个可行结点，搜索就进入其左子树。对于右子树，先计算上界函数，以判断是否进入右子树。 上界函数bound():当前价值cw+剩余容量可容纳的最大价值<=当前最优价值bestp。 假设规定左叉标1（代表选择该物品装入背包），右叉标0（代表不选择该物品装入背包）。  注意：  左子树的解的上界与父节点相同，不用计算。右子树的解的上界：较好的计算方法是将剩余物品依其单位重量价值排序，然后依次装入物品，直到装不下时，再装入该物品的一部分来装满背包。由此得到的价值是右子树中解的上界（尽管这不是一个可行解，但可以证明其价值是最优值的上界）。每次进入右子树之前都会计算右子树的界，如果右子树的界大于当前的界，才能进入右子树（即右子树的界满足约束条件）。每走到一个叶结点时就更新当前的界。 | | | | | |
| 实验步骤 | 1.程序运行时，从0结点开始出发：只要遇到1就一直往左走（先逐个将物品装入背包，直到装不下再向右走），直到背包装不下物品，才向右走。  2.根据限界函数判断是否进入右子树，如果不进入就回溯。  3.当走到叶结点时更新上界，然后回溯。 | | | | | |
| 关键代码 | *//按单位价值排序*  void knapsack()  {      int i, j;      for (i = 1; i <= n; i++)          perp[i] = v[i] / w[i];      int k = 0;      for (i = 1; i <= n - 1; i++)*//选择排序perp[],order[],sortv[],sortw[]*      {          k = i;          for (j = i + 1; j <= n; j++)              if (perp[k] < perp[j])                  k = j;          swap(&perp[i], &perp[k]);          swap(&order[i], &order[k]);          swap(&v[i], &v[k]);          swap(&w[i], &w[k]);      }  }  *//计算上界函数*  int bound(int i)  {      int leftw = c - cw;      int 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;  }  *//回溯函数*  void backtrack(int i)  {      if (i > n)      {          bestp = cp;          for (int i = 1; i <= n; i++)              bestway[i] = put[i];          return;      }      if (cw + w[i] <= c)      {          cw += w[i];          cp += v[i];          put[i] = 1;          backtrack(i + 1);          cw -= w[i];          cp -= v[i];          put[i] = 0;      }      if (bound(i + 1) > bestp)*//符合条件搜索右子树*          backtrack(i + 1);  } | | | | | |
| 算法复杂度分析 | 上界函数bound()需要O(n)时间，在最坏的情况下有O(2^n)个右子结点需要计算上界，所以回溯算法backtrack需要的计算时间为O(n2^n)。  knapsack()函数两层for循环嵌套，时间复杂度为O(n2)。  空间复杂度为O(n)。  另外如果用<algorithm>中的sort函数的话，排序的时间复杂度可以降到O(nlogn)。但是这样做需要修改代码：  struct Thing  {      double w;      double v;      int order;      int put;      double perp;      int bestway;  } thing[MAX];  bool cmp(Thing a, Thing b)  {      return a.v / a.w > b.v / b.w;  }  *//按单位价值排序*  void knapsack()  {      sort(thing + 1, thing + n + 1, cmp);  } | | | | | |
| 测试结果  （含运行时间） | **小规模数据**      **中规模数据**        **大规模数据** | | | | | |
| 实验心得 | 通过这个实验，我深入学习了解了回溯法。  回溯法思路的简单描述是：把问题的解空间转化成图或者树的结构表示，然后使用深度优先搜索策略进行遍历，遍历的过程中记录和寻找所有可行解或者最优解。  基本思想类同于：   * 图的深度优先搜索 * 二叉树的后序遍历   回溯法按深度优先策略搜索问题的解空间树。首先从根节点出发搜索解空间树，当算法搜索至解空间树的某一节点时，先利用剪枝函数判断该节点是否可行（即能得到问题的解）。如果不可行，则跳过对该节点为根的子树的搜索，逐层向其祖先节点回溯；否则，进入该子树，继续按深度优先策略搜索。  回溯法的基本行为是搜索，搜索过程中使用剪枝函数来避免无效的搜索。剪枝函数包括两类：1. 使用约束函数，剪去不满足约束条件的路径；2.使用限界函数，剪去不能得到最优解的路径。  虽然回溯法看起来时间复杂度很高，但是因为有剪枝函数，所以实际的运行时间并不长，甚至小于时间复杂度为O(nc)(c为背包容量)的动态规划算法。（下图为knapsack函数优化后）    问题的关键在于如何定义问题的解空间，转化成树（即解空间树）。解空间树分为两种：子集树和排列树。两种在算法结构和思路上大体相同。  经过这次实验，我对于用回溯法求解0-1背包问题的相关代码已基本熟悉，算法知识得到了复习与巩固。在写代码与调试的过程中，在解决问题过程中，丰富了个人编程的经历和经验，提高了个人解决问题的能力。 | | | | | |
| 实验得分 |  | 助教签名 | |  | | |

