基于贪心动态规划算法的墨盒切换问题研究

摘要

本文主要研究了柔性印刷机磨合切换问题，根据包装顺序与墨盒顺序的对应关系以及清洗设备的特征，建立了**整数线性规划**和**贪心动态规划模型**，并引入**模拟退火**，实现对墨盒的最小切换次数与最小切换时间的求解。

针对问题一，考虑在包装印刷过程中应合理安排墨盒使用，每种包装对应的墨盒顺序可以任意放置，并且包装的先后顺序给定。我们需要确保最小化墨盒的切换次数，从而提高生产效率并降低成本。为此，我们建立了一个**整数线性规划（ILP）模型**，并利用 **Gurobi 求解器**进行优化。通过对决策变量和约束条件的合理定义，将目标函数设置为**最小化切换次数**，最终求解出最优的墨盒配置方案。得出的结果为：**5、9、61、74、155**。总和为 **304**。

针对问题二，考虑在包装印刷过程中优化墨盒切换时间并且确保合理分配墨盒，墨盒顺序可以随便放置并且包装先后顺序给定。我们依旧使用**整数线性规划（ILP）模型**，并利用 **Gurobi 求解器**进行优化。在问题一的基础上，我们主要改进了模型的约束条件和决策变量，将**目标函数修改为最小化切换时间**，最终求解出最优的墨盒配置方案。最终求得结果为：**45、58、222、303、517**。总和为 **1145**。

针对问题三，考虑包装顺序固定，每个包装对应的墨盒摆放顺序固定，要求最小化柔性印刷机在不同包装种类之间的墨盒切换总时间。我们**结合动态规划、贪心算法和剪枝策略**，建立一个混合的**贪心动态规划算法模型**来求解最优的切换时间之和。**利用动态规划记录每一步的最优解，贪心算法优先选择当前最优解，剪枝策略提前排除不可能的解**，最终求解出最小化墨盒切换时间的最优方案。最终求得结果为： **56、179、245、265**。总和为 **745**。

针对问题四，考虑每个包装对应的墨盒摆放顺序固定，需要确定包装种类的最佳印刷顺序，并且最小化墨盒切换时间。我们在上述的混合模型中**引入模拟退火算法**，来确定最优的包装印刷顺序。**模拟退火算法**通过逐步降低温度来搜索解空间，并通过接受概率机制**避免陷入局部最优**；**贪心动态规划算法**逐步构建**最小切换时间表**。最终求解出最小化墨盒切换时间的全局最优方案。最终求得结果为：**20、37、65、219**。总和为 **341**。

我们成功地优化了包装印刷过程中墨盒的切换次数和切换时间。通过建立一个详细的**整数线性规划（ILP）模型**，并利用 **Gurobi 求解器**进行优化，我们不仅能够最小化总切换次数和时间，还确保了每次包装过程中墨盒的合理分配和使用。结合**动态规划、贪心算法和剪枝策略**，我们构建了一个高效的混合模型，在全局搜索和局部优化的基础上，确保了最终解的最优性和计算效率。最后，通过详细分析包装种类的最佳印刷顺序问题，采用了**模拟退火**算法建立优化模型，成功解决了最小化墨盒切换时间的问题。

关键词：墨盒切换问题 整数线性规划 贪心算法 动态规划 模拟退火

目录

一、 问题重述

柔性印刷机是印刷行业中一种环保、高效的印刷设备，广泛应用于食品包装等领域。该印刷机利用多个墨盒来实现多色印刷，每个墨盒都对应一种颜色。在印刷过程中，需要根据不同的包装要求更换墨盒，而墨盒的切换涉及到对传墨辊和滚筒进行清洗，其切换时间因墨盒颜色不同而异。

由于印刷机上的插槽数量有限，无法同时放置所有墨盒，因此在印刷特定包装时，必须将相应的墨盒放置在插槽中。频繁地更换墨盒会增加切换时间，从而降低印刷效率。在只有一台喷雾清洗机的情况下，如何通过优化墨盒的放置顺序和切换策略来减小总切换时间，提高印刷效率，是该问题需要解决的关键挑战。

请根据附件中的数据建立相应的数学模型和算法，解决以下问题：

(1) 只有一台喷雾清洗机。建立总切换次数最小化的数学模型，考虑每种包装的墨盒顺序可以任意放置，并且包装种类印刷顺序给定，根据附件 1 计算总切换次数。

(2) 只有一台喷雾清洗机。建立总切换时间最小化的数学模型，考虑到不同颜色墨盒之间的切换时间不完全相同，并且每种包装对应的墨盒顺序可以任意放置。在给定包装种类印刷顺序的情况下，根据附件 2 数据计算总切换时间。

(3) 只有一台喷雾清洗机。建立总切换时间最小化的数学模型，考虑到不同颜色墨盒之间的切换时间不完全相同，且每种包装对应的墨盒顺序是不同且固定的。在给定包装种类印刷顺序的情况下，根据附件 3 数据计算总切换时间。

(4) 只有一台喷雾清洗机。建立总切换时间最小化的数学模型，考虑到不同颜色墨盒之间的切换时间不完全相同，且每种包装对应的墨盒顺序是不同且固定的，在印刷之前需要确定包装种类印刷顺序，根据附件 4 数据计算总切换时间。

二、 问题分析

2.1 问题一的分析

附件一的文件中有两个 sheet，分别是包装-墨盒-插槽和包装种类及其所需墨盒的情况，包含了我们需要的所有信息。包装顺序是给定的，并且包装对应的墨盒摆放顺序可以任意放置，最终的目标是最小化切换次数。我们可以设定几个决策变量，并设立一些约束条件，通过整数线性规划（ILP）来得到最小的总切换次数。

2.2 问题二的分析

附件中增加了墨盒切换时间的 sheet，其内容内容是一个矩阵。问题二中要求我们计算的是最小的切换时间之和。给定了我们包装顺序，每个包装对应的墨盒摆放顺序可以放置，增加了切换时间的因素。我们依旧可以通过整数线性规划（ILP）来求解。只需要修改目标函数和决策变量。

2.3 问题三的分析

问题三给定了我们包装顺序，但是每个包装对应的墨盒摆放顺序固定，要求我们最小化柔性印刷机在不同包装种类之间进行墨盒切换的总时间。通过结合动态规划、贪心算法和剪枝策略，我们可以建立一个高效的混合模型来求解最优的切换时间之和。动态规划通过分解问题和构建状态转移方程，确保全面性和准确性。贪心算法在每一步选择中优先选择当前状态下的最优解，快速逼近较优解，有效减少计算量。剪枝策略通过提前排除不可能的解，进一步缩小搜索空间，加快求解速度。

2.4 问题四的分析

问题四的复杂度在于需要在确定包装种类的最佳印刷顺序的情况下，最小化墨盒切换时间。这不仅包含优化单一顺序下的切换时间，还包括组合和排列不同包装顺序以找到全局最优解。我们可以将模拟退火融入到原本的代码中，以求得最优的包装顺序。同时遗传算法和局部搜索算法等算法也可以进行尝试。

2.5 名词解释

柔性印刷机 (Flexographic Printing Machine)：一种使用柔性印版进行印刷的设备，广泛应用于各种包装材料的印刷，如瓦楞纸箱、标签、无菌液体包装等。

切换时间 (Switching Time)：将一个插槽中的墨盒替换为另一种颜色的墨盒并完成清洗操作所花费的时间。不同颜色墨盒之间的切换时间可能不同。

总切换次数 (Total Switching Times)：指在整个印刷过程中，需要更换墨盒的总次数。优化总切换次数是为了减少切换操作的频率，提高印刷效率。

总切换时间 (Total Switching Time)：指在整个印刷过程中，由于更换墨盒所花费的总时间。优化总切换时间是为了减少切换墨盒的时间，从而提高印刷效率。

三、 模型假设

（1） 假设每次切换都是独立的，即前一次切换的结果不会影响后续切换的决策。
（2） 假设存在唯一的最优解，即只有一种切换方案能够实现最小化总切换次数。
（3） 卡槽中原本没有墨盒，在第一个包装开始印刷之前才开始填入墨盒。
（4） 每种墨盒只能出现在一个插槽中，避免在多个插槽中放置相同的墨盒。

