计算机学院

算法导论大作业报告

多平台商品采购优化算法的探索

姓名：付$\boxed{F}$浩

学号：2312180

专业：计算机科学与技术

2025 年 6 月 10 日

目录

本次作业所有代码、测试数据已上传至Github。

1 问题描述

在当前互联网时代，电商平台的多样化为消费者带来了极大的便利，但也带来了新的购物决策难题。不同平台的同一商品价格各异，且每个平台通常会收取固定的配送费用。对于希望节省开支的消费者而言，如何在多个平台之间权衡商品价格与配送费来选择最优的采购方案，成为一个实际且具有挑战性的问题。

现在，给定一组待购商品和若干电商平台（商店），每个平台对不同商品有各自的定价和配送费。消费者希望在不超过 k 个平台下单的前提下，完成所有商品的购买，并使总花费（商品价格之和加所选平台的配送费之和）最小化。其具体问题描述如下：

输入：

- n 个物品（编号 0 到 $n-1$）
- m 个商店（编号 0 到 $m-1$）
- k：最多允许选择的商店数量
- 配送费数组 $\text{delivery}[m]$：$\text{delivery}[j]$ 表示商店 j 的配送费
- 价格矩阵 $\text{prices}[n][m]$：$\text{prices}[i][j]$ 表示物品 i 在商店 j 的价格（-1 表示商店 j 不销售该物品）

输出：

- 最小总花费
- 选择的商店列表
- 每个物品的购买商店分配方案
- 若无解（无法在 k 个商店内购齐所有物品），输出 -1

这个问题兼顾了实际电商购物中的价格敏感性与平台选择限制，具有重要的现实意义和算法挑战，本文我们将采用本学期《算法导论》课程学习的贪心、动态规划等算法来尝试解决该问题，并通过性能分析和实验验证算法的有效性。

2 贪心算法

2.1 算法思路

针对该问题，我们首先采用了“枚举子集 + 贪心分配”的算法策略：通过枚举所有不超过 k 个商店的组合，并在每种组合下采用贪心方式为每个物品分配最低价格的商店，实现配送费与商品价格的整体最小化。具体流程如下：

1. **枚举商店子集**：对所有商店的子集 S（$|S| \leq k$）进行枚举。每个子集 S 用**位掩码**表示，便于高效遍历和存储。例如，m 个商店可用 m 位二进制数表示，第 j 位为 1 表示选择商店 j。

2. **贪心分配物品**：对于每个枚举到的商店子集 S，判断其是否能覆盖所有物品。若可行，则对每个物品 i，在 S 中选择价格最低且有售该物品的商店 j 进行购买。

3. **总花费计算**：总花费为所选商店的配送费之和，加上每个物品在 S 中最低价格的总和。
4. **可行性判定**：若某物品在 S 中无可购买渠道，则该状态不可行，直接跳过。
5. **最优解选择**：在所有 $|S| \le k$ 的可行 S 中，取总花费最小者，并记录对应的商店组合和物品分配方案。

该方法通过枚举所有不超过 k 个商店的组合，并在每种组合下采用贪心分配，兼顾了配送费与商品价格的全局最优权衡。与暴力枚举所有分配方案相比，枚举子集 + 贪心分配大大减少了计算量，适用于 k 和 m 不大的实际场景。

2.2 代码实现

具体代码参见grredy.cpp

Algorithm 1 多平台商品采购优化——贪心算法伪代码

Input: n（物品数），m（商店数），k（最多可选商店数），delivery$[m]$（配送费），prices$[n][m]$（价格，-1 不可买）

Output: 最小总花费、商店选择、物品分配，或 -1（无解）

$best\_cost \leftarrow +\infty$
for 每个商店子集 S，$|S| \le k$ **do**
  $cost \leftarrow S$ 的配送费之和
  for 每个物品 i **do**
    在 S 中选价格最小的可买商店 j，记 min_price
    if 不存在可买商店 **then**
      跳到下一个 S
    end if
    $cost \leftarrow cost + min\_price$
    记录物品 i 的分配商店
  end for
  if $cost < best\_cost$ **then**
    更新 $best\_cost$、S 和分配方案
  end if
end for
if 未找到可行方案 **then**
  输出 -1
else
  输出 $best\_cost$、商店列表和分配方案
end if

