**算法分析与设计实验报告**

**第 3 次实验**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 邹林壮 | 学号 | 202208040412 | | 班级 | 计算机科学与技术（拔尖班）2201 |
| 时间 | 2024.11.23 | 地点 | 院楼432 | | | |
| 实验名称 | 回溯法求解 0-1 背包 | | | | | |
| 实验目的 | 通过本实验，深入理解和掌握回溯法的设计思想，特别是如何通过回溯法解决0-1背包问题。同时，通过实际编程练习，提高对C++语言的运用能力，以及对算法性能的分析能力。 | | | | | |
| 实验原理 | 0-1背包问题是一个经典的优化问题，目标是在不超过背包容量的情况下，选择物品使得总价值最大。回溯法是一种通过试错来寻找问题解的方法，它尝试分步解决问题的每个部分，并且在过程中剪枝，以减少不必要的搜索。并且能够根据不同的输入用例，能准确的输出用例中的值，并计算出程序运行所需要的时间。 | | | | | |
| 实验步骤 | 1. **确定输入输出**   输入为物品件数n，每件物品的质量w，价值v，背包的容量c,  输出为装入物品的最大价值bestv,  约束是容量c，且使用回溯，也就是约束函数   1. **确定限界函数**   在解空间树的当前扩展节点处，仅当要进入右子树时才计算限界bound，以判断是否将右子树减去。进入左子树时不需要计算上界，其上界与父节点的上界相同。  我设置了两个限界函数进行实现。  限界函数1：仅简单地将后续物品的价值与当前价值相加，与bestv进行比较从而实现剪枝。   |  | | --- | | int bound(int i){  ll nv=cv;  for(int j=i;j<=n;j++){  nv+=oj[j].v;  }  return nv;  } |   限界函数2：将剩下的物品按照单位价值进行排序，按照分数背包的解决思路，求得剩下物品的最大价值，与当前价值相加后与bestv进行比较从而实现剪枝。   |  | | --- | | int bound(int i){  ll cleft=c-cw;  ll nv=cv;  int j=i;  for(;j<=n&&oj[j].w<=cleft;j++){  cleft-=oj[j].w;  nv+=oj[j].v;  }  if(j<=n)nv+=oj[j].v\*cleft/oj[j].w;  return nv;  } |  1. **回溯法求解**   实现一个递归函数，尝试将每个物品放入背包中，进行回溯后，再去掉该物品，判断是否可以进行右子树，从而再次回溯。直到回溯到达叶子节点，进行bestv的更新。  ④**读取输入数据**  从文件data.txt中读取物品的价值和重量。  计算最大价值：调用函数计算在给定背包容量下能获得的最大价值，并记录执行时间。  输出结果：将计算结果和执行时间写入文件result.txt，并在控制台输出相关信息。 | | | | | |
| 关键代码 | 1. **void** backtrack(**int** i){ 2. **if**(i>n){ 3. bestv=cv; 4. bs=s; 5. **return**; 6. } 7. **if**(cw+oj[i].w<=c){ 8. cw+=oj[i].w; 9. cv+=oj[i].v; 10. s.push\_back(i); 11. backtrack(i+1); 12. cw-=oj[i].w; 13. cv-=oj[i].v; 14. s.pop\_back(); 15. } 16. **if**(bound(i+1)>bestv){ 17. backtrack(i+1); 18. } 19. } | | | | | |
| 测试结果 | **(1)正确性：**  在洛谷上进行测试，在限界函数比较宽松的情况下，会超时，但是改进限界函数为分数背包的形式时，在时间和空间上均满足题目要求，且数据量达到要求，保证了正确性。  **(2)复杂度：**  时间复杂度：  **1.排序操作**：代码中使用了 sort 函数对物品按照价值密度（单位重量的价值）进行排序，n是物品的数量，则排序的时间复杂度为。  **2.回溯操作**：在最坏情况下，回溯算法会探索所有可能的物品组合。由于使用了贪心策略进行预处理（排序），实际上回溯的深度会大大减少。最坏情况下，回溯的深度为 n（每个物品都可能被考虑是否放入背包）。对于每个深度 i，算法会进行一次 bound 函数的调用，对于每个回溯调用，bound 函数会被调用一次，其时间复杂度为 O(n)，因为它需要遍历所有剩余的物品以计算最大价值,最多有个节点需要计算bound，所以时间复杂度为。  综合以上，总的时间复杂度为。  空间复杂度：  **1.物品数组**：代码中定义了一个大小为N的数组 oj 来存储所有物品的信息，其中 N 是一个常数，表示物品数量的上限，这部分的空间复杂度为 O(N)。  **2.回溯栈**：在回溯过程中，最坏情况下，递归栈的深度可以达到n（每个物品都可能被考虑），递归栈的空间复杂度为O(n)。  **3.辅助向量**：代码中使用了 bs 和 s 两个向量来存储当前的最优解和当前路径。在最坏情况下，这两个向量的大小都不会超过 n。因此，这部分的空间复杂度为 O(n)。  综合以上，总的空间复杂度为 O(n)  **(3)构造了不同规模数据集并进行图像化演示：**  采用不同规模和大小的数据        **运行时间趋势**：  1.小规模和中规模数据中，时间几乎为 0，显示优化效果良好。  2.大规模数据中，满足时间复杂度的趋势。  **最优解趋势**：  1.随物品数量增多，最优解的总价值显著提高，符合预期。  2.更大规模数据可以获得更优解，但计算成本也相应增加。 | | | | | |
| 实验心得 | 1.通过本次实验，我对回溯法算法的设计与分析方法有了更深刻的理解。  2.学会了通过回溯法（深度优先搜索）来解决0-1背包问题，并学会使用剪枝来减小不必要的搜索，从而提高搜索效率。  3.这扩展了我解决问题的思路，通过合理的形式遍历解空间，并通过一定方法来缩小搜索空间，这不仅提高了我的编程能力，也增强了我对算法性能分析的兴趣和信心。 | | | | | |
| 实验得分 |  | 助教签名 | |  | | |

