算法实验3

邓一川 pb19000050

问题设定

0-1背包问题： 假设有 n 个物品和一个容量为 C 的背包，和物品的重量 $\{w_i\}_{i\in[n]}$，以及物品的价值 $\{v_i\}_{i\in[n]}$，求解如何选择物品，才能使装进背包的物品价值总和最大。

本次实验我们要求使用以下方法解决0-1背包问题：

- 分治法（Divide-and-conquer）
- 动态规划法（Dynamic Programming）
- 贪心算法（Greedy Method）
- 回溯法（Back-Tracking Method）
- 分支限界法（Branch-and-Bound Method）
- 蒙特卡洛算法（Monte Carlo Method）

算法简述

- 分治法（Divide-and-conquer）

  在分治法中，我们对于“是否取用物品”建立一个二叉搜索树，每一层表示一个物品，每个节点的分叉表示选择将这个物品加入背包与否。对该二叉树进行深度优先搜索，每访问一个节点就会记录当前的路径，从而记录当前取用的物品，然后分别递归访问左右子树（若当前访问的物品重量大于背包剩余容量则仅访问右子树），记录得到分别得到的最大价值和对应取用的物品集合，再将这个最优值和最优集合返回。当访问至叶子节点时，将总记录和总价值返回，如此递归，当返回至根节点时，我们就得到了全局最优解。

- 动态规划法（Dynamic Programming）

  在动态规划法中，我维护一个两个二维数组 `dp[0..n][0..C]` 和 `package_list[0..n][0..C]`，元素 `dp[i][j]` 表示包含前 `i` 个物品，背包容量为 `j` 的字问题的最优价值，而 `package_list[i][j]` 记录了对应的最优序列。当我们对第 `[i][j]` 个子问题进行访问时，我们进行以下操作：

  - 若当前物品价值大于背包剩余容量，则该问题的解与第 `[i-1][j]` 个子问题相同；
  - 否则，研究将该物品装入背包后的总价值，该价值为 `value[i] + dp[i - 1][j - weight[i]]`，将这个值与第 `[i-1][j]` 个子问题的最优值作比较，我们就可以得到最优解法是装入当前物品与否，然后进行对应的最优值和最优序列的记录即可。

- 贪心算法（Greedy Method）

  贪心算法原理非常简单，我们将物品按照单位价值进行排序，然后从前往后装进包里，直到装不下，即可获得一个解，但此解未必最优。

- 回溯法（Back-Tracking Method）

  回溯法在分治法的基础上进行了剪枝操作，我们维护一个序列 `rem_sum[0..n]`，其元素表示第 `[i]` 个物品及其之后的物品价值总和。我们访问到某一个节点时，计算当前已经装入的物品和当前访问物品及其之后物品的价值之和，并将其与目前记录的最优价值进行比较，如果比最有价值小，那么我们将直接退出访问该节点，如此，我们实现了剪枝，简化了算法的复杂度。

- 分支限界法（Branch-and-Bound Method）

在分支限界法中，我们维护一个堆 `queue`，将我们访问过的节点按照最大上界进行排序，由此，当我们访问到某个节点，其可以达到的最优值比我们预估的最优值小，那么我们直接退出该节点，并且从 `queue` 中获取最大上界的节点，直接跳跃至该节点进行访问。

- 蒙特卡洛算法（Monte Carlo Method）

  蒙特卡洛算法是一种随机算法，我们直接将物品序列进行随机排列，然后从前往后逐个装入背包，直到装不下为止。进行多次随机尝试，取其中价值最大的解作为全局解，由此我们可以得到一个解，但未必是最优的。

代码展示

- 分治法（Divide-and-conquer）

```
def divide_conquer_recursion(i=0, remain=C, optimal_list=None):
    if optimal_list is None:
        optimal_list = []

    if i == n:
        return 0, optimal_list

    if goods[i][0] > remain:
        return divide_conquer_recursion(i + 1, remain, optimal_list)  # 
cannot putting it in, so we visit next

    else:  # can put it in
        not_put_it_in, not_in_list = divide_conquer_recursion(i + 1, remain,
                                                               optimal_list)
 # not putting it in, so we visit next
        optimal_list_copy = optimal_list.copy()
        optimal_list_copy.append(i)
        later_values, in_list = divide_conquer_recursion(i + 1,
                                                          remain - goods[i]
[0], optimal_list_copy)  # putting it in
        put_it_in = goods[i][1] + later_values

        if put_it_in > not_put_it_in:
            return put_it_in, in_list
        else:
            return not_put_it_in, not_in_list
```