2.3 复杂度分析

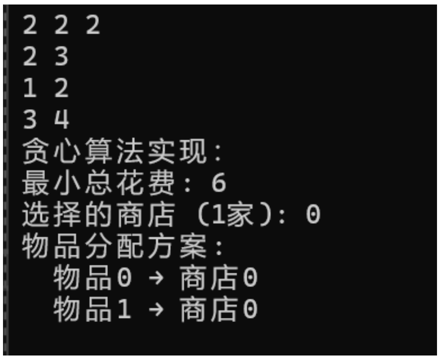
时间复杂度：

- **子集枚举：** 共有 2^m 个商店子集需要枚举。
- **每个子集处理：** 对于每个子集，需对 n 个物品分别在至多 m 个商店中查找最低价格，时间复杂度为 $O(n \times m)$。

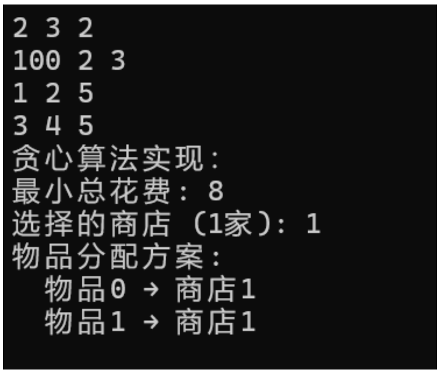
- **总时间复杂度：** $O(2^m \times n \times m)$。
- **适用范围：** 当 $m \leq 20$ 且 $n \leq 1000$ 时（常见电商场景），算法可在 1 秒内完成计算。

空间复杂度： 主要用于存储价格矩阵和分配方案，为 $O(n \times m)$。

3 实验结果与分析

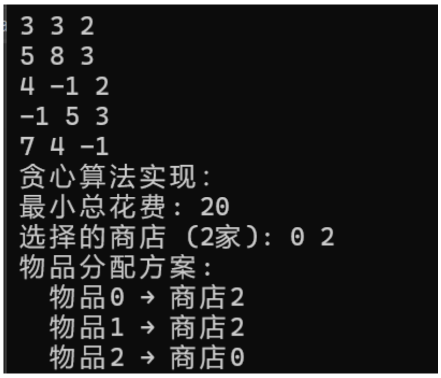


(a) 基础场景



(b) 高价配送费优化

(c) 无解场景



(d) 多商店协作

图 3.1: 不同实验场景下的结果对比

更多测试数据请参考test_data，经实验两种算法均已成功通过测试数据。

4 动态规划算法

4.1 算法思路

但实际上，第一次看到该问题，由于每个物品的决策依赖于之前已选的商店集合，且存在最优子结构和重叠子问题，我们可能会联想到动态规划算法，那么该问题能用动态规划算法实现吗？性能又如何？

答案是肯定的，动态规划算法的核心思想是：用状态压缩（即用二进制掩码表示商店集合）描述“当前已选的商店集合”，并逐步为每个物品分配购买商店，动态维护最优解。具体思路如下：

1. **状态定义**：令 $dp[i][mask][j]$ 表示前 i 个物品已分配，当前已选商店集合为 $mask$（m 位二进制，1 表示选中），共选了 j 个商店时的最小总花费。为节省空间，可将 i 维滚动优化，仅保留当前和上一物品的状态。

2. **状态转移**：

- 对于第 i 个物品，枚举其购买商店 s，若 s 已在 $mask$ 中，则直接在 $mask$ 下购买，花费为 $dp[i-1][mask][j]+prices[i][s]$。
- 若 s 不在 $mask$，且 $j<k$，则可以新增商店 s，花费为 $dp[i-1][mask'][j-1]+prices[i][s]+delivery[s]$，其中 $mask'=mask\setminus\{s\}$。