**附录：完整代码**

**1.使用限界函数1**

|  |
| --- |
| **#include<bits/stdc++.h>**  **using namespace std;**  **#define ll long long**  **const int N=1e6+10;**  **struct object{**  **ll w;**  **ll v;**  **};**  **object oj[N];**  **ll n;**  **ll c;**  **ll cw=0,cv=0;**  **ll bestv=0;**  **int bound(int i){**  **ll nv=cv;**  **for(int j=i;j<=n;j++){**  **nv+=oj[j].v;**  **}**  **return nv;**  **}**  **void backtrack(int i){**  **if(i>n){**  **bestv=cv;**  **return;**  **}**  **if(cw+oj[i].w<=c){**  **cw+=oj[i].w;**  **cv+=oj[i].v;**  **backtrack(i+1);**  **cw-=oj[i].w;**  **cv-=oj[i].v;**  **}**  **if(bound(i+1)>bestv){**  **backtrack(i+1);**  **}**  **}**  **int main(){**  **cin>>c>>n;**  **int w,v;**  **for(int i=1;i<=n;i++){**  **cin>>w>>v;**  **oj[i].w=w;**  **oj[i].v=v;**  **}**  **backtrack(1);**  **cout<<bestv<<endl;**  **}** |

**2.使用限界函数2**

|  |
| --- |
| **#include<bits/stdc++.h>**  **using namespace std;**  **#define ll long long**  **const int N=1e6+10;**  **struct object{**  **ll w;**  **ll v;**  **int pos;**  **double p;**  **};**  **bool cmp(const object &a,const object &b){**  **return a.p>b.p;**  **}**  **object oj[N];**  **ll n;**  **ll c;**  **ll cw=0,cv=0;**  **ll bestv=0;**  **vector<int> bs,s;**  **int bound(int i){**  **ll cleft=c-cw;**  **ll nv=cv;**  **int j=i;**  **for(;j<=n&&oj[j].w<=cleft;j++){**  **cleft-=oj[j].w;**  **nv+=oj[j].v;**  **}**  **if(j<=n)nv+=oj[j].v\*cleft/oj[j].w;**  **return nv;**  **}**  **void backtrack(int i){**  **if(i>n){**  **bestv=cv;**  **bs=s;**  **return;**  **}**  **if(cw+oj[i].w<=c){**  **cw+=oj[i].w;**  **cv+=oj[i].v;**  **s.push\_back(i);**  **backtrack(i+1);**  **cw-=oj[i].w;**  **cv-=oj[i].v;**  **s.pop\_back();**  **}**  **if(bound(i+1)>bestv){**  **backtrack(i+1);**  **}**  **}**  **int main(){**  **cin>>c>>n;**  **int w,v;**  **for(int i=1;i<=n;i++){**  **cin>>w>>v;**  **oj[i].w=w;**  **oj[i].v=v;**  **oj[i].pos=i;**  **oj[i].p=(double)v/(double)w;**  **}**  **sort(oj+1,oj+n+1,cmp);**  **backtrack(1);**  **cout<<bestv<<endl;**  **for(int i:bs){**  **cout<<oj[i].pos<<" ";**  **}**  **}** |