- 动态规划法（Dynamic Programming）

```
def dynamic_method():
    package_list = [[[] for j in range(C + 1)] for i in range(n + 1)]
    # dp[i][j] means the max value we can get if we put the first i goods in
the bag with capacity j
    dp = [[0 for j in range(C + 1)] for i in range(n + 1)]

    for i in range(1, n + 1):  # i is the number of goods
        for j in range(1, C + 1):  # j is the capacity
            if goods[i - 1][0] > j:  # if the weight of the current good is
larger than the capacity
                package_list[i][j] = package_list[i - 1][j].copy()  # we can
not put the good in
```

```
                dp[i][j] = dp[i - 1][j]  # we can only take the previous one

            else:  # if the weight of the current good is smaller than the capacity
                put_it_in = goods[i - 1][1] + dp[i - 1][j - goods[i - 1][0]]  # we put the good in
                not_put_it_in = dp[i - 1][j]  # we don't put the good in

                if put_it_in > not_put_it_in:  # if we can put the good in
                    package_list[i][j] = package_list[i - 1][j - goods[i - 1][0]].copy()  # we put the good in
                    package_list[i][j].append(i - 1)  # we put the good in
                    dp[i][j] = put_it_in  # we put the good in
                else:  # if we can't put the good in
                    package_list[i][j] = package_list[i - 1][j]  # we don't put the good in
                    dp[i][j] = not_put_it_in  # we don't put the good in

    return dp[n][C], package_list[n][C]  # return the optimal value and the optimal list
```

- 贪心算法（Greedy Method）

```
def greedy_method():
    optimal_list = []
    goods_copy = goods.copy()  # copy the goods list
    goods_copy.sort(key=take_unit_value, reverse=True)  # sort the goods by the unit value
    capacity = C  # the capacity is the same as the bag
    value_all = 0  # the value of all the goods taken into the bag

    for good in goods_copy:  # for each good
        if capacity >= good[0]:  # if the capacity is larger than the weight of the good
            capacity -= good[0]  # reduce the capacity
            value_all += good[1]  # add the value of the good
            optimal_list.append(good[2])  # add the index of the good to the optimal list

    return value_all, optimal_list  # return the optimal value and the optimal list
```

- 回溯法（Back-Tracking Method）

```
def back_tracking():
    goods_copy = goods.copy()  # copy the goods list
    goods_copy.sort(key=take_unit_value, reverse=True)  # sort the goods by the unit value
    rem_sum = [0]  # the sum of the remaining value of the goods

    for i in range(n-1, -1, -1):  # for each good
        rem_sum.insert(0, rem_sum[0] + goods_copy[i][1])  # add the weight of the good to the sum
```

```
    # level is the level of the recursion,
    # cur_weight is the weight of the bag,
    # cur_val is the value of the bag,
    # optimal is the optimal value
    def dfs(level, cur_weight, cur_val, optimal, package_list):
        if level == n:
            return max(optimal, cur_val), package_list

        good = goods_copy[level]
        flag = 0
        package_list_copy = package_list.copy()

        if cur_weight + good[0] <= C:  # if the weight of the bag is larger
than the weight of the good
            package_list_copy.append(good[2])  # add the index of the good
to the optimal list
            optimal, left_list = dfs(level + 1, cur_weight + good[0],
                                     cur_val + good[1], optimal,
package_list_copy)  # we can put the good in
            left_optimal = optimal
            optimal_list = left_list
            flag = 1

        # if the value of the bag is larger than the sum of the remaining
value of the goods
        if cur_val + rem_sum[level + 1] > optimal:
            # we can not put the good in
            optimal, right_list = dfs(level + 1, cur_weight, cur_val,
optimal, package_list)
            if flag == 0:
                return optimal, right_list
            flag = 2

        if flag == 0:
            return optimal, package_list
        elif flag == 1:
            return optimal, left_list
        else:
            if left_optimal >= optimal:
                return left_optimal, left_list
            else:
                return optimal, right_list

    return dfs(0, 0, 0, 0, [])
```