**附录：完整代码**

*//回溯法实现01背包*

#include <iostream>

#include <fstream>

#include <windows.h>

#include <stdio.h>

using namespace std;

#define MAX 2000

int n;*//物品数量*

int c = 500;*//背包容量*

double v[MAX];*//各个物品的价值*

double w[MAX];*//各个物品的重量*

int cw = 0;*//当前背包重量*

int cp = 0;*//当前背包中物品价值*

int bestp = 0;*//当前最优价值*

double perp[MAX];*//单位物品价值排序后*

double order[MAX];*//物品编号*

int put[MAX] = {0};*//设置是否装入*

int bestway[MAX] = {0};*//回溯最优路径*

void swap(double \*a, double \*b)

{

    double temp = \*a;

    \*a = \*b;

    \*b = temp;

}

*//按单位价值排序*

void knapsack()

{

    int i, j;

    for (i = 1; i <= n; i++)

        perp[i] = v[i] / w[i];

    int k = 0;

    for (i = 1; i <= n - 1; i++)*//选择排序perp[],order[],sortv[],sortw[]*

    {

        k = i;

        for (j = i + 1; j <= n; j++)

            if (perp[k] < perp[j])

                k = j;

        swap(&perp[i], &perp[k]);

        swap(&order[i], &order[k]);

        swap(&v[i], &v[k]);

        swap(&w[i], &w[k]);

    }

}

*//计算上界函数*

int bound(int i)

{

    int leftw = c - cw;

    int 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;

}

*//回溯函数*

void backtrack(int i)

{

    if (i > n)

    {

        bestp = cp;

        for (int i = 1; i <= n; i++)

            bestway[i] = put[i];

        return;

    }

    if (cw + w[i] <= c)

    {

        cw += w[i];

        cp += v[i];

        put[i] = 1;

        backtrack(i + 1);

        cw -= w[i];

        cp -= v[i];

        put[i] = 0;

    }

    if (bound(i + 1) > bestp)*//符合条件搜索右子树*

        backtrack(i + 1);

}

int main()

{

    int i;

    ifstream infile("01beibao\_in.txt", ios::in);

    ofstream outfile("01beibao\_out.txt", ios::out);

    infile >> n;

    for (i = 1; i <= n; i++)

    {

        infile >> w[i] >> v[i];

        order[i] = i;

    }

    infile.close();

    double time = 0;

    LARGE\_INTEGER nFreq, nBeginTime, nEndTime;

    QueryPerformanceFrequency(&nFreq);

    QueryPerformanceCounter(&nBeginTime);*//开始计时*

    knapsack();

    backtrack(1);

    QueryPerformanceCounter(&nEndTime);*//停止计时*

    time = (double)(nEndTime.QuadPart - nBeginTime.QuadPart) / (double)nFreq.QuadPart;*//计算程序执行时间单位为s*

    outfile << "最大价值为：" << bestp << "\n";

    outfile << "需要装入的物品是：\n";

    outfile << "序号 重量 价值\n";

    for (i = 1; i <= n; i++)

    {

        if (bestway[i] == 1)

            outfile << order[i] << "    " << w[i] << "    " << v[i] << endl;

    }

    outfile << "程序耗时" << time \* 1000 << "ms" << endl;

    outfile.close();

    return 0;

}

*//01背包数据生成器*

#include <iostream>

#include <fstream>

#include <time.h>

#define random(a, b) (rand() % (b - a + 1) + a)

using namespace std;

int main()

{

    int t;

    cin >> t;

    ofstream outfile("01beibao\_in.txt", ios::out);

    outfile << t << endl;

    srand((int)time(NULL));

    for (int i = 1; i <= 2 \* t; i++)

    {

        outfile << random(1, 100) << " ";

        if (i % 20 == 0)

            outfile << endl;

    }

    outfile.close();

    return 0;

}