**3.数据测试**

|  |
| --- |
| **#include <bits/stdc++.h>**  **#include <chrono> // 引入 chrono 库**  **using namespace std;**  **#define ll long long**  **const int N = 1e6 + 10;**  **struct Object {**  **ll weight, value, index;**  **double ratio;**  **};**  **bool compare(const Object &a, const Object &b) {**  **return a.ratio > b.ratio;**  **}**  **Object objects[N];**  **ll c, n, current\_weight = 0, current\_value = 0, best\_value = 0;**  **vector<ll> best\_solution, temp\_solution;**  **// 生成测试数据**  **void generate\_test\_data(const string &filename, ll num\_items, ll max\_weight, ll max\_value, ll capacity) {**  **ofstream out(filename);**  **if (!out) {**  **cerr << "Error: Unable to open file " << filename << endl;**  **return;**  **}**  **out << capacity << " " << num\_items << "\n"; // 背包容量和物品数量**  **for (ll i = 0; i < num\_items; i++) {**  **ll weight = rand() % max\_weight + 1; // 随机重量**  **ll value = rand() % max\_value + 1; // 随机价值**  **out << weight << " " << value << "\n";**  **}**  **out.close();**  **cout << "Generated data for " << filename << " with " << num\_items << " items.\n";**  **}**  **// 计算上界**  **ll calculate\_bound(ll i) {**  **ll remaining\_capacity = c - current\_weight;**  **ll potential\_value = current\_value;**  **ll j = i;**  **for (; j <= n && objects[j].weight <= remaining\_capacity; j++) {**  **remaining\_capacity -= objects[j].weight;**  **potential\_value += objects[j].value;**  **}**  **if (j <= n) {**  **potential\_value += objects[j].value \* remaining\_capacity / objects[j].weight;**  **}**  **return potential\_value;**  **}**  **// 回溯搜索**  **void backtrack(ll i) {**  **if (i > n) {**  **best\_value = current\_value;**  **best\_solution = temp\_solution;**  **return;**  **}**  **if (current\_weight + objects[i].weight <= c) {**  **current\_weight += objects[i].weight;**  **current\_value += objects[i].value;**  **temp\_solution.push\_back(objects[i].index);**  **backtrack(i + 1);**  **current\_weight -= objects[i].weight;**  **current\_value -= objects[i].value;**  **temp\_solution.pop\_back();**  **}**  **if (calculate\_bound(i + 1) > best\_value) {**  **backtrack(i + 1);**  **}**  **}**  **// 测试数据**  **void test\_data(const string &filename) {**  **ifstream in(filename);**  **if (!in) {**  **cerr << "Error: Unable to open file " << filename << endl;**  **return;**  **}**  **// 加载数据**  **in >> c >> n;**  **for (ll i = 1; i <= n; i++) {**  **in >> objects[i].weight >> objects[i].value;**  **objects[i].index = i;**  **objects[i].ratio = (double)objects[i].value / objects[i].weight;**  **}**  **in.close();**  **// 排序物品**  **sort(objects + 1, objects + n + 1, compare);**  **// 计时并运行算法**  **auto start\_time = chrono::high\_resolution\_clock::now();**  **backtrack(1);**  **auto end\_time = chrono::high\_resolution\_clock::now();**  **// 输出结果**  **cout << "Test for " << filename << ":\n";**  **cout << "Best value: " << best\_value << "\n";**  **cout << "Time taken: "**  **<< chrono::duration\_cast<chrono::milliseconds>(end\_time - start\_time).count()**  **<< " milliseconds\n\n";**  **}**  **int main() {**  **srand(time(0)); // 设置随机种子**  **// 生成测试数据**  **generate\_test\_data("01knap\_backtrack\_small.txt", 10, 100, 200, 500); // 小规模**  **generate\_test\_data("01knap\_backtrack\_medium.txt", 10000, 500, 1000, 10000); // 中规模**  **generate\_test\_data("01knap\_backtrack\_large.txt", 1000000, 1000, 2000, 1000000); // 大规模**  **cout << "\nData generation complete.\n\n";**  **// 测试小规模数据**  **test\_data("01knap\_backtrack\_small.txt");**  **// 测试中规模数据**  **test\_data("01knap\_backtrack\_medium.txt");**  **// 测试大规模数据**  **test\_data("01knap\_backtrack\_large.txt");**  **return 0;**  **}** |