暨南大学本科上机课实验报告

学生姓名：肖健成　　学号：2021103268

学院：网络空间安全　　专业：网络空间安全

课程名称:《算法分析与设计》　　实验名称：用回溯法求解 0-1 背包问题

实验类型：验证-设计-综合　　实验时间：2023 年 11 月 22 日晚　　地点：513

上机内容：2.6

（1）熟悉实验环境，掌握回溯法求解 0-1 背包问题任意解的算法原理；

（2）利用回溯法设计具有数据输入、处理和输出功能的 0-1 背包问题求解算法，实现算法的编译和运行，记录实验过程并整理实验结果；

（3）编写算法实现对 0-1 背包问题求解，基于回溯法的效率评估。

算法描述：

回溯法是一种解决组合优化问题的算法，用于解决 0-1 背包问题。0-1 背包问题的目标是在给定一组物品的重量和价值以及一个背包的最大容量的情况下，选择哪些物品放入背包以最大化总价值，但不能超过背包的容量。以下是回溯法设计的 0-1 背包问题求解算法的详细分析：

算法思路：

定义问题：首先，定义问题的输入和输出。输入包括物品的重量和价值列表以及背包的容量。输出是最大化的总价值和所选的物品组合。

回溯函数：创建一个递归函数，该函数将尝试在每个决策点（每个物品）上做出两种选择：将物品放入背包或不放入背包。函数的参数通常包括当前物品的索引、当前背包的重量、当前总价值以及当前的解（表示哪些物品被选中）。

决策过程：在回溯函数中，对于每个物品，尝试以下两种选择：

将物品放入背包：如果将物品放入背包不会导致背包超过容量限制，则递归调用回溯函数，并将相应的物品标记为已选中。同时，更新当前背包的重量和总价值。

不将物品放入背包：递归调用回溯函数，但不将该物品标记为已选中。

递归终止条件：当递归函数达到以下条件之一时，停止递归：

已经遍历完所有的物品（达到物品的数量上限）。

当前背包的重量达到或超过背包的容量限制。

记录最优解：在回溯过程中，记录当前最大的总价值和对应的物品组合。每当找到新的最大总价值时，更新这些值。

返回结果：在完成回溯过程后，返回最大总价值以及被选中的物品组合。

算法特点：

回溯法是一种穷举法，它会尝试所有可能的组合，因此可以找到最优解。

由于需要尝试所有可能的组合，回溯法可能会在大规模问题上变得非常慢，因此不

适用于非常大的背包问题。

通过合理的剪枝策略（例如，当已选中的物品总价值已经小于当前最大总价值时，可以提前终止分支），可以改进回溯法的性能。

回溯法是一种通用的解决方法，可应用于多种组合优化问题，不仅仅是 0-1 背包问题。

```
function knapsack_backtracking(values, weights, capacity):
    n = length(values)  # 物品数量
    max_value = 0  # 最大总价值
    current_weight = 0  # 当前背包重量
    current_value = 0  # 当前总价值
    current_solution = array of size n initialized with 0  # 当前解，表示哪些物品被选中
    best_solution = array of size n initialized with 0  # 最优解
    solutions = empty list  # 存储所有可行解的列表

    function backtrack(index):
        nonlocal max_value
        if index == n or current_weight == capacity:  # 如果已遍历完所有物品或达到背包容量上限
            if current_value > max_value:  # 如果当前总价值更大
                max_value = current_value  # 更新最大总价值
                best_solution = current_solution.copy()  # 更新最优解
            solutions.append(current_solution.copy())  # 存储当前解到可行解列表
            return

        # 尝试将当前物品放入背包
        if current_weight + weights[index] <= capacity:
            current_solution[index] = 1  # 将物品标记为已选中
            current_weight += weights[index]  # 更新当前背包重量
            current_value += values[index]  # 更新当前总价值
            backtrack(index + 1)  # 递归探索下一个物品
            current_solution[index] = 0  # 回溯：将物品标记为不选中
            current_weight -= weights[index]  # 回溯：恢复背包重量
            current_value -= values[index]  # 回溯：恢复总价值

        # 尝试不将当前物品放入背包
        backtrack(index + 1)  # 递归探索下一个物品

    backtrack(0)  # 从第一个物品开始递归

    return solutions, best_solution, max_value  # 返回所有可行解、最优解和最大总价值
```