- 分支限界法（Branch-and-Bound Method）

```
def branch_and_bound_method():
    goods_copy = goods.copy()  # copy the goods list
    goods_copy.sort(key=take_unit_value, reverse=True)  # sort the goods by
the unit value

    optimal = 0
    queue = [(0.0, 0, 0, 0, [])]
    heapq.heapify(queue)  # min-root heap
```

```
    return_list = []

    while queue:  # while the queue is not empty
        item = heapq.heappop(queue)  # get the item with the minimum value
        upper_bound, index, cur_weight, cur_val, cur_list = item[:]  # get
the upper bound, the index of the good, the weight of the bag, and the value
of the bag
        upper_bound = - upper_bound  # get the upper bound

        if optimal < cur_val:  # if the upper bound is larger than the
optimal value
            optimal = cur_val
            return_list = cur_list

        if index == n or upper_bound < optimal:  # if the index is the last
good or the upper bound is smaller than the optimal value
            continue  # we can not put the good in

        weight, value, identity = goods_copy[index]  # get the weight and
value of the good
        upper_bound = cur_val + (C - cur_weight) * value / weight  # get the
upper bound

        if cur_weight + weight <= C:  # if the weight of the bag is larger
than the weight of the good
            cur_list_copy = cur_list.copy()  # copy the optimal list
            cur_list_copy.append(identity)  # add the index of the good to
the optimal list
            heapq.heappush(queue, (-upper_bound, index + 1, cur_weight +
weight,
                                   cur_val + value, cur_list_copy))  # put
the good in

        heapq.heappush(queue, (-upper_bound, index + 1, cur_weight, cur_val,
cur_list))  # not put the good in

    return optimal, return_list
```

- 蒙特卡洛算法（Monte Carlo Method）

```
def monte_carlo_method():
    optimal = 0
    best_list = []

    for i in range(100000):  # for each iteration
        good_list = []  # the list of the goods taken into the bag
        random_list = random.sample(range(n), n)  # get a random list of the
goods
        value = 0
        weight = 0
        for j in range(n):  # for each good
            if weight + goods[random_list[j]][0] > C:
                break
            value += goods[random_list[j]][1]
            weight += goods[random_list[j]][0]
```

```
                good_list.append(random_list[j])

            if value > optimal:  # if the value of the bag is larger than the optimal value
                optimal = value
                best_list = good_list

    return optimal, best_list
```

测试结果

对于不同 n 和 C，进行如下尝试：

```
number of goods: 20
Content of package: 950
Max weight: 200
max_value: 100
divide_conquer took 0.4070272445678711 seconds
Optimal value: 732
Optimal list: [0, 1, 3, 5, 6, 10, 11, 14, 15, 16, 18]

dynamic_method took 0.024004220962524414 seconds
Optimal value: 732
Optimal list: [0, 1, 3, 5, 6, 10, 11, 14, 15, 16, 18]

greedy_method took 0.0 seconds
Optimal value: 732
Optimal list: [5, 11, 3, 0, 10, 18, 1, 6, 16, 15, 14]

back_tracking took 0.00400233268737793 seconds
Optimal value: 732
Optimal list: [5, 11, 3, 0, 10, 18, 1, 6, 16, 15, 14]

branch_and_bound_method took 0.0010020732879638672 seconds
Optimal value: 732
Optimal list: [5, 11, 3, 0, 10, 18, 1, 6, 16, 15, 14]

monte_carlo_method took 2.0482213497161865 seconds
Optimal value: 715
Optimal list: [16, 6, 11, 3, 10, 15, 14, 5, 1, 18]
```

```
number of goods: 10
Content of package: 950
Max weight: 200
max_value: 100
divide_conquer took 0.0010004043579101562 seconds
Optimal value: 468
Optimal list: [0, 1, 2, 3, 4, 6, 7, 8, 9]

dynamic_method took 0.013999700546264648 seconds
Optimal value: 468
Optimal list: [0, 1, 2, 3, 4, 6, 7, 8, 9]

greedy_method took 0.0 seconds
```

```
Optimal value: 468
Optimal list: [0, 4, 3, 6, 2, 1, 8, 9, 7]

back_tracking took 0.0 seconds
Optimal value: 468
Optimal list: [0, 4, 3, 6, 2, 1, 8, 9, 7]

branch_and_bound_method took 0.0009989738464355469 seconds
Optimal value: 468
Optimal list: [0, 4, 3, 6, 2, 1, 8, 9, 7]

monte_carlo_method took 1.4659490585327148 seconds
Optimal value: 468
Optimal list: [7, 9, 0, 8, 2, 1, 4, 6, 3]
```