3. **初始状态**：$dp[0][0][0]=0$，其余为无穷大。

4. **目标**：枚举所有 $mask$ 和 $j\le k$，使得所有物品都被分配，取最小的 $dp[n][mask][j]$ 作为答案。

5. **可行性判定**：若所有 $dp[n][mask][j]$ 都为无穷大，则无解。

6. **方案恢复**：通过记录转移路径，可反推出最优的商店集合和每个物品的分配方案。

该方法利用状态压缩和动态规划，有效避免了对所有商店组合的暴力枚举，适合 m 和 k 不大的场景，能在保证最优性的同时提升效率。

4.2 代码实现

具体代码参见dp.cpp

Algorithm 2 多平台商品采购优化——动态规划算法伪代码

Input: n（物品数），m（商店数），k（最多可选商店数），delivery$[m]$（配送费），prices$[n][m]$（价格，-1 不可买）

Output: 最小总花费、商店选择、物品分配，或 -1（无解）

预处理：对每个物品 i 和商店组合 $mask$，计算 $minPrice[i][mask]$，即在 $mask$ 覆盖的商店中购买物品 i 的最低价格（不可买为 INF）
初始化 $dp[j][mask]$ 为 INF，表示用 j 个商店、商店集合为 $mask$ 时前 i 个物品的最小总花费
$dp[0][0] \leftarrow 0$
for 每个物品 $i=0$ 到 $n-1$ **do**
    新建 $new\_dp[j][mask] \leftarrow INF$
    for $j=0$ 到 k **do**
        for 每个商店组合 $mask$ **do**
            if $dp[j][mask]=INF$ **then continue**
            end if ▷ 情况 1：在已选商店中购买
            if $minPrice[i][mask]\neq INF$ **then**
                $new\_dp[j][mask] \leftarrow \min(new\_dp[j][mask], dp[j][mask]+minPrice[i][mask])$
            end if ▷ 情况 2：新增商店购买
            if $j<k$ **then**
                for 商店 $shop=0$ 到 $m-1$ **do**
                    if $mask$ 已包含 $shop$ 或 $prices[i][shop]=INF$ **then continue**
                    end if
                    $new\_mask \leftarrow mask|(1 \ll shop)$

```
                new_dp[j+1][new_mask] ← min(new_dp[j+1][new_mask], dp[j][mask]+delivery[shop]+prices[i][shop])
              end for
            end if
          end for
      end for
      dp ← new_dp
end for
min_cost ← INF
best_mask, best_j ← −1
for j = 0 到 k do
    for 每个 mask do
        if dp[j][mask] < min_cost then
            min_cost ← dp[j][mask]
            best_mask ← mask, best_j ← j
        end if
    end for
end for
if min_cost = INF then
    输出 −1
else
    反推 best_mask 下每个物品的分配商店（在 best_mask 覆盖的商店中选价格最小者）
    输出 min_cost、所选商店列表、物品分配方案
end if
```

4.3 复杂度分析

时间复杂度：

- **预处理**：计算每个物品在所有商店组合下的最低价格，复杂度为 $O(n \times m \times 2^m)$。
- **动态规划**：状态总数为 $O(k \times 2^m)$，每步转移最多枚举 m 个商店，整体复杂度为 $O(n \times k \times m \times 2^m)$。

空间复杂度：$O(k \times 2^m)$，可通过滚动数组进一步优化空间占用。

4.4 两个算法性能对比分析

在前边的内容中，我们针对多平台商品采购优化问题，系统分析并实现了“枚举子集 + 贪心分配”与“状态压缩动态规划”两类算法。通过对比可知，贪心算法适用于商店数较小（$m \leq 20$）的场景，具有实现简单、效率高的优点，但无法利用子问题结构；而动态规划算法通过状态转移和子问题复用，显著降低了时间和空间复杂度，适合 k 和 m 更小但问题规模更大的情形。