源程序：

```
def knapsack_backtracking(values, weights, capacity):
    n = len(values)  # 物品数量
    solutions = []  # 存储所有可行解的列表
    best_solution = None  # 最优解

    def backtrack(index, current_weight, current_value,
current_solution):
        nonlocal max_value, best_solution

        if index == n or current_weight == capacity:  # 到达决策树
底部或达到背包容量
            solutions.append(current_solution[:])  # 存储当前解

            if current_value > max_value:  # 当前解的总价值更大
                max_value = current_value  # 更新最大总价值
                best_solution = current_solution[:]  # 更新最优解
            return

        # 尝试将当前物品放入背包
        if current_weight + weights[index] <= capacity:  # 检查是否
能放入背包
            current_solution[index] = 1  # 将物品标记为已选中
            current_weight += weights[index]  # 更新当前背包重量
            current_value += values[index]  # 更新当前总价值
            backtrack(index + 1, current_weight, current_value,
current_solution)
            current_solution[index] = 0  # 回溯：将物品标记为不选中
            current_weight -= weights[index]  # 回溯：恢复背包重量
            current_value -= values[index]  # 回溯：恢复总价值

        # 尝试不将当前物品放入背包
        backtrack(index + 1, current_weight, current_value,
current_solution)

    max_value = 0  # 初始的最大总价值
    current_solution = [0] * n  # 当前解的标志，0 表示不放入，1 表示
放入
    backtrack(0, 0, 0, current_solution)  # 从第一个物品开始递归
```

```
    return solutions, best_solution, max_value  # 返回所有可行解、最优解和最大总价值

def main():
    n = int(input("请输入物品数量: "))
    capacity = int(input("请输入背包容量: "))

    values = []
    weights = []

    for i in range(n):
        weight, value = map(int, input(f"请输入第 {i+1} 个物品的重量和价值（以空格分隔）: ").split())
        weights.append(weight)
        values.append(value)

    solutions, best_solution, max_value = knapsack_backtracking(values, weights, capacity)

    # 输出结果
    print("\n所有可行解: ")
    for solution in solutions:
        print(solution)

    print("\n最优解:", best_solution)  # 输出最优解
    print("最大价值:", max_value)  # 输出最大总价值

if __name__ == "__main__":
    main()
```

复杂度分析：

时间复杂度分析：

回溯过程的递归树： 0-1 背包问题的回溯法可以看作一棵递归树，其中每个节点表示一个决策点，每个决策点都有两个分支，即选择放入当前物品和选择不放入当前物品。递归树的深度等于物品的数量，每个节点有两个分支，因此总的决策路径数为 2^n，其中 n 是物品的数量。

每个节点的处理时间： 在每个决策点，算法需要执行一系列操作，包括检查是否可以放入当前物品、更新当前背包重量和总价值、递归进入下一层，以及进行回溯操作。这些操作都是常数时间的，不依赖于问题规模。

总体时间复杂度： 由于存在 2^n 个决策路径，每个决策路径上的操作都是常数时间，因此总体时间复杂度是 O(2^n)。这是指数级的复杂度，因此回溯法对于大规模的问题效率较低。

空间复杂度分析：

递归栈空间： 在递归过程中，会生成一个递归调用栈，栈的深度等于物品的数量 n，因为每个决策点对应一个递归函数调用。每个递归函数调用需要存储当前的解、背包重量、总价值等信息，因此每个递归函数调用的空间复杂度是 O(n)。所以，总的递归栈空间复杂度是 O(n)。

其他空间： 在算法中，还需要存储当前最优解、当前可行解等信息，这些数据结构的空间复杂度也是 O(n)。所以，总体空间的复杂度也是 O(n)。

时间复杂度	空间复杂度
O(2^n)	O(n)

程序的运行结果及截图：

下面是我们的测试用例

number＝4，capacity＝7

i	1	2	3	4
w(重量)	3	5	2	1
v(价值)	9	10	7	4

实验结果如下

```
请输入物品数量: 4
请输入背包容量: 7
请输入第 1 个物品的重量和价值（以空格分隔）: 3 9
请输入第 2 个物品的重量和价值（以空格分隔）: 5 10
请输入第 3 个物品的重量和价值（以空格分隔）: 2 7
请输入第 4 个物品的重量和价值（以空格分隔）: 1 4

所有可行解:
[1, 0, 1, 1]
[1, 0, 1, 0]
[1, 0, 0, 1]
[1, 0, 0, 0]
[0, 1, 1, 0]
[0, 1, 0, 1]
[0, 1, 0, 0]
[0, 0, 1, 1]
[0, 0, 1, 0]
[0, 0, 0, 1]
[0, 0, 0, 0]

最优解: [1, 0, 1, 1]
最大价值: 20
```

个人总结：回溯法的时间复杂度较大，面对较大的数据集，动态规划法相对较好。
收获：首次自己实现回溯法求解 0-1 背包问题算法，提高了自己的能力。