Presentation Script: Branch-and-Bounding Implementation and Application

Branch-and-Bounding 的实现与应用**

1.4 Part 4. Knapsack - Branch and Bound Strategy

Based on the implementation of knapsack_bounding, with the idea of greedy strategy, we can now implement the branh-and-bound version of backracking algorithm. Instead of calculating the 16 total value for every combination of items, we use a bound to estimate the maximum value we can

achieve with the remaining capacity. The bound gives us an upper limit for the total value of a branch (partial solution). If the bound is less than or equal to the current best solution, we prune that branch and do not explore further, considering only the choice of excluding the item then.

- Hello everyone! Today I will explain an efficient algorithm for solving the **01 Knapsack Problem**: **Branch-and-Bounding**.
- 大家好！今天我要为大家讲解一种解决 **01背包问题** 的高效算法：**Branch-and-Bounding。**
- We will build on the branch-and-bound algorithm, combining it with a greedy strategy to optimize the search process.
- 我们将在分支限界算法的基础上，结合贪心策略，通过优化搜索过程。
- I will also use an intuitive example to help you better understand the core ideas of the branch-and-bound algorithm.
- 我还会用一个通俗易懂的例子，帮助大家更好地理解分支限界算法的核心思想。

1. Background: The Basics of Branch-and-Bounding

1. 背景：Branch-and-Bounding 的基本概念

- **Branch-and-Bounding is an optimization algorithm designed to solve the 01 Knapsack Problem.**
- 是一种优化算法，适用于解决 01背包问题。
- **It combines two main ideas:**
- 它结合了两个主要思想：
    1. **Branching**: Recursively explore whether to include or exclude each item, forming a search tree of solutions.
    2. **分支（Branching）**：递归地探索每个物品是否选择（分支为选或不选两种情况），构造解的搜索树。
    3. **Bounding**: Use an "upper bound" (bound) to estimate the potential maximum value of the current solution.

4. **限界（Bounding）**：通过“上界值”（bound）估算当前解的潜在最大价值。

- If the bound is less than or equal to the current best solution, we cut the branch to save unnecessary computation.
- 如果上界值小于等于当前最优解，我们剪枝以节省不必要的计算。

Why Do We Need Branch-and-Bounding?

为什么需要分支限界?

- Without pruning, we would need to explore all possible combinations of items, which grows exponentially with the number of items.
- 如果没有剪枝，我们需要穷举 2^n 种可能的物品组合，计算量会随着物品数量呈指数级增长。
- The goal of **Branch-and-Bounding** is:
- 目标是：
  - Use "upper bound estimation" to avoid invalid branches.
  - 利用“上界估算”避免无效分支。
  - Combine with a greedy strategy to priorite exploring promising branches.
  - 结合贪心策略优先探索更有潜力的分支。
- This greatly improves the efficiency of solving the problem.
- 这大大提高了问题求解的效率。

---

2. Code Implementation and Logic

2. 代码实现与逻辑

Now, let's understand the details of Branch-and-Bounding through its code implementation.

---

here is a python of Code Framework

```
python
def knapsack_branchAndBound(values: list, weights: list, capacity: int) -> list:
    '''
    Arguments:
    - values: the list of values of items
    - values: 每个物品的价值列表
    - weights: the list of weights of items
    - weights: 每个物品的重量列表
    - capacity: the capacity of knapsack
    - capacity: 背包的容量
    Return:
    - optX: the optimal solution
    - optX: 最优的选择方案
    '''
    optP = 0  # Current best profit
    optP = 0  # 当前最优解的利润
    optX = []  # Current best selection
```

```
    optX = []  # 当前最优解的物品选择
    N = len(values)  # Number of items
    N = len(values)  # 物品总数
```

Recursive Function: `knapsack_branchAndBound_recursive`

递归函数：`knapsack_branchAndBound_recursive`

```
python
def knapsack_branchAndBound_recursive(currX: list = []) -> None:
        '''
        Recursive part of the algorithm.
        递归部分。
        Arguments:
        - currX: current solution (item selection)
        - currX: 当前解（物品选择）
        '''
```

- This function is used to explore each branch and calculate whether it is worth further exploration.
- it total have 5 steps.
- 此函数用于探索每个分支，并判断其是否值得进一步探索。

Step 1 is Feasibility Check

步骤1：可行性检查

```
python
currl = len(currX)
        currX_ = currX + [0] * (N - currl)  # Complete the solution to full
length
        currX_ = currX + [0] * (N - currl)  # 补全解到完整长度
        currW = dot(weights, currX_)  # Calculate total weight
        currW = dot(weights, currX_)  # 计算总重量
        currP = dot(values, currX_)  # Calculate total profit
        currP = dot(values, currX_)  # 计算总利润
```

- If all items are considered，then we go thourgh this if:
- 如果所有物品都被考虑：

```
python
if len(currX) == N:
            if currW <= capacity and currP > optP:
                optP = currP  # Update the best profit
                optP = currP  # 更新最优解的价值
                optX = currX[:]  # Update the best selection
                optX = currX[:]  # 更新最优选择方案
            return
```

Step 2 is Bound Calculation and Pruning

步骤2：上界计算与剪枝

```
python
bound = getBound(values, weights, capacity, currX, fractional)
        if bound <= optP:
            return  # Prune the branch if its bound is less than the best
solution
            return  # 如果当前分支的上界值小于最优解，则剪枝
```

Step 3 is Construct Choices**

步骤3：构造选择

```
python
if currW + weights[currI] <= capacity:
            choS = [0, 1]  # Two choices: exclude or include the current item
            choS = [0, 1]  # 两种选择：不选或选当前物品
        else:
            choS = [0]  # Only exclude the current item
            choS = [0]  # 只能选择不选当前物品
```