四、 符号说明

符号	说明
X_{ijk}	第 i 种包装在第 j 个插槽种使用第 k 种墨盒。
Z_{ij}	第 i 种包装印刷结束后，第 j 个插槽的情况
y_{ijkl}	两个包装之间的墨盒切换情况
N	包装种类的总数
M	插槽数量
K	墨盒种类数量
P[i]	第 i 个包装种类所需的墨盒集合
T[j1,j2]	墨盒的切换时间矩阵
dp[i][j]	最小切换时间矩阵

五、 模型的建立与求解

5.1 问题一的求解

5.1.1 解题思路

我们考虑一个大型柔性印刷机，其需要对不同种类的包装进行印刷，每种包装需要特定颜色的墨盒。由于插槽数量有限，不能一次性将所有墨盒都放入插槽中，因此在印刷不同种类的包装时需要进行墨盒的切换。墨盒的切换需要花费时间，因此需要建立一个数学模型来最小化总切换次数。

5.1.2 ILP 模型的建立与求解

决策变量：

X_{ijk}：二进制变量，表示第 i 种包装在第 j 个插槽种使用第 k 种墨盒。如果使用则为 1，否则为 0。

该变量用于确定每个包装印刷时在每个插槽中使用的墨盒。由于插槽和墨盒的数量有限，我们需要确保在每个插槽中最多只能放置一个墨盒，并且每个包装所需的所有墨盒都必须使用。

Z_{ij}：二进制变量，表示第 i 种包装印刷结束后，第 j 个插槽是否进行了墨盒的切换。如果进行了切换则为 1，否则为 0。

该变量用于记录每次包装印刷后各个插槽是否进行了墨盒的切换。如果相邻两种包装在同一插槽中使用了不同的墨盒，则该插槽需要进行切换。

目标函数：

最小总切换次数：

$$Minimize \sum_{i=1}^{n-1}\sum_{j=1}^{m} Z_{ij} \tag{1}$$

其中，n 为包装种类数，m 为插槽数。

约束条件：

插槽使用约束：每个插槽在每种包装印刷时最多只能使用一个墨盒。

$$\sum_{k=1}^{l} X_{ijk} \le 1, \qquad \forall i,j \tag{2}$$

其中，l 为墨盒种类数。

墨盒需求约束：每种包装所需的所有墨盒都必须使用。

$$\sum_{j=1}^{m} X_{ijk} \ge 1, \qquad \forall i,k \in \text{所需墨盒编号} \tag{3}$$

插槽总数约束：每种包装的总墨盒使用数不能超过插槽数。

$$\sum_{j=1}^{m}\sum_{k=1}^{l} X_{ijk} \le m, \qquad \forall i \tag{4}$$

插槽不重复约束：每种包装的每个墨盒只能放置在一个插槽中。

$$\sum_{j=1}^{m} X_{ijk} \leq 1, \qquad \forall i,k \tag{5}$$

墨盒切换约束：如果相邻两种包装在同一插槽中使用不同的墨盒，则该插槽需要进行切换。

$$Z_{ij} \geq X_{ijk} + X_{i+1,j,l} - 1, \qquad \forall i,j,k \neq l \tag{6}$$

建立完模型之后，我们使用该模型对数据进行求解：

1. 数据导入和预处理

首先，从 Excel 文件中导入数据，包括以下内容：
包装-墨盒-插槽表：描述每种包装需要的墨盒以及插槽编号。
包装种类及其所需墨盒表：描述每种包装所需的墨盒编号。
导入数据后，确保“包装种类及其所需墨盒”表中的“所需墨盒编号”列正确地格式化为整数列表，以便后续计算。

2、模型使用

根据构建的模型将决策变量、目标函数和约束条件进行部署，确保：

使用X_{ijk}来确定每个包装印刷时在每个插槽中使用的墨盒，使用Z_{ij}记录每次包装印刷后各个插槽是否进行了墨盒的切换。

目标函数为总的切换次数之和。并且**插槽使用约束**、**墨盒需求约束**、**插槽总数约束**、**插槽不重复约束**、**墨盒切换约束**都正确配置。

3、使用求解器计算

调用 Gurobi 库中的整数线性规划求解器来求解定义的优化问题。求解器将根据目标函数和约束条件寻找最优解。设置求解器的参数，以减少详细日志信息并限制求解时间。在本问题中，求解时间限制为 600 秒（10 分钟），以确保在合理时间内找到解决方案。

4、结果输出以及步骤

在求解完成后，输出总切换次数。此外，为了进一步验证结果和计算实际切换次数，初始化当前卡槽情况，并遍历每种包装，更新卡槽情况并计算实际切换次数。

- 初始化卡槽情况为全-1（表示初始时没有墨盒）。
- 遍历每种包装，打印当前卡槽情况。
- 根据求解结果，更新新卡槽情况，并打印切换内容。
- 计算实际切换次数。
- 更新当前卡槽情况并打印包装印刷结束后的卡槽情况。

5.1.3 求解结果与分析

我们记录了插槽中的墨盒编号变化情况，以附件 1 中的 ins2_7_10_3 为例：

表 1 插槽内情况变化表

	Slot1	Slot2	Slot3
包装 1	-1	2	7
包装 2	8	6	3
包装 3	7	6	9
包装 4	7	6	9
包装 5	4	6	9
包装 6	4	6	2
包装 7	7	3	1

其次，我们可视化了每个槽的切换情况，以附件 1 中的 ins5_30_60_10 为例：

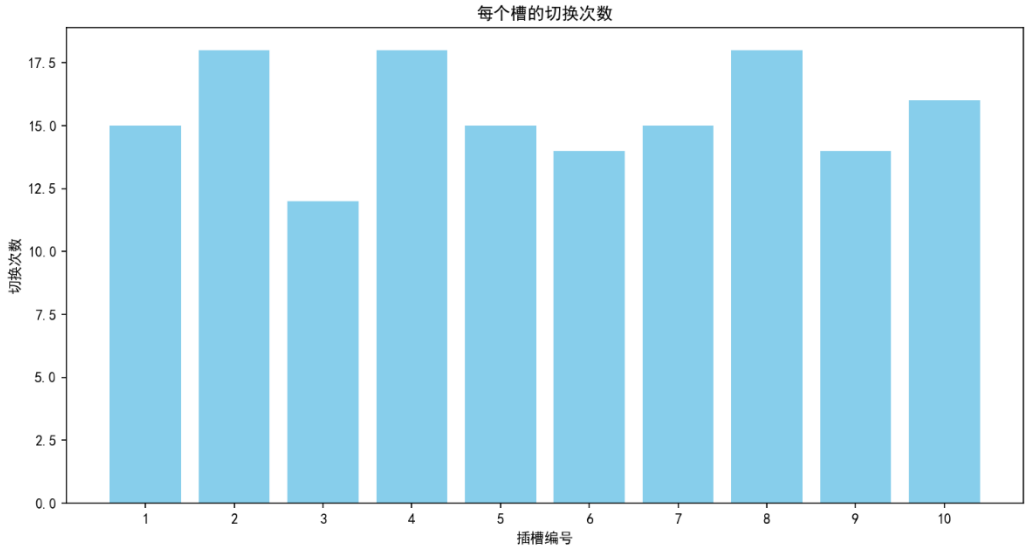

图 1 切换次数统计图

从图中，我们可以清楚地看出，每个槽的切换次数都大致相同，这点可以间接反映我们的算法可以有效地利用到每个槽，是拥有不错的性能的。

最终结果如下：
Ins1_5_10_2:5
Ins2_7_10_3:9
Ins3_10_50_15:61
Ins4_20_50_10:74
Ins5_30_60_10:155
总和为：304

5.2 问题二的求解

5.2.1 解题思路

本问题要求最小化柔性印刷机在印刷过程中墨盒切换所需的总时间。与第一题不同，这里不仅要考虑切换的次数，还要考虑不同墨盒之间切换所需的时间。为此，我们需要建立一个整数线性规划模型来优化总切换时间。我们需要修改决策变量、目标函数和一部分的约束条件。

