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：用分支限界法求解 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：2.7
（1）熟悉实验环境，掌握分支限界法求解 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、进行算法效率的合理评估，最后整理实验结果。

算法描述：

分支限界法是一种用于求解组合优化问题的算法，它通过逐步构造解空间树（搜索树）来找到问题的最优解。在 0-1 背包问题中，我们的目标是在给定的一组物品中选择一些放入背包，使得背包的总重量不超过容量限制，同时总价值最大。

以下是分支限界法求解 0-1 背包问题的详细步骤：

1.初始化： 初始化一个优先队列（通常使用最小堆），将初始节点加入队列。初始节点是一个空节点，表示空背包。

2.循环： 从队列中取出一个节点进行处理，直到队列为空。每次处理一个节点时，进行以下步骤：

a. 计算上界： 计算当前节点的上界（也称为松弛值），以便判断是否值得扩展该节点。上界的计算方式根据问题的性质而定。对于 0-1 背包问题，通常通过贪心法计算上界。

b. 判断是否值得扩展： 比较计算得到的上界和当前已知的最优解，如果上界小于等于当前最优解，则该节点不值得扩展，丢弃该节点。

c. 生成子节点： 如果节点值得扩展，生成该节点的所有可能的子节点。对于 0-1 背包问题，子节点分为两类：装入当前物品和不装入当前物品。计算这些子节点的上界，并将它们加入队列中。

d. 更新最优解： 如果生成的子节点包含更好的解，则更新当前已知的最优解。

3.结束： 当队列为空或者无法生成更优的解时，算法结束。此时，已经找到了问题的最优解。

在 0-1 背包问题中，分支限界法的优势在于它能够通过上界的计算，有效地剪枝，减少搜索空间，从而提高求解效率。

伪代码：

```
class Node:
    int ubound      // 上界
    int value       // 当前节点的价值
    int weight      // 当前节点的重量
    int layer       // 当前节点在搜索树中的层数
    Node parent     // 当前节点的父节点

function calculate_upper_bound(node):
    // 计算节点的上界，根据问题特性实现
    // 对于 0-1 背包问题，可以使用贪心法计算上界
    // 返回上界值

function branch_and_bound_knapsack(num_items, capacity, weights, values):
    priority_queue = PriorityQueue()    // 优先队列，使用最小堆实现
    first_node = Node(ubound=calculate_upper_bound(empty_node), value=0,
weight=0, layer=0, parent=None)
    priority_queue.enqueue(first_node)

    max_value = 0    // 记录当前的最大价值
    max_node = None   // 记录当前的最优解节点

    while not priority_queue.isEmpty():
        current_node = priority_queue.dequeue()    // 弹出队列中的第一个节点

        if current_node.weight + weights[current_node.layer] <= capacity:
            // 计算左孩子节点（装入当前物品）
            left_node = Node(ubound=calculate_upper_bound(left_child_node),
                             value=current_node.value                    +
values[current_node.layer],
                             weight=current_node.weight                  +
weights[current_node.layer],
                             layer=current_node.layer + 1,
                             parent=current_node)
            if left_node.layer < num_items:
                priority_queue.enqueue(left_node)

            if max_value < left_node.value:
                max_value = left_node.value
                max_node = left_node

        if current_node.ubound > max_value:
            // 计算右孩子节点（不装入当前物品）
            right_node = Node(ubound=calculate_upper_bound(right_child_node),
                              value=current_node.value,
```

```
                                    weight=current_node.weight,
                                    layer=current_node.layer + 1,
                                    parent=current_node)
                if right_node.layer < num_items:
                    priority_queue.enqueue(right_node)

        return max_value, max_node
```

源程序：

```
class KnapsackSolver:
    def __init__(self, num, cap):
        # 初始化背包问题求解器
        self.goods = self.get_goods(num)  # 获取物品列表
        self.num = num  # 物品数量
        self.cap = cap  # 背包容量
        self.x = [0] * num  # 初始化解空间，记录最终选择的物品

    class Node:
        # 节点类，表示搜索树中的节点
        def __init__(self, ubound=0, value=0, weight=0, layer=0, parent=None):
            # 初始化节点的属性
            self.ubound = ubound  # 节点的上界
            self.value = value  # 当前节点的价值
            self.weight = weight  # 当前节点的重量
            self.layer = layer  # 当前节点在搜索树中的层数
            self.parent = parent  # 当前节点的父节点

    def get_goods(self, num):
        # 获取物品列表，按照单位重量的价值降序排列
        goods = []
        for i in range(num):
            goods.append(list(map(int, input(f'请输入第{i + 1}个物品的重量和价值，用空格隔开：').split())))
        goods.sort(key=lambda y: (y[1] / y[0]))  # 按照单位重量的价值降序排列
        goods.reverse()
        return goods

    def max_bound(self, node):
```

```
        # 计算节点的上界
        b_value = node.value
        rest_capacity = self.cap - node.weight
        layer = node.layer

        while layer < self.num and rest_capacity > 
self.goods[layer][0]:
            b_value += self.goods[layer][1]
            rest_capacity -= self.goods[layer][0]
            layer += 1

        if rest_capacity != 0 and layer < self.num:
            b_value += rest_capacity * (self.goods[layer][1] / 
self.goods[layer][0])

        return b_value

    def breadth_first_search(self):
        # 利用分支限界法求解 0-1 背包问题
        priority_queue = []  # 构造节点的优先队列
        first_node = self.Node()  # 创建根节点
        first_node.ubound = self.max_bound(first_node)  # 计算根节
点的上界

        priority_queue.append(first_node)
        max_value = 0  # 保存当前的最大价值
        max_node = first_node

        while len(priority_queue) != 0:
            current_node = priority_queue.pop(0)  # 弹出队列中的第
一个节点

            if current_node.weight + 
self.goods[current_node.layer][0] <= self.cap:
                # 计算左孩子节点
                left_node = self.Node(weight=current_node.weight + 
self.goods[current_node.layer][0],
                                      value=current_node.value + 
self.goods[current_node.layer][1],
                                      layer=current_node.layer + 1, 
parent=current_node)
                left_node.ubound = self.max_bound(left_node)

                if left_node.layer < self.num:
```

```
                    priority_queue.append(left_node)

                if max_value < left_node.value:
                    max_value = left_node.value
                    max_node = left_node

            if current_node.ubound > max_value:
                # 计算右孩子节点
                right_node = self.Node(weight=current_node.weight,
value=current_node.value,
                                        layer=current_node.layer +
1, parent=current_node)
                right_node.ubound = self.max_bound(right_node)

                if right_node.layer < self.num and
right_node.ubound > max_value:
                    priority_queue.append(right_node)

        current_node = max_node

        while current_node:
            temp_value = current_node.value
            current_node = current_node.parent

            if current_node and current_node.value != temp_value:
                # 此父节点存在且价值不同，意味着进行了装填
                self.x[current_node.layer - 2] = 1  # 构造最优解

        return max_value

    def print_result(self):
        # 打印最终结果
        print(f"最大价值为：{self.breadth_first_search()}")
        print(f'选取方案为：{[i + 1 for i in range(self.num) if
self.x[i]]}')


if __name__ == '__main__':
    # 主程序入口
    num, cap = map(int, input('请输入物品数量和背包容积，用空格隔
开：').split())
    knapsack_solver = KnapsackSolver(num, cap)
    knapsack_solver.print_result()
```