Step 4 is Sort by Bound

步骤4：按上界排序

```
python
for i in range(len(choS)):
            nextChoices.append(currX[:] + [choS[i]])
            nextBound = getBound(values, weights, capacity, currX + [choS[i]],
fractional)
            nextBounds.append(nextBound)
        if nextBounds[0] <= optP:
            return
```

Step 5 is Recursively Explore

步骤5：递归探索

```
python
for i in range(len(nextChoices)):
            knapsack_branchAndBound_recursive(nextChoices[i])
        knapsack_branchAndBound_recursive([])
```

---

here is an Example to help you better Understanding Through the Practice**

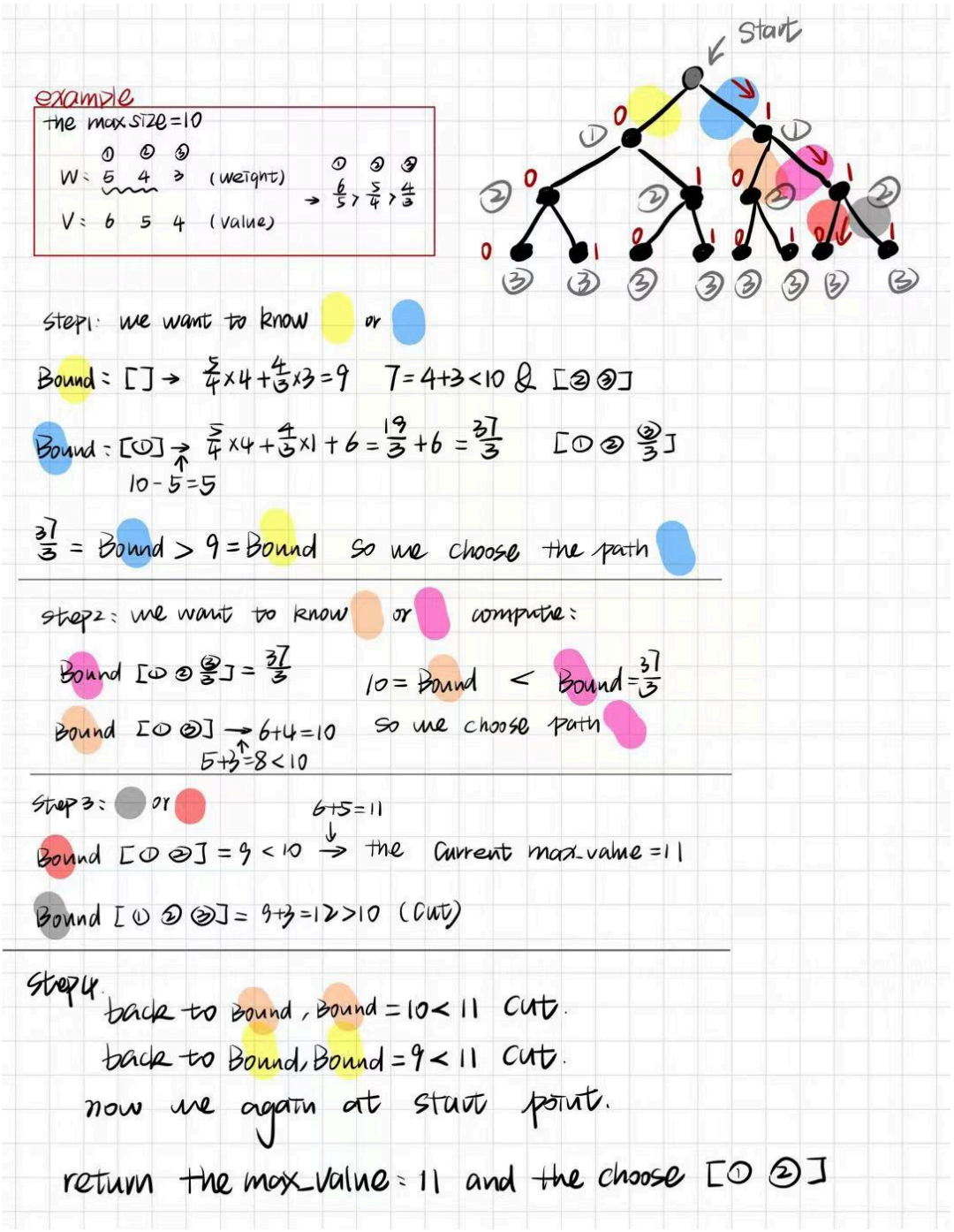
3. 示例：通过实践理解

Problem

问题

- Knapsack capacity: 10
- 背包容量：10
- Items:
- 物品信息：
  - Item 1: weight 5, value 6
  - 物品1：重量5，价值6
  - Item 2: weight 4, value 5
  - 物品2：重量4，价值5
  - Item 3: weight 3, value 4

- 物品3：重量3，价值4



the Optimal Solution is 11

- The final solution is `[1, 1, 0]` with a total value of 11.

Finally we conclude the Algorithm Process

算法过程

1. Start with an empty knapsack.

   从空背包开始。

2. Branch based on whether to include each item.

   根据是否选择物品分支。

3. Use bounds to prune unpromising branches.

使用上界剪枝无望分支。

4. Greedily explore high-potential branches first.

   优先探索高潜力分支。

- Branch-and-Bounding is an efficient optimization algorithm for solving the 01 knapsack problem.
- Branch-and-Bounding 是一种高效解决 01 背包问题的优化算法。
- It combines branching, bounding, and greedy strategy to achieve significant speedup.
- 它结合了分支、限界和贪心策略，大幅提升效率。

Thank you! *