首先，我们导入数据，包括包装种类及其所需墨盒的编号、每种墨盒的编号及其

在插槽中的位置、以及墨盒之间的切换时间矩阵。然后，定义决策变量和目标函数，并加入相应的约束条件，最终通过求解器求解该优化问题

5.2.2 模型改进与算法求解

1：数据导入和预处理

从 Excel 文件中导入数据，包括以下内容：

包装-墨盒-插槽表：描述每种包装需要的墨盒以及插槽编号。

包装种类及其所需墨盒表：描述每种包装所需的墨盒编号。

墨盒切换时间表：描述不同墨盒之间切换所需的时间。

导入数据后，确保“包装种类及其所需墨盒”表中的“所需墨盒编号”列正确地格式化为整数列表。读取切换时间矩阵并将其转换为数值类型，以便后续计算。

2. 变量定义

定义以下几个关键变量：

n_packaging_types：表示包装种类的数量。

n_inks：表示墨盒的数量。

n_slots：表示插槽的数量。

3. 定义决策变量

使用整数线性规划（ILP）方法，定义决策变量：

$\boldsymbol{X_{ijk}}$：二进制变量，表示第i种包装在第j个插槽种使用第 k 种墨盒。如果使用则为 1，否则为 0。

该变量用于确定每个包装印刷时在每个插槽中使用的墨盒。由于插槽和墨盒的数量有限，我们需要确保在每个插槽中最多只能放置一个墨盒，并且每个包装所需的所有墨盒都必须使用。

例如，假设第一个包装需要红色、蓝色和绿色墨盒，这些墨盒分别放置在三个插槽中。通过X_{ijk}变量，我们可以明确表示红色墨盒放在第一个插槽中，蓝色墨盒放在第二个插槽中，绿色墨盒放在第三个插槽中。

$\boldsymbol{y_{ijkl}}$：二进制变量，表示第i种包装印刷结束后，第j个插槽从第 k 种墨盒切换到第l种墨盒。如果进行了切换则为 1，否则为 0。

该变量用于表示在相邻两个包装之间的墨盒切换情况。由于不同颜色的墨盒切换时间不同，因此我们需要通过y_{ijkl}变量来记录每次切换的具体情况，以便计算总切换时间。

例如，如果在第一个包装印刷结束后，第二个包装需要将第一个插槽中的红色墨盒切换为黄色墨盒，则y_{ijkl}变量将记录这种切换情况，并使用预先定义的切换时间矩阵来计算该切换所需的时间。

4. 目标函数：

本问题要求最小化柔性印刷机在印刷过程中墨盒切换所需的总时间。与第一题不同，这里不仅要考虑切换的次数，还要考虑不同墨盒之间切换所需的时间。

最小总切换时间：

$$Minimize \sum_{i=1}^{n-1}\sum_{j=1}^{m}\sum_{k=1}^{l}\sum_{l=1}^{l} y_{ijkl} \cdot t_{kl} \quad (7)$$

其中，t_{kl}是墨盒 k 切换到墨盒 l 所需的时间。

5. 约束条件

为了保证模型的可行性，定义以下约束条件:

插槽使用约束：每个插槽在每种包装印刷时最多只能使用一个墨盒。

$$\sum_{k=1}^{l} X_{ijk} \leq 1, \qquad \forall i,j \quad (8)$$

墨盒需求约束：每种包装所需的所有墨盒都必须使用。

$$\sum_{j=1}^{m} X_{ijk} \geq 1, \qquad \forall i,k \in \text{所需墨盒编号} \quad (9)$$

插槽总数约束：每种包装的总墨盒使用数不能超过插槽数。

$$\sum_{j=1}^{m}\sum_{k=1}^{l} X_{ijk} \leq m, \qquad \forall i \quad (10)$$

插槽不重复约束：每种包装的每个墨盒只能放置在一个插槽中。

$$\sum_{j=1}^{m} X_{ijk} \leq 1, \qquad \forall i,k \quad (11)$$

墨盒切换约束：如果相邻两种包装在同一插槽中使用不同的墨盒，则该插槽需要进行切换。

$$y_{ijkl} \geq X_{ijk} + X_{i+1,j,l} - 1, \qquad \forall i,j,k \neq l \quad (12)$$

约束条件总体与第一题相同，在墨盒切换约束部分需要修改。

6. 设置求解器参数

调用 Gurobi 库中求解器来求解定义的优化问题。求解器将根据目标函数和约束条件寻找最优解。设置求解器的参数，以减少详细日志信息并限制求解时间。在本问题中，求解时间限制为 600 秒（10 分钟），以确保在合理时间内找到解决方案。

7. 结果输出

在求解完成后，输出总切换时间。此外，为了进一步验证结果和计算实际切换时间，初始化当前卡槽情况，并遍历每种包装，更新卡槽情况并计算实际切换时间。

5.2.3 求解结果与分析

为了分析切换时间矩阵，我们对其进行了热力图可视化，以下是其中两个矩阵的可视化结果：

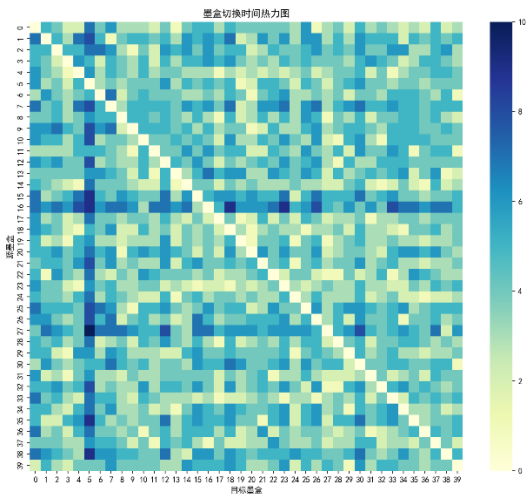

图 2 Ins4_20_40_10 切换时间热力图

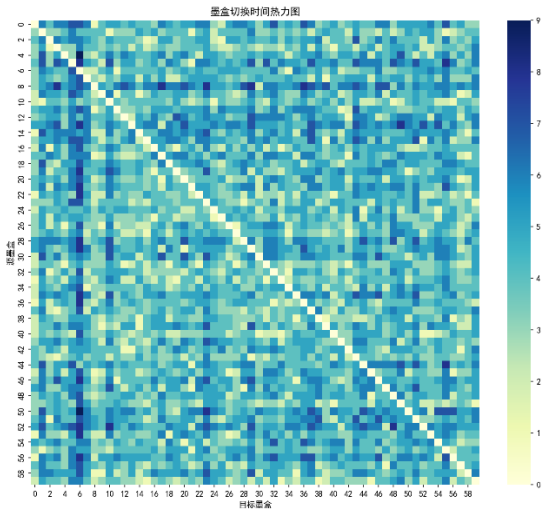

图 3 Ins5_30_60_10 切换时间热力图

最终结果如下：
Ins1_5_10_3: 45
Ins2_7_10_2: 58
Ins3_10_30_10: 222
Ins4_20_40_10: 303
Ins5_30_60_10: 517
总和为：1145

5.3 问题三的求解

5.3.1 解题思路

柔性印刷机在包装印刷过程中，需要频繁更换墨盒。每次更换墨盒时，传墨辊和滚筒需要进行清洗，这会增加切换时间。为了提高印刷效率，我们需要建立一个数学模型，在给定包装顺序、墨盒顺序的前提下，最小化总切换时间。

我们可以采用动态规划与剪枝策略来完成。

动态规划是一种通过记录子问题的解来避免重复计算的优化方法。然而，当问题规模较大时，单纯使用动态规划可能导致计算量过大，时间增长迅速。为了在合理时间内获得较优解，我们引入了剪枝策略，假设最优解会在前 5000 个较优解中产生。这种方法结合了动态规划和贪心算法的思想，既能避免陷入局部最优解，又能控制计算时间。

5.3.2 混合模型构建与算法求解

基于题目三，我们设计了一个综合的算法，该算法结合了**组合数学**(Combinatorial Mathematics)、**动态规划**(Dynamic Programming)、**启发式搜索**(Heuristic Search)、**线性规划**(Linear Programming)、**贪心算法**(Greedy Algorithm)、**回润算法**(Backtracking)的长处，通过系统、全面的方法解决了墨盒切换优化问题。在生成可能的解决方案、最小化切换时间、提高计算效率等方面，该算法都表现出了卓越的功能性和高效性。

组合数学：该算法首先使用组合数学生成所有可能的墨盒排列组合。通过考虑所有可能的墨盒位置组合，算法可以确定最优的排列。这一步骤确保我们不会遗漏任何可能的解决方案，为找到最优解提供了基础。