具体对比如下：

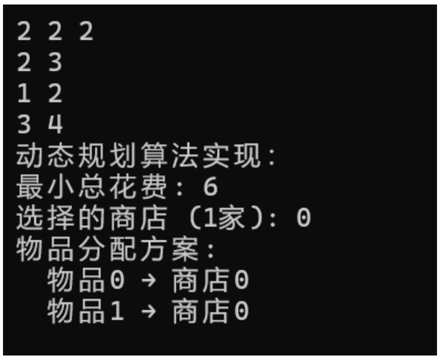
- **贪心算法**：
  - **优点**：实现简单，易于理解，代码量少，适合快速原型开发。对于商店数量较少的情况，能够在极短时间内得到最优解。

- **缺点**：需要枚举所有不超过 k 个商店的子集，子集数为 $O(2^m)$，当 m 较大时（如 $m > 20$）计算量急剧增加，难以扩展到大规模实例。
- **适用场景**：m 较小、k 较小、对运行时间要求较高但问题规模有限的实际应用。

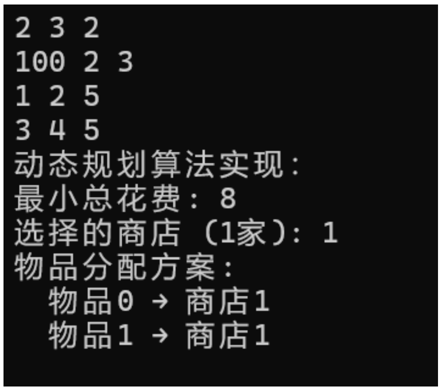
- **动态规划算法**：
  - **优点**：利用状态压缩和子问题最优性，避免了重复计算，能处理更大规模的 n 和更复杂的约束，保证全局最优解。
  - **缺点**：状态空间为 $O(k \times 2^m)$，实现复杂度较高，内存消耗较大，m 和 k 过大时依然会遇到瓶颈。
  - **适用场景**：n 较大、m 和 k 适中（如 $m \leq 18$，$k \leq 5$），需要精确解和方案恢复的场合。

总结：贪心算法适合小 m、快实现的场景，动态规划适合 n 大、需精确解的需求。实际应用中可根据问题规模和精度要求灵活选择，必要时结合剪枝、启发式等手段提升效率。

4.5 实验结果与分析

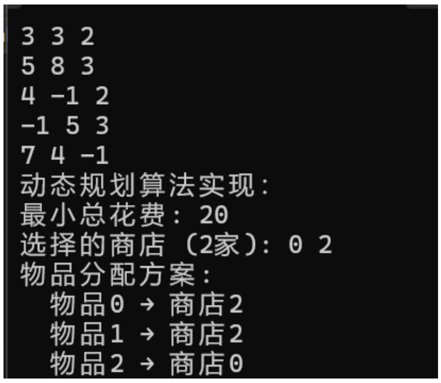


(a) 基础场景



(b) 高价配送费优化

(c) 无解场景



(d) 多商店协作

图 4.2: 不同实验场景下的结果对比

更多测试数据请参考test_data，经实验两种算法均已成功通过测试数据。

5 延申思考

5.1 NP-Hard 证明

多平台商品采购优化问题可以被证明为 NP-Hard 问题。我们可以通过归约已知的 NP-Hard 问题来证明这一点。

1. **归约自集合覆盖问题**：集合覆盖问题是 NP-Hard 的经典问题。我们可以将每个平台视为一个集合，每个商品对应于集合中的元素。目标是选择不超过 k 个集合，使得所有元素（商品）都被覆盖(购买)。