复杂度分析：
时间复杂性：限界函数时间复杂度为 O(n),而最坏情况有 2^(n ＋1) – 2个节点，若对每个节点用限界函数判断，则其时间复杂度为 O(n*2^n).而算法中时间复杂度主要依赖限界函数，则算法的时间复杂度为 O(n*2^n)。
空间复杂度：
1.节点空间： 搜索树中的节点数量是指数级别的，因此节点的空间复杂度为 O(2^n)。
2.优先队列空间： 优先队列中存储搜索树中的节点，因此优先队列的空间复杂度也为 O(2^n)。
3.其他空间： 其他空间复杂度主要来自一些临时变量和数组，例如保存物品信息、计算上界的过程中使用的空间等。这些空间通常是线性的，因此为 O(n)。
综合考虑，分支限界法求解 0-1 背包问题的总空间复杂度为 O(2^n)。

时间复杂度	空间复杂度
O(n*2^n)	O(2^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

下面是我们的输出结果，输出结果正确，实验结果正确。

```
请输入物品数量和背包容积，用空格隔开：4 7
请输入第1个物品的重量和价值，用空格隔开：3 9
请输入第2个物品的重量和价值，用空格隔开：5 10
请输入第3个物品的重量和价值，用空格隔开：2 7
请输入第4个物品的重量和价值，用空格隔开：1 4
最大价值为：20
选取方案为：[1, 3, 4]
```

个人总结：

通过这次实验，我深入理解了分支限界法的基本原理，并学会了如何将其应用于解决实际问题。实现算法的过程中，我提高了编程，数据结构实现和算法设计的能力。在有理论基础的情况下加以代码实现，加深了我对算法原理的认识。

与此同时，我还理解了分支限界法与回溯法求解 01 背包问题的区别：

1. 定义:

回溯法： 回溯法是一种通过逐步试探和回退的方式搜索解空间的算法。在搜索过程中，通过深度优先搜索策略，尝试每一种可能的解，并在发现不可行解或者找到解时进行回退，继续搜索其他可能的分支。

分支限界法： 分支限界法也是一种搜索解空间的算法，但与回溯法不同，它通过剪枝策略避免了对显然不会产生最优解的分支的搜索。分支限界法在搜索过程中维护一个上界，当发现某一分支的上界小于当前已知的最优解时，就剪掉该分支。

2. 搜索过程:

回溯法： 回溯法采用深度优先搜索，逐步试探每一种可能的解。在搜索的过程中，会记录当前的状态，如果发现当前路径不可行，则回溯到上一个状态重新选择。

分支限界法： 分支限界法也使用深度优先搜索，但在搜索的过程中会根据问题的性质，通过计算上界的方式剪掉一些分支。这样，分支限界法能够更有效地减少搜索空间，提高搜索效率。

3. 剪枝策略:

回溯法： 回溯法通常没有显式的剪枝策略，而是通过在搜索过程中的状态判断来进行回溯。如果当前状态不满足问题的约束条件，就会回退到上一个状态重新选择。

分支限界法： 分支限界法通过计算上界，并与当前已知的最优解进行比较，决定是否继续搜索某一分支。当某一分支的上界小于等于当前最优解时，就可以剪掉该分支，

不再继续搜索。

4. 效率:

回溯法： 回溯法通常需要搜索整个解空间，因此在某些情况下可能会进行大量的无效搜索，效率较低。

分支限界法： 分支限界法通过剪枝策略能够有效减少搜索空间，提高搜索效率。在处理一些大规模问题时，分支限界法相对于回溯法通常具有更好的性能。

5. 应用场景:

回溯法： 回溯法适用于那些解空间较小，或者解空间中的可行解分布较为均匀的问题。

分支限界法： 分支限界法适用于那些解空间庞大，但通过一些特定的性质可以剪掉大量分支的问题，提高搜索效率。