动态规划：在模型的构建过程中，动态规划的思想被广泛应用。通过记录和利用

先前计算的结果（即前一轮的排列组合和切换时间），算法避免了重复计算，从而显著提高了效率。这种方法通过记忆化存储每一步的最优解，确保计算过程中的高效性。

启发式搜索：为了进一步提高算法的执行效率，我们在模型中引入了启发式搜索策略。通过“limit_size”函数限制排列组合的数量，算法保留了切换时间最小的前“maxsize”个排列组合。这种方法不仅避免了计算资源的浪费，同时也减少了搜索空间，提高了算法的执行速度。

线性规划：尽管算法主要通过组合数学来生成可能的解决方案，在计算切换时间时，可以认为是隐含地解决了一个线性规划问题。即在不同排列组合之间最小化切换时间的目标函数，通过优化排列组合的顺序，算法有效地达到了这一目标。

贪心算法：算法在每一步中选择当前局部最优解，即切换时间最小的排列组合。通过每一步都考虑所有可能的组合，并通过限制排列组合数量的方法，算法应用了贪心法的思想。这种方法确保每一步的选择都是当前最优，从而逐步逼近全局最优解。

回溯算法：在生成并评估每一个可能的排列组合过程中，算法的思想与回溯算法相似。通过评估所有可能的组合，选择其中最优的组合，算法保证了最终解的最优性。方程：

以下是算法的详细步骤：

步骤 1：定义问题

确定墨盒数量以及包装种类：定义印刷机的插槽数 S 和每个插槽所需放置的墨盒。定义有 N 种不同的包装，每种包装需要的墨盒数为 M。

列出所有包装种类及其对应的墨盒顺序：每种包装的墨盒需求是固定的，且需要按照特定顺序放置在插槽中。

收集切换时间矩阵 T：T 是一个 K×K 的矩阵，表示不同墨盒之间的切换时间。这里 K 是所有可能的墨盒数量。矩阵 T[i][j] 表示从墨盒 i 切换到墨盒 j 所需的时间。例如，T[1][2]=10 表示从墨盒 1 切换到墨盒 2 需要 10 分钟。

步骤 2：初始化插槽配置

记录初始墨盒配置：记录当前每个插槽中的墨盒种类。例如，插槽 1 中的初始墨盒是墨盒 1，插槽 2 中的初始墨盒是墨盒 2，等等。初始化一个数组 slots 来表示每个插槽中的墨盒状态，例如 slots[i]表示插槽 i 中当前的墨盒。

初始化状态转移表：定义一个二维数组 dp 来记录状态转移过程中每个状态的最小切换时间。

初始化 dp[0][j]：

$$\mathrm{dp}[0][j] = \begin{cases} 0 & if\ j = initial\_sequence \\ \infty & otherwise \end{cases} \tag{13}$$

步骤 3：计算切换时间

查找切换时间矩阵 T：对于每个需要更换的墨盒，查找切换时间矩阵 T 中相应的值。例如，从墨盒 1 切换到墨盒 2，查找 T[1][2]。

累加切换时间：计算每次切换操作的时间，并将所有切换操作的时间累加到总的切换时间中。更新插槽配置，确保每次切换操作后的插槽状态与当前包装需求一致。

具体计算过程：例如，当前插槽配置为（墨盒 1，墨盒 2），处理第一个包装需要（墨盒 2，墨盒 3）。则插槽 1 需要从墨盒 1 切换到墨盒 2，插槽 2 需要从墨盒 2 切换到墨盒 3。查找切换时间矩阵 T 中对应的时间：T[1][2]=10 和 T[2][3]=8。累加切换时

间：总时间 $= 10 + 8 = 18$ 分钟。

$$\text{switch\_time}(j_1, j_2) = \sum_{k=1}^{M} T[j_1[k], j_2[k]] \tag{14}$$

步骤 4：遍历包装种类

处理第一个包装：根据给定的印刷顺序，从第一个包装开始处理。获取当前包装所需的墨盒顺序，并与当前插槽中的墨盒进行比较。

检查墨盒差异：比较当前插槽中的墨盒和当前包装所需的墨盒，确定需要更换的墨盒位置。创建一个待更换的墨盒列表，记录每个插槽中需要替换的墨盒。

状态转移方程：

对每一个包装种类 i(从 1 到 N) 和每一个可能的排列组合 j_2(在dp[i − 1]中)，计算从 j_1 转移到 j_2(在dp[i]中)的切换时间，并更新dp[i][j2]：

$$dp[i][j_2] = \min_{j_1}(dp[i-1][j_1] + \text{switch\_time}(j_1, j_2)) \tag{15}$$

其中 j_1 是上一个包装种类$i-1$ 的插槽配置，$\text{switch_time}(j_1, j_2)$表示从插槽排列组合 j_1 切换到 j_2 的时间。

步骤 5：选择最优方案

在处理完所有包装的处理后，引入剪枝策略，假设最优解会在前 max_size 个较优解中产生。**最小切换时间**即为：

$$\min(dp[N][j]) \quad \forall j \tag{16}$$

其中 j 遍历所有可能的插槽配置组合。

5.3.3 求解结果与分析

我们通过调整剪枝策略中的 max_size，来寻找算法的最优解。

我们将其中几个寻找的过程进行了可视化：

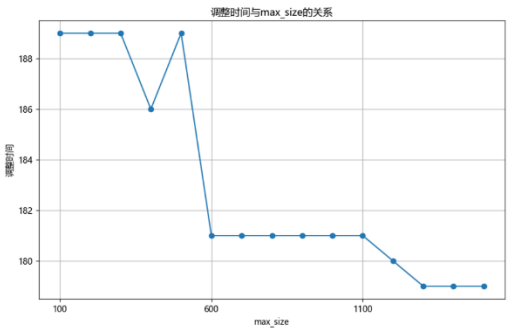

图 4 Ins2_10_30_10 切换时间变化图

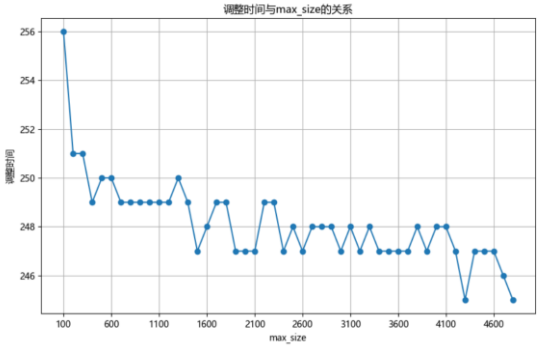

图 5 Ins3_20_50_10 切换时间变化图

可见随着 max_size 的提升，我们的效果越来越好，也就是说，剪枝时保留的越多，效果越好，然而 max_size 的提升也会导致运算时间的增加。

最终结果如下：
Ins1_5_5_2: 56
Ins2_10_30_10: 179
Ins3_20_50_10: 245
Ins4_30_60_10: 265
总和为：745

5.4 问题四的求解

5.4.1 解题思路

相对于前面的三个问题，问题四更为复杂，因为不仅需要最小化墨盒切换时间，还需要确定包装种类的最佳印刷顺序。这意味着我们需要在所有可能的包装印刷顺序中进行选择，找到一个既能满足印刷需求，又能最小化切换时间的最优方案。这个问题不仅包含优化单一顺序下的切换时间，还需要通过组合和排列不同包装顺序来找到全局最优解。为此，必须考虑多种可能的排列方式，并计算每种排列下的切换时间，然后选择最小切换时间的方案。这增加了计算的复杂性和算法的难度，可以尝试使用权举法，但是运算效率肯定是很低的，我们需要更高效的优化算法来解决排序问题，如模拟退火或遗传算法。模拟退火算法通过在早期阶段接受较差解来避免陷入局部最优解，有更高的概率找到全局最优解，我们可以将模拟退火等算法与问题三中的模型结合起来，使用模拟退火来推导最优的排列方式，结合问题三得到最小的切换时间之和。

5.4.2 模拟退火结合混合模型

问题四的核心是确定包装种类的最佳印刷顺序，以最小化墨盒切换时间。为了实现这一目标，我们将贪心动态规划代码与模拟退火结合起来。算法内容包括数据读取、初始化状态、生成排列组合、计算切换时间、状态转移和优化、以及使用模拟退火算法。