2. **归约过程**：给定一个集合覆盖实例 (U, S, k)，其中 U 是元素集合，S 是集合的集合，k 是最大选择集合数。我们可以构造一个多平台商品采购优化实例，其中：- 每个元素 $u \in U$ 对应一个商品 i。
- 每个集合 $s \in S$ 对应一个平台 j，并且如果 $u \in s$，则 $\text{prices}[i][j] = 1$（表示该平台可以购买该商品）。
- 每个平台的配送费 $\text{delivery}[j]$ 可以设置为 0，以简化问题。

3. **等价性**：在这个构造中，选择 k 个平台相当于选择 k 个集合，使得所有商品都被覆盖。因此，如果我们能解决多平台商品采购优化问题，就能解决集合覆盖问题，从而证明了前者的 NP-Hard 性。

5.2 扩展模型

5.2.1 捆绑销售

在实际电商平台中，捆绑销售是一种常见的促销策略。我们可以将捆绑销售引入到多平台商品采购优化问题中，以进一步降低总花费。

1. **捆绑定义**：假设每个平台可以提供一些捆绑商品（例如，购买 A 和 B 可享受折扣），我们可以将捆绑商品视为一个新的虚拟商品。

2. **捆绑价格矩阵**：修改价格矩阵 $\text{prices}[n][m]$，使其包含捆绑商品的价格。例如，如果平台 j 提供捆绑商品 b，则 $\text{prices}[b][j]$ 表示购买捆绑商品的价格。

3. **优化目标**：在原有的最小化总花费的基础上，增加捆绑商品的考虑。我们可以通过修改贪心算法和动态规划算法，使其能够处理捆绑商品的情况。

5.2.2 阶梯配送费

在实际电商平台中，配送费通常是阶梯式的，即购买的商品数量越多，单件商品的配送费可能会降低。我们可以将阶梯配送费引入到多平台商品采购优化问题中。

1. **阶梯配送费定义**：假设每个平台的配送费是分段的，例如购买 1 件商品时配送费为 d_1，购买 2 件商品时为 d_2，以此类推。

2. **配送费函数**：我们可以定义一个配送费函数 $\text{delivery}[j][count]$，表示在平台 j 上购买 $count$ 件商品时的配送费。

3. **优化目标**：在原有的最小化总花费的基础上，增加阶梯配送费的考虑。我们可以修改贪心算法和动态规划算法，使其能够处理阶梯配送费的情况。

5.2.3 多目标优化

在实际电商平台中，消费者的决策往往不仅仅基于价格，还可能考虑其他因素，如配送时间、商品质量等。我们可以将多目标优化引入到多平台商品采购优化问题中。

1. **多目标定义**：假设我们有多个目标函数，例如最小化总花费、最小化配送时间等。我们可以将这些目标函数组合成一个向量 $\mathbf{f} = (f_1, f_2, \ldots, f_k)$。

2. **Pareto 优化**：我们可以使用 Pareto 优化方法来寻找最优解。一个解是 Pareto 最优的，如果不存在其他解在所有目标上都优于它。

3. **优化算法**：我们可以修改贪心算法和动态规划算法，使其能够处理多目标优化问题。例如，在贪心算法中，我们可以在选择商店时同时考虑多个目标函数的权重。

6 总结

本文针对多平台商品采购优化问题，提出了两种算法：枚举子集 + 贪心分配和状态压缩动态规划。通过对比分析，我们发现：贪心算法适用于商店数较小的场景，具有实现简单、效率高的优点，但无法利用子问题结构；而动态规划算法通过状态转移和子问题复用，显著降低了时间和空间复杂度，适合更大规模的实例。

在实验中，我们验证了两种算法在不同场景下的有效性，并通过性能分析和实验结果对比，展示了两种算法的优缺点。最后，我们还探讨了该问题的 NP-Hard 性质以及可能的扩展模型，如捆绑销售、阶梯配送费和多目标优化等。

通过本文的研究，我们不仅深入理解了多平台商品采购优化问题的复杂性，还掌握了贪心和动态规划两种重要算法的应用。希望本文能在实际电商场景中的决策提供有价值的参考。