杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

程序设计实践报告

基于 C 语言的背包问题回溯法解题报告

卓越学院
计算机科学英才班
周靖凯

目录

1 引言

背包问题作为计算机科学中的经典问题，长期以来吸引了众多研究者的关注。它不仅在理论计算领域具有重要地位，而且在实际应用中极具价值，如在资源分配、财务规划等多个领域均有广泛应用。传统的背包问题侧重于在重量或体积限制下的价值最大化，而现实世界中的问题往往更为复杂，涉及多个限制维度，如重量、体积及物品件数等。

多维度背包问题增加了额外的限制条件，如物品的件数限制，使得问题更加复杂。这种增加的复杂性不仅挑战了传统算法的有效性，也促使研究者探索更为高效的解决方案。

本文介绍了一种解决增加了物品件数限制的多维度背包问题的算法。我们采用了深度优先搜索（DFS）的方法，通过递归探索所有可能的物品组合，寻找在给定重量、体积和件数限制下的最优解。本研究的主要贡献在于提出了一种简单直观的方法来处理这一复杂问题，并通过剪枝策略优化了算法的效率。

本文首先介绍了算法的核心思路和实现细节，随后进行了性能分析，探讨了算法在不同情景下的表现和潜在的优化空间。最后，我们对算法的实际应用价值和未来的改进方向进行了讨论。

2 问题描述

本问题是经典背包问题的一个变种，考虑物品的重量、体积和价值三个维度。给定 n 种物品，每种物品的重量为 w_i，体积为 v_i，价值为 p_i。背包的最大装重量为 W，最大体积为 V，物品不能超过件数 C。目标是选择一组物品放入背包，使得装入的物品总价值最大化，同时不超过背包的重量和体积限制。

3 算法思路

3.1 深度优先搜索（DFS）

算法的核心是使用深度优先搜索（DFS）策略。这种方法通过递归地探索所有可能的物品组合，以寻找最优解。在每一步，算法决定是否将当前物品加入背包，并相应地更新当前的总重量、总体积和总价值。

3.2 递归函数结构

递归函数 `dfs` 接受当前考虑的物品集合、背包的重量和体积限制、当前总价值、剩余可加入的物品件数以及当前考虑的物品索引。函数在考虑完所有物品后返回当前总价值。对于每件物品，算法考虑不加入该物品的情况以及在满足重量、体积和件数限制的情况下加入该物品的情况，并返回这两种选择中总价值较大的一种。

3.3 物品件数限制

与传统背包问题相比，本算法在每次递归调用时减少剩余可加入物品的件数，以确保在达到最大件数限制时停止添加新物品。

3.4 算法实现

在 `main` 函数中，算法首先读取物品的数量、背包的重量和体积限制以及最大可携带物品件数。然后，为每件物品分配空间并读取其重量、体积和价值。最后，调用 `knapsack` 函数，该函数进一步调用 `dfs` 函数计算并输出背包中物品的最大总价值。

4 性能分析

4.1 深度优先搜索（DFS）的复杂度

本算法的核心是深度优先搜索（DFS）。在最坏情况下，DFS 的时间复杂度为 $O(2^n)$，其中 n 是物品的数量。这种复杂度源于对每件物品加入或不加入背包的决定。然而，在实际应用中，由于背包的重量、体积限制或物品件数限制，搜索可能会提前终止，因此实际性能可能优于理论最大复杂度。

4.2 递归调用的影响

算法使用递归来实现 DFS，递归调用的深度可能对性能产生显著影响。在物品数量较多的情况下，递归深度的增加可能导致栈溢出或性能下降。优化递归调用或考虑迭代方法可能有助于提高性能。

4.3 剪枝优化

算法中的剪枝步骤有效减少了搜索空间。当当前组合的重量或体积超过限制，或者可加入的物品件数为零时，算法将停止进一步搜索。这种策略有助于提高算法的整体效率。

4.4 内存使用

由于每次递归调用都可能产生新的变量实例，算法的内存使用随递归深度的增加而增加。在处理大规模数据时，可能导致显著的内存使用。优化数据结构和减少变量复制可以降低内存消耗。

4.5 综合评估

总体来说，尽管该算法理论上可以解决多维度背包问题，但在处理大量物品时可能面临性能和内存使用方面的挑战。通过实施剪枝策略和优化内存使用，可以在一定程度上提升实际性能。对于大规模问题，可能需要更高效的算法，如动态规划或贪心算法。

5 小结

本文介绍的算法旨在解决增加了物品件数限制的多维度背包问题。通过实现深度优先搜索（DFS）策略，算法能够探索所有可能的物品组合，以寻找在满足重量、体积和件数限制条件下的最大价值组合。

5.1 算法的优势

算法的主要优势在于其简洁性和直观性。通过递归实现，算法能够清晰地表达问题的决策过程。此外，算法中的剪枝步骤有助于提高效率，特别是在部分组合明显不符合条件时，能够快速排除。

5.2 潜在的改进

尽管当前实现在小到中等规模的数据集上表现良好，但在处理大规模数据集时，性能和内存使用仍有改进空间。未来的工作可以集中在优化递归调用、减少内存消耗，或者探索更高效的算法，如动态规划。

A 附录

A.1 完整测试代码

knapsack.c

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    int w, v, p;
} Goods;

int dfs(const Goods a[], const int n, const int retw,
   const int retv, const int sump, const int retc, const
   int cur)
{
    if (cur >= n)
        return sump;
    int res = dfs(a, n, retw, retv, sump, retc, cur + 1);
    if (retw >= a[cur].w && retv >= a[cur].v && retc >= 1)
    {
        int now = dfs(a, n, retw - a[cur].w, retv - a[cur
           ].v, sump + a[cur].p, retc - 1, cur + 1);
        if (now > res)
            res = now;
    }
    return res;
}

int knapsack(const Goods a[], const int n, const int w,
   const int v, const int c)
```

```
{
    return dfs(a, n, w, v, 0, c, 0);
}

int main()
{
    int n, w, v, c;
    scanf("%d%d%d%d", &n, &w, &v, &c);
    Goods *a = (Goods *)malloc(sizeof(Goods) * n);
    for (int i = 0; i < n; i++)
        scanf("%d%d%d", &a[i].w, &a[i].v, &a[i].p);
    printf("%d", knapsack(a, n, w, v, c));
    free(a);
    return 0;
}
```