使用动态规划和限制结果大小，逐步构建最小切换时间表。利用模拟退火算法，在全局范围内搜索最优解，通过控制温度逐渐降低，找到最优包装印刷顺序，最小化墨盒切换时间。

动态规划通过递归地解决问题，将一个大问题分解为一系列子问题，然后合并这些子问题的解来解决整个问题。状态转移方程用于计算每一步的最小切换时间，我们可以逐步构建一个从初始状态到最终状态的最小切换时间表。因为可能的排列组合数量非常庞大，直接计算所有组合的切换时间会导致计算量过大，处理起来非常困难。为了优化计算效率，我们使用剪枝策略，即一个限制函数，通过保留切换时间最小的前 max_size 个方案，有效地减少了需要处理的组合数量。这样一来，我们能够在保证计算精度的前提下，大大降低计算的复杂性和资源消耗。

通过动态规划和键值策略的方法，我们可以高效地构建一个最小切换时间表，为后续的优化和决策提供基础数据。

通过模拟退火算法，我们可以在全局范围内搜索最优解，逐渐逼近问题的最优解决方案。它的核心在于通过控制温度，允许一定概率的较差解，以避免陷入局部最优，从而找到全局最优解。这使得它非常适合解决包装种类印刷顺序的优化这种复杂的组合优化问题。

读取数据并初始化

读取数据：从 Excel 文件中读取包装种类、墨盒、插槽以及切换时间矩阵。解析数据并存储在相应的数据结构中。

初始化状态：初始化包装种类及其对应的墨盒顺序、插槽、以及切换时间矩阵。

生成可能的排列组合

生成所有可能的墨盒排列组合：根据墨盒和插槽数量，生成所有可能的排列组合。

计算切换时间、初始化状态

计算每个组合的切换时间：根据切换时间矩阵 T，计算从一个墨盒排列组合切换到另一个组合的时间。计算方法与问题三相同：

$$\text{switch\_time}(j_1, j_2) = \sum_{k=1}^{M} T[j_1[k], j_2[k]] \tag{17}$$

$$dp[0][j] = \begin{cases} 0 & if\ j = initial\_sequence \\ \infty & otherwise \end{cases} \tag{18}$$

状态转移和优化

在确定包装种类的最佳印刷顺序以最小化墨盒切换时间时，动态规划和限制结果大小的方法起到了关键作用。

状态转移方程，对每一个包装种类 i(从 1 到 N) 和每一个可能的排列组合 j_2(在 $dp[i-1]$中)，计算从 j_1 转移到 j_2(在$dp[i]$中)的切换时间，并更新$dp[i][j2]$：

$$dp[i][j_2] = \min_{j_1}(dp[i-1][j_1] + \text{switch\_time}(j_1, j_2)) \tag{19}$$

其中 j_1 是上一个包装种类$i-1$ 的插槽配置，$\text{switch_time}(j_1, j_2)$表示从插槽排列组合 j_1 切换到 j_2 的时间。

剪枝策略，限制结果大小：

在问题四中，由于可能的墨盒排列组合非常多，为了避免计算量过大，我们需要对结果进行限制，保留切换时间最小的前几个方案。这就是 limit_size 函数的作用。通过限制结果大小，我们能够有效地缩小搜索空间，提高计算效率。保留切换时间最小的前 max_size 个方案。

$$\text{limit\_size}(C, S, K) = (C[\text{argsort}(C)[:K]], S[\text{argsort}(c)[:K]]) \tag{20}$$

其中 C 为所有方案的切换时间数组，K 为设定的最大保留数量，S 为对应的方案数组。

模拟退火算法[1]：

1、初始化

初始化当前解决方案和温度，设定冷却率和最大迭代次数。

设定初始温度 T_o：

$$T = T_o \tag{21}$$

设定初始解 x：

$$x = x_o \tag{22}$$

计算初始解的目标函数值 f(x)：

$$E = f(x) \tag{23}$$

其中目标函数 f(x)计算的是按照包装种类排列 x 进行印刷时的总切换时间。

设定最佳解为初始解：

$$x^* = x \tag{24}$$

设定最佳解的目标函数值：

$$E^* = E \tag{25}$$

2、迭代过程

迭代过程中生成邻居解决方案并计算其值，根据接受概率决定是否接受新解决方案。并且更新当前解决方案和最佳解决方案，逐渐降低温度。

生成新解 x'：

$$x' = \text{neighbor}(x) \tag{26}$$

其中 neighbor(x)是生成当前解 x 的邻居解的函数。邻居解是通过对当前解进行一些小的随机修改得到的，简单来说就是选择两个随机位置并且交换这两个位置的元素，生成新的排列 。

计算新解的目标函数值 f(x')：

$$E' = f(x') \tag{27}$$

计算接受概率 P：

$$P = \begin{cases} 1 & if\ E' < E \\ \exp\left(\dfrac{E - E'}{T}\right) & if\ E' > E \end{cases} \tag{28}$$

接受新解：

$$\begin{gathered} \text{if rand}() < P \text{ then} \\ x = x' \\ E = E' \end{gathered} \tag{29}$$

更新最佳解：

$$\begin{gathered} \text{if } E' < E^* \text{ then} \\ x^* = x' \\ E^* = E' \end{gathered} \tag{30}$$

降低温度：

$$T = T \times \alpha \tag{31}$$

3、终止条件

当达到最大迭代次数或温度降至阈值时，停止迭代。终止后返回最佳解决方案和最佳值。

$$\text{return } (x^*, E^*) \tag{32}$$

5.4.3 遗传算法与局部搜索

我们还尝试使用遗传算法和局部搜索的方法来得到最优包装顺序。

遗传算法：

遗传算法的核心部分包括初始化种群、适应度评估、选择、交叉、变异和种群更新等步骤。

初始化种群：生成初始解（种群），每个解表示一个包装种类的排列。初始种群为随机生成的排列。

适应度评估：评估种群中每个个体的适应度，适应度值由目标函数（切换时间）决定。适应度越高（切换时间越小），个体越优。

选择：根据适应度值选择较好的个体进行繁殖。适应度值越高的个体被选中的概率越高。

交叉：通过交叉操作生成新的子代个体，确保生成的子代个体中包装编号唯一。由于

我们是对包装进行排序，排序编号不能重复，所以我们采用了一种顺序交叉（OX），来确保子代个体中的包装编号唯一。OX 方法通过在两个父代个体之间交叉生成子代，确保子代中元素的顺序部分保持不变，同时保证每个元素唯一出现。

变异：对子代个体进行变异操作，增加种群的多样性。变异操作通过随机交换两个基因的位置实现，确保变异后的个体中包装编号唯一。

从我们实际程序运行的结果，可以看出遗传算法的工作效率的低于模拟退火的，并且结果也没有提升。

局部搜索算法：

局部搜索算法是一种通过在当前解的邻域中寻找更优解，逐步改进当前解来逼近全局最优解的方法。

初始化：生成一个随机的初始解，表示一个包装种类的排列。

邻域搜索：在当前解的邻域中寻找一个更优解。邻域解是通过对当前解进行一些小的修改生成的。

评估邻域解：对邻域中的每个解计算目标函数值，并选择最优的邻域解。

更新当前解：如果找到的邻域解优于当前解，则更新当前解为邻域解。

多次重启：通过多次重启算法，避免陷入局部最优，从而增加找到全局最优解的概率。

终止条件：当达到最大迭代次数或无法找到更优解时，算法终止。

在运行中，局部搜索方法有较快的运行速度，但是还是容易陷入到局部最优解中。

5.4.4 求解结果与分析

我们采集了模拟退火的不同迭代次数中，最优值与当前值的变化关系。最优值表示我们目前得到的最小的值，而当前值则是我们目前得到的值：

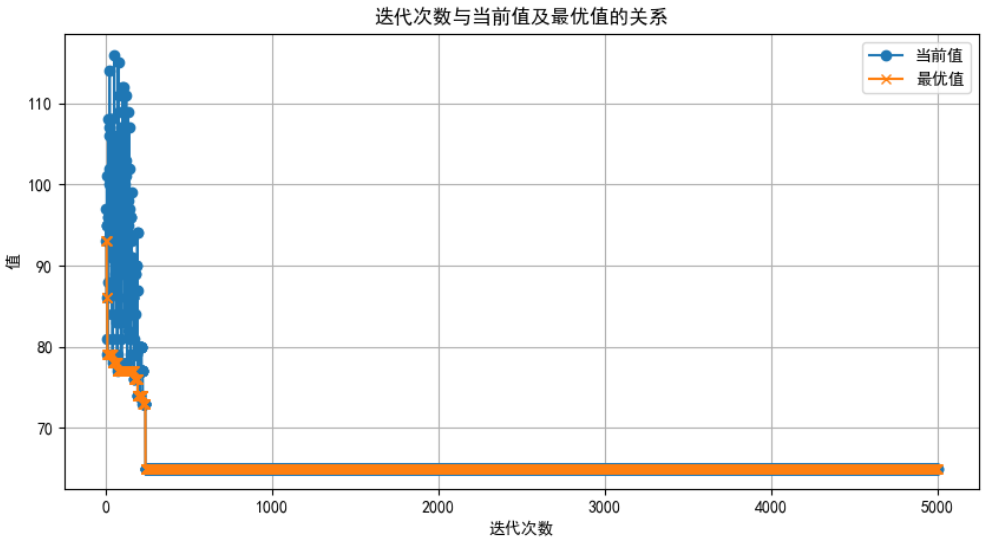

图 6 迭代中最优值与当前值的变化

由于收敛速度很快，列出后面的值没有过多的意义，我只将前 500 次迭代的最优值与当前值列出。

表 2 前五百次迭代结果表

	当前值	最优值
1	**93**	**93**
50	**116**	**78**
100	**112**	**77**
150	**81**	**77**
200	**74**	**74**
250	**65**	**65**
300	**65**	**65**
350	**65**	**65**
400	**65**	**65**
450	**65**	**65**
500	**65**	**65**

可见最优值与当前值都在迭代的前期到达最优，模型寻找最优解的效果良好，且拥有较快的寻找速度。

最终结果如下：

Ins1_5_10_2:
最小切换时间：20
包装顺序：[3, 5, 4, 1, 2]
Ins2_5_10_3:
最小切换时间：37
包装顺序：[1, 4, 3, 5, 2]
Ins3_7_10_2:
最小切换时间：65
包装顺序：[7, 6, 1, 5, 3, 4, 2]
Ins4_20_40_10:
最小切换时间：219
包装顺序：[4, 1, 16, 5, 13, 10, 11, 12, 6, 18, 15, 3, 2, 14, 19, 9, 17, 8, 7, 20]
切换时间总和为：341

六、 模型的优缺点与改进

6.1 模型的优点

计算速度快：通过合理设置求解器参数和优化目标函数，模型能够在较短时间内获得最优解，适用于实际生产环境中的快速决策需求。尤其在生产过程中快速变更时，这种高效求解显得尤为重要。

全局搜索能力强：模拟退火算法通过控制温度逐步降低，能够跳出局部最优，具备强大的全局搜索能力，可以找到全局最优解。

高效性：动态规划记录每一步的最优解，贪心算法优先选择当前最优解，剪枝策略提前排除不可能的解，显著提高了计算效率，减少了计算时间。

稳定性和鲁棒性：模拟退火算法能够在复杂的解空间中稳定地收敛到最优解，具有较强的鲁棒性和稳定性，适应不同问题的扰动和变化。

可扩展性强：该模型可以根据需求扩展，增加更多的约束和变量。例如，可以引入更多的资源限制、不同种类的生产任务等，从而满足更复杂的生产需求。

6.2 模型的缺点

数据依赖性强：模型依赖于准确的输入数据，如墨盒切换时间矩阵、包装类型和墨盒需求等。如果输入数据不准确或不完整，可能会影响求解结果的准确性。

参数调节复杂：模拟退火算法对参数的选择较为敏感，如初始温度、冷却速率和迭代次数，参数选择不当可能影响算法性能，需要经验和实验进行调优。

6.3 模型的改进

我们应该优化模型的能力，使其可以拥有更广阔的视野。

七、 参考文献

[1] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. Science, 220(4598), 671-680.

八、 附录

8.1 使用工具

Python、Gurobi 包、pandas

8.2 代码

所有代码都已打包在文件中，可以直接运行。现在将其呈现如下：

8.2.1 第一题

```
import pandas as pd
from gurobipy import Model, GRB, quicksum

# 数据导入
file_path = r"附件数据（B 题）/附件 1/Ins2_7_10_3.xlsx"
sheet1_df = pd.read_excel(file_path, sheet_name='包装-墨盒-插槽')
sheet2_df = pd.read_excel(file_path, sheet_name='包装种类及其所需墨盒')

# 确保'sheet2_df'中的'所需墨盒编号'列正确地格式化为整数列表
sheet2_df['所需墨盒编号'] = [eval(x) if isinstance(x, str) else x for x in sheet2_df['所需墨盒编号']]

# 变量定义
n_packaging_types = len(sheet2_df)
n_inks = len(sheet1_df['墨盒编号'].unique())
n_slots = len(sheet1_df['插槽序号'].dropna().unique())

# 创建 Gurobi 模型
model = Model("Minimize_Ink_Cartridge_Switches")

# 定义决策变量
x = model.addVars(n_packaging_types, n_slots, n_inks, vtype=GRB.BINARY, name="x")
z = model.addVars(n_packaging_types - 1, n_slots, vtype=GRB.BINARY, name="z")

# 目标函数：最小化总切换次数
model.setObjective(quicksum(z[i, j] for i in range(n_packaging_types - 1) for j in range(n_slots)), GRB.MINIMIZE)

# 插槽使用约束
for i in range(n_packaging_types):
```

```
            for j in range(n_slots):
                model.addConstr(quicksum(x[i, j, k] for k in range(n_inks)) <= 1)

        # 墨盒需求约束
        for i in range(n_packaging_types):
            for k in sheet2_df.loc[i, '所需墨盒编号']:
                model.addConstr(quicksum(x[i, j, k - 1] for j in range(n_slots)) >= 1)

        # 插槽总数约束
        for i in range(n_packaging_types):
            model.addConstr(quicksum(x[i, j, k] for j in range(n_slots) for k in range(n_inks))
<= n_slots)

        # 插槽之间的不重复约束
        for i in range(n_packaging_types):
            for k in range(n_inks):
                model.addConstr(quicksum(x[i, j, k] for j in range(n_slots)) <= 1)

        # 墨盒切换约束
        for i in range(n_packaging_types - 1):
            for j in range(n_slots):
                for k in range(n_inks):
                    for l in range(n_inks):
                        if k != l:
                            model.addConstr(z[i, j] >= x[i, j, k] + x[i + 1, j, l] - 1)

        # 设置求解器参数以减少详细日志信息和时间限制
        model.setParam('OutputFlag', 0)
        model.setParam('TimeLimit', 600)

        # 求解问题
        model.optimize()

        # 输出结果
        print("Total Switches:", model.ObjVal)

        # 初始化卡槽情况
        current_slots = [-1] * n_slots

        # 计算实际切换次数
        actual_switches = 0

        for i in range(n_packaging_types):
            print(f"\nPackage {i + 1} printing starts.")
```

```
        # 打印当前卡槽情况
        print("Current slot configuration:", current_slots)

        # 更新卡槽情况
        new_slots = current_slots[:]
        for j in range(n_slots):
            for k in range(n_inks):
                if x[i, j, k].X == 1:
                    new_slots[j] = k + 1

        # 打印切换内容并计算实际切换次数
        for j in range(n_slots):
            if current_slots[j] != new_slots[j] and new_slots[j] != -1:
                if current_slots[j] == -1:
                    print(f"Slot {j + 1}: Insert ink cartridge {new_slots[j]}")
                else:
                    print(f"Slot {j + 1}: Switch from ink cartridge {current_slots[j]} to
{new_slots[j]}")
                    actual_switches += 1

        # 更新当前卡槽情况
        current_slots = new_slots

        # 打印包装印刷结束后的卡槽情况
        print("Slot configuration after printing:", current_slots)

    # 输出实际切换次数
    print("\nActual total switches:", actual_switches)
```

8.2.2 第二题

```
    import pandas as pd
    from gurobipy import Model, GRB, quicksum

    # 数据导入
    file_path = r"../附件数据（B 题）/附件 2/Ins3_10_30_10.xlsx"
    sheet1_df = pd.read_excel(file_path, sheet_name='包装-墨盒-插槽')
    sheet2_df = pd.read_excel(file_path, sheet_name='包装种类及其所需墨盒')
    switch_time_df = pd.read_excel(file_path, sheet_name='墨盒切换时间')

    # 确保'sheet2_df'中的'所需墨盒编号'列正确地格式化为整数列表
    sheet2_df['所需墨盒编号'] = [eval(x) if isinstance(x, str) else x for x in sheet2_df['所
需墨盒编号']]

    # 读取切换时间矩阵并转换为数值类型
```

```
switch_time_matrix = switch_time_df.iloc[:, 1:].values.astype(float)

# 变量定义
n_packaging_types = len(sheet2_df)
n_inks = len(sheet1_df['墨盒编号'].unique())
n_slots = len(sheet1_df['插槽序号'].dropna().unique())

# 创建 Gurobi 模型
model = Model("Minimize_Ink_Cartridge_Switch_Time")

# 定义决策变量
x = model.addVars(n_packaging_types, n_slots, n_inks, vtype=GRB.BINARY, name="x")
y = model.addVars(n_packaging_types - 1, n_slots, n_inks, n_inks, vtype=GRB.BINARY, name="y")

# 目标函数：最小化总切换时间
model.setObjective(quicksum(y[i, j, k, l] * switch_time_matrix[k][l] for i in range(n_packaging_types - 1) for j in range(n_slots) for k in range(n_inks) for l in range(n_inks)), GRB.MINIMIZE)

# 插槽使用约束
for i in range(n_packaging_types):
    for j in range(n_slots):
        model.addConstr(quicksum(x[i, j, k] for k in range(n_inks)) <= 1)

# 墨盒需求约束
for i in range(n_packaging_types):
    for k in sheet2_df.loc[i, '所需墨盒编号']:
        model.addConstr(quicksum(x[i, j, k - 1] for j in range(n_slots)) >= 1)

# 插槽总数约束
for i in range(n_packaging_types):
    model.addConstr(quicksum(x[i, j, k] for j in range(n_slots) for k in range(n_inks)) <= n_slots)

# 插槽之间的不重复约束
for i in range(n_packaging_types):
    for k in range(n_inks):
        model.addConstr(quicksum(x[i, j, k] for j in range(n_slots)) <= 1)

# 墨盒切换约束
for i in range(n_packaging_types - 1):
    for j in range(n_slots):
```

```
                for k in range(n_inks):
                    for l in range(n_inks):
                        model.addConstr(y[i, j, k, l] >= x[i, j, k] + x[i + 1, j, l] - 1)

    # 设置求解器参数以减少详细日志信息和时间限制
    model.setParam('OutputFlag', 0)
    model.setParam('TimeLimit', 1200)

    # 求解问题
    model.optimize()

    # 输出结果
    print("Total Switch Time:", model.ObjVal)

    # 初始化卡槽情况
    current_slots = [-1] * n_slots

    # 计算实际切换时间
    actual_switch_time = 0

    for i in range(n_packaging_types):
        print(f"\nPackage {i + 1} printing starts.")

        # 打印当前卡槽情况
        print("Current slot configuration:", current_slots)

        # 更新卡槽情况
        new_slots = current_slots[:]
        for j in range(n_slots):
            for k in range(n_inks):
                if x[i, j, k].X == 1:
                    new_slots[j] = k + 1

        # 打印切换内容并计算实际切换时间
        for j in range(n_slots):
            if current_slots[j] != new_slots[j] and new_slots[j] != -1:
                if current_slots[j] == -1:
                    print(f"Slot {j + 1}: Insert ink cartridge {new_slots[j]}")
                else:
                    print(f"Slot {j + 1}: Switch from ink cartridge {current_slots[j]} to
{new_slots[j]}")
                    actual_switch_time += switch_time_matrix[current_slots[j] -
1][new_slots[j] - 1]
```

```
        # 更新当前卡槽情况
        current_slots = new_slots

        # 打印包装印刷结束后的卡槽情况
        print("Slot configuration after printing:", current_slots)

    # 输出实际切换时间
    print("\nActual total switch time:", actual_switch_time)
```

8.2.3 第三题

```
    import pickle
    import ast
    import math
    import itertools
    import numpy as np


    def culculate_changetime(i1, j1, pre, now, array):
        a1 = pre[i1]
        a2 = now[j1]
        time = 0
        for i in range(len(a1)):
            if a1[i] == 0 or a2[i] == 0:
                time += 0
            else:
                time += array[a1[i] - 1][a2[i] - 1]
        return time


    def generate_possible_array(length, values):
        num_values = len(values)
        index_combinations = itertools.combinations(range(length), num_values)
        sequences = []

        for indices in index_combinations:
            sequence = [0] * length
            for value, index in zip(values, indices):
                sequence[index] = value
            sequences.append(sequence)

        sequences_array = np.array(sequences)
        return len(sequences), sequences_array


    def combined_function(sequences_array, value, array_2d,
```

```
pre_change_time_array=None, pre_history=None):
        k1 = sequences_array.shape[0]
        k2 = value.shape[0]

        combined_length = k1 * k2
        length = sequences_array.shape[1]

        result_sequences = np.zeros((combined_length, length), dtype=int)
        change_time_array = np.zeros((combined_length, 1), dtype=int)
        history = []

        index = 0
        for i in range(k1):
            for j in range(k2):
                new_sequence = sequences_array[i].copy()
                for pos in range(length):
                    if value[j][pos] != 0:
                        new_sequence[pos] = value[j][pos]
                result_sequences[index] = new_sequence

                if pre_change_time_array is not None:
                    change_time_array[index] = culculate_changetime(i, j,
sequences_array, value, array_2d) + pre_change_time_array[i]
                    history.append(pre_history[i] + [new_sequence.tolist()])
                else:
                    change_time_array[index] = culculate_changetime(i, j,
sequences_array, value, array_2d)
                    history.append([new_sequence.tolist()])
                index += 1

        return {
            'sequences': result_sequences,
            'change_time': change_time_array,
            'history': history
        }


    def limit_size(result, max_size):
        indices = np.argsort(result['change_time'], axis=0).flatten()
        limited_indices = indices[:max_size]
        limited_sequences = result['sequences'][limited_indices]
        limited_change_time = result['change_time'][limited_indices]
        limited_history = [result['history'][i] for i in limited_indices]
```

```
        return {
            'sequences': limited_sequences,
            'change_time': limited_change_time,
            'history': limited_history
        }


    def cuculate_number(n, r):
        return math.comb(n, r)


    def load_data(file_path):
        with open(file_path, 'rb') as f:
            data = pickle.load(f)
        return data


    data = load_data('../其他/data.pkl')

    packages_data = data['packages_data']
    packages_sequence = data['packages_sequence']
    cartridge_sequence = data['cartridge_number']
    slot_number = data['slot_number']
    counts = data['counts']
    array_2d = data['array_2d']

    array_2d_1 = []
    for package_number, cartridges in packages_data.items():
        cartridge_list = [ast.literal_eval(cartridge) for cartridge in cartridges]
        array_2d_1.append(cartridge_list[0])

    max_size = 5000

    sequences_count, sequences_array =
generate_possible_array(counts['slot_number_count'], array_2d_1[0])
    history = [[sequence.tolist()] for sequence in sequences_array]

    for row in array_2d_1[1:]:
        _, value = generate_possible_array(counts['slot_number_count'], row)
        if row == array_2d_1[1]:
            result = combined_function(sequences_array, value, array_2d)
            result = limit_size(result, max_size)
            min_time = np.min(result['change_time'])
            min_time_index = np.argmin(result['change_time'])
```

```
            print("Minimum Time in Iteration 1:", min_time)
            print("Sequence for Minimum Time:", result['history'][min_time_index])
            sequences_array = result['sequences']
            pre_change_time_array = result['change_time']
            history = result['history']
        else:
            result = combined_function(sequences_array, value, array_2d,
pre_change_time_array, history)
            result = limit_size(result, max_size)
            min_time = np.min(result['change_time'])
            min_time_index = np.argmin(result['change_time'])
            print("Minimum Time in Current Iteration:", min_time)
            print("Sequence for Minimum Time:", result['history'][min_time_index])
            sequences_array = result['sequences']
            pre_change_time_array = result['change_time']
            history = result['history']
```

8.2.4 第四题

```
import pandas as pd
import pickle
import ast


# 从 Excel 表格中读取数据
def read_packages_from_excel(file_path):
    # 读取 Excel 文件中的所有表格
    xls = pd.ExcelFile(file_path)
    sheet_names = xls.sheet_names  # 获取所有表格的名字

    packages_data = {}
    slot_number = []
    cartridge_number = []
    packages_sequence = []

    for sheet_name in sheet_names:
        df = pd.read_excel(file_path, sheet_name=sheet_name, header=None)

        # 处理表格中的每一行
        if sheet_name == '包装-墨盒-插槽':
            for index, row in df.iterrows():
                if index == 0:  # 跳过标题行
                    continue
                # 获取墨盒编号，假设在第二列
                cartridge_number.append(row[1])
```

```
                    # 获取插槽编号，假设在第三列
                    if not pd.isna(row[2]):
                        slot_number.append(row[2])
            if sheet_name == '包装种类及其所需墨盒':
                for index, row in df.iterrows():
                    if index == 0:  # 跳过标题行
                        continue
                    package_number = row[0]
                    cartridge_data = row[1]
                    if package_number not in packages_data:
                        packages_data[package_number] = []
                    packages_data[package_number].append(cartridge_data)
                    packages_sequence.append(package_number)

        return packages_data, packages_sequence, cartridge_number, slot_number

    if __name__ == '__main__':
        file_path = r'../附件数据（B 题）/附件 4/Ins4_20_40_10.xlsx'
        packages_data, packages_sequence, cartridge_number, slot_number = read_packages_from_excel(file_path)
        sheet_name = "墨盒切换时间"

        # 读取指定工作表的数据
        df = pd.read_excel(file_path, sheet_name=sheet_name)

        # 将数据存储在二维数组中
        data_array = df.values

        # 计算每行的后几个元素的数量（假设为 n）
        n = len(data_array)  # 假设后 5 个元素需要赋值给 array_2d

        # 创建一个空的二维数组
        array_2d = []

        # 将 data_array 每行的后 n 个元素赋值给 array_2d
        for row in data_array:
            array_2d_row = row[-n:]  # 获取每行的后 n 个元素
            array_2d.append(array_2d_row)


        # print(packages_data)
        # print(packages_sequence)
        # print(cartridge_number)
        # print(slot_number)
```

```
    # print(array_2d)
    #  保存数据到文件
    with open('../其他/data.pkl', 'wb') as f:
        pickle.dump({
            'array_2d':array_2d,
            'packages_data': packages_data,
            'packages_sequence': packages_sequence,
            'cartridge_number': cartridge_number,
            'slot_number': slot_number,
            'counts': {
                'array_2d_count':len(array_2d),
                'packages_sequence_count': len(packages_sequence),
                'cartridge_sequence_count': len(cartridge_number),
                'slot_number_count': len(slot_number)
            }
        }, f)

    # print("Data has been saved to data.pkl")
import pickle
import ast
import math
import itertools
import numpy as np
import random

import pandas as pd


def culculate_changetime(i1, j1, pre, now, array):
    a1 = pre[i1]
    a2 = now[j1]
    time = 0
    for i in range(len(a1)):
        if a1[i] == 0 or a2[i] == 0:
            time += 0
        else:
            time += array[a1[i] - 1][a2[i] - 1]
    return time


def generate_possible_array(length, values):
    num_values = len(values)
    index_combinations = itertools.combinations(range(length), num_values)
    sequences = []
```

```
        for indices in index_combinations:
            sequence = [0] * length
            for value, index in zip(values, indices):
                sequence[index] = value
            sequences.append(sequence)

        sequences_array = np.array(sequences)
        return len(sequences), sequences_array


    def combined_function(sequences_array, value, array_2d, pre_change_time_array=None, pre_history=None):
        k1 = sequences_array.shape[0]
        k2 = value.shape[0]

        combined_length = k1 * k2
        length = sequences_array.shape[1]

        result_sequences = np.zeros((combined_length, length), dtype=int)
        change_time_array = np.zeros((combined_length, 1), dtype=int)
        history = []

        index = 0
        for i in range(k1):
            for j in range(k2):
                new_sequence = sequences_array[i].copy()
                for pos in range(length):
                    if value[j][pos] != 0:
                        new_sequence[pos] = value[j][pos]
                result_sequences[index] = new_sequence

                if pre_change_time_array is not None:
                    change_time_array[index] = culculate_changetime(i, j, sequences_array, value, array_2d) + pre_change_time_array[i]
                    history.append(pre_history[i] + [new_sequence.tolist()])
                else:
                    change_time_array[index] = culculate_changetime(i, j, sequences_array, value, array_2d)
                    history.append([new_sequence.tolist()])
                index += 1

        return {
            'sequences': result_sequences,
```

```
            'change_time': change_time_array,
            'history': history
        }


def limit_size(result, max_size):
        indices = np.argsort(result['change_time'], axis=0).flatten()
        limited_indices = indices[:max_size]
        limited_sequences = result['sequences'][limited_indices]
        limited_change_time = result['change_time'][limited_indices]
        limited_history = [result['history'][i] for i in limited_indices]

        return {
            'sequences': limited_sequences,
            'change_time': limited_change_time,
            'history': limited_history
        }


def cuculate_number(n, r):
        return math.comb(n, r)


def load_data(file_path):
        with open(file_path, 'rb') as f:
            data = pickle.load(f)
        return data


data = load_data('../其他/data.pkl')

packages_data = data['packages_data']
packages_sequence = data['packages_sequence']
cartridge_sequence = data['cartridge_number']
slot_number = data['slot_number']
counts = data['counts']
array_2d = data['array_2d']

array_2d_1 = []
for package_number, cartridges in packages_data.items():
        cartridge_list = [ast.literal_eval(cartridge) for cartridge in cartridges]
        array_2d_1.append(cartridge_list[0])
```

```
    def one_round(changed_array):
        max_size = 300
        sequences_count, sequences_array = generate_possible_array(counts['slot_number_count'], changed_array[0])
        history = [[sequence.tolist()] for sequence in sequences_array]

        for row in changed_array[1:]:
            _, value = generate_possible_array(counts['slot_number_count'], row)
            if row == changed_array[1]:
                result = combined_function(sequences_array, value, array_2d)
                result = limit_size(result, max_size)
                min_time = np.min(result['change_time'])
                min_time_index = np.argmin(result['change_time'])
                sequences_array = result['sequences']
                pre_change_time_array = result['change_time']
                history = result['history']
            else:
                result = combined_function(sequences_array, value, array_2d, pre_change_time_array, history)
                result = limit_size(result, max_size)
                min_time = np.min(result['change_time'])
                min_time_index = np.argmin(result['change_time'])
                sequences_array = result['sequences']
                pre_change_time_array = result['change_time']
                history = result['history']

        return min_time, result['history'][min_time_index]


    def simulated_annealing(array_2d_1, initial_temp, cooling_rate, max_iterations):
        current_solution = array_2d_1[:]
        current_value, _ = one_round(current_solution)
        best_solution = current_solution[:]
        best_value = current_value

        temp = initial_temp

        for iteration in range(max_iterations):
            # Generate a neighbor by swapping two random elements
            new_solution = current_solution[:]
            i, j = random.sample(range(len(new_solution)), 2)
            new_solution[i], new_solution[j] = new_solution[j], new_solution[i]

            new_value, _ = one_round(new_solution)
```

```
            # Calculate acceptance probability
            if new_value < current_value or random.random() < math.exp((current_value
- new_value) / temp):
                current_solution = new_solution
                current_value = new_value

                if new_value < best_value:
                    best_solution = new_solution
                    best_value = new_value

            # Cool down the temperature
            temp *= cooling_rate

            print(f'Iteration {iteration + 1}: Current Value = {current_value}, Best Value
= {best_value}")

        return best_solution, best_value


    # Parameters for simulated annealing
    initial_temp = 10000
    cooling_rate = 0.96
    max_iterations = 1000

    best_sequence, best_value = simulated_annealing(array_2d_1, initial_temp,
cooling_rate, max_iterations)

    print("\nBest sequence found:", best_sequence)
    print("Minimum adjustment time:", best_value)

    # Load the Excel file
    df = pd.read_excel(file_path, sheet_name='包装种类及其所需墨盒')

    # Create a dictionary for mapping cartridge sequences to package type IDs
    package_types_dict = {}
    for idx, row in df.iterrows():
        package_types_dict[tuple(ast.literal_eval(row['所需墨盒编号']))] = int(row['包装
种类编号'])

    # Convert best sequence to package type IDs
    package_type_indices = [package_types_dict[tuple(seq)] for seq in best_sequence]

    print("\nBest sequence found (package types):", package_type_indices)
```