基于整数规划的柔性印刷机墨盒切换问题

摘 要

柔性版印刷时，需要减少总墨盒切换时间来提升印刷效率。本文针对不同的需求，分别以最小化总切换次数与最小化总切换时间为目标，构建 0-1 整数规划模型，并对问题一到问题五提供了两种建模方案：对于方案一，本文设计遗传算法求解；对于方案二，本文调用 Gurobi 求解；同时，对问题一与问题二设计贪婪算法求解。最后，比较两种建模方案以及不同求解算法之间的优劣。

针对问题一，以最小化墨盒总切换次数为目标，构建两种不同方案的 **0-1 整数规划**模型。**贪婪策略**设计时，尽可能保留未来使用频率高的墨盒，优先替换未来出现频次最低的墨盒。贪婪策略求得问题一的解为：**5、8、48、58、133**（次）；方案一采用**遗传算法**得到的解为：**5、8、68、77、158**（次）；方案二调用 **Gurobi** 求得解为：**5、8、48、58、130**（次）。表明问题一中贪婪策略的设计表现良好，其时间复杂度为$\boldsymbol{O(n\cdot(s^2+m\log m))}$。

针对问题二，以最小化墨盒总切换时间为目标，构建两种不同方案的 **0-1 整数规划**模型。贪婪策略设计时，定义墨盒切换的代价，切换代价与时间和被切换对象未来出现的频率有关，优先替换切换代价最大的墨盒。贪婪策略求得问题二的解为：**40、68、130、315、407**（min）；对方案一采用遗传算法求解结果为：**32、51、256、419、639**（min）；对方案二调用 **Gurobi** 求得解为：**32、51、111、上界 209、上界 358**（min）。其中贪婪策略的时间复杂度为$\boldsymbol{O(k^2+n\cdot s\cdot m)}$。

针对问题三，不同包装的墨盒顺序不同。在遗传算法对方案一求解时，筛选出满足问题三条件的可行解；在方案二中新增**紧前约束**限定墨盒排布顺序。方案一求解结果为：**56、288、396、438**（min），方案二调用 **Gurobi** 对附件一求解的结果为 **56**（min）。

针对问题四，印刷之前包装种类的印刷顺序不确定。在方案二中，新增**虚拟起点**与**虚拟终点**，构建基于 **TSP** 问题的整数规划模型对原问题进行简化，并调用 **Gurobi** 对方案二求解，作为原问题的近似解。对方案一进行启发式求解结果为：**21、37、70、367**（min），方案二求解结果为：**37、64、98、上界 264**（min）。

针对问题五，多台清洗机共同完成任务。改进遗传算法并对方案一求解，结果为：**16、158、237、285**（min）。方案二在问题四的基础上增加机器数量，使由虚拟起点出发到达虚拟终点的机器数量不超过 M。构建基于 **M-TSP** 的整数规划模型对问题五进行化简，并取 M 等于 2，对附件一求解，其最小总切换时间为 **14**（min）。

关键词：0-1 整数规划　贪婪策略　遗传算法　Gurobi　M-TSP

一、问题重述

1.1 问题背景

柔性印刷机墨盒切换的问题是指在柔性印刷生产过程中，由于需要印刷不同颜色或类型的产品，因此需要更换墨盒的过程。这一过程中涉及到墨盒的取出和安装，以及传墨辊和滚筒的清洗，耗时较长，将放置在插槽中的墨盒更换为另一种颜色的墨盒并完成清洗操作所花费的时间称为切换时间。针对某个插槽来说，将放置在该插槽中的墨盒取出，将另一个墨盒放入该插槽中的操作称为一次切换操作。而且不同颜色墨盒之间的切换时间可能不同。传墨辊和滚筒采用喷雾清洗机进行清洗，一台喷雾清洗机一次只能清洗一套传墨辊和滚筒，所以通过减小总切换时间来大幅提升印刷效率[2]。

柔性印刷机墨盒切换的数学优化问题可以使用多种方法来处理。以下是一些常见的方法，整数规划、动态规划、贪婪算法等[1][3]。

1.2 问题的提出

由于不同颜色墨盒之间的切换时间可能不同，喷雾清洗机数量有限等种种条件的限制之下，根据附件 1-5 给定的数据和要求建立恰当的模型来使总切换次数或者时间最小化。

问题一：在只有一台喷雾清洗机、每种包装的墨盒顺序可任意放置的条件下，给定印刷包装的顺序，令总切换次数最小化。

问题二：假设不同颜色墨盒之间的切换时间可能不同，在只有一台喷雾清洗机、每种包装的墨盒顺序可任意放置的条件下，给定印刷包装的顺序，令总切换时间最小化。

问题三：假设不同颜色墨盒之间的切换时间可能不同，在只有一台喷雾清洗机、每种包装的墨盒顺序不同且固定的条件下，给定印刷包装的顺序，令总切换时间最小化。

问题四：假设不同颜色墨盒之间的切换时间可能不同，在只有一台喷雾清洗机、每种包装的墨盒顺序不同且固定的条件下，需先确定印刷包装的顺序，令总切换时间最小化。

问题五：假设不同颜色墨盒之间的切换时间可能不同，在有两台喷雾清洗机、每种包装的墨盒顺序不同且固定的条件下，需先确定印刷包装的顺序，令总切换时间最小化。

二、问题分析

总目标：本问题的主要目标为站在大幅提高柔性印刷机的印刷效率的角度上，通过综合考虑喷雾清洗机数量、墨盒顺序、印刷包装顺序等情况来制定合适的墨盒更换策略，从而减少不同颜色墨盒之间的总切换时间或总切换次数。

2.1 问题一的分析

对于问题一，要求在不考虑包装对应的墨盒顺序，且不考虑不同包装的印刷顺序的前提下，优化包装印刷时每个插槽内墨盒放置的方案，最小化墨盒的总切换次数。本文构建两种基于不同决策变量设置的 0-1 整数规划模型，并分析其对应的约束条件。

由于此类优化问题往往是 NP 难的，在求解时首先考虑设计一种基于贪婪策略的近似算法：在一族印刷任务中，不同包装印刷时所需的墨盒种类可能重复出现，为了尽可能减少此类墨盒的更换次数，在每次印刷新的包装时，判别是否需要更换墨盒，若需要更换，则优先替换那些未来出现频次最低的墨盒，否则保留当前插槽内墨盒组合。

贪婪策略可以很快得到结果，但往往无法保证最优性。因此，对问题一的两种整数规划模型，分别调用 Gurobi 求解器精确求解、设计遗传算法进行启发式求解，并对比分析两种整数规划模型之间，以及各种求解方法之间得到结果的优劣。

2.2 问题二的分析

对于问题二，优化的目标由总切换次数变为总切换时间，相应的两种 0-1 整数规划的目标函数以及部分参数的含义发生改变。由于问题的限制条件并没有发生本质改变，因此约束条件继承问题一中相应约束条件，只是部分约束依据新的目标进行调整。

在问题二的求解中，继承问题一的思想，对两种 0-1 整数规划模型分别调用 Gurobi 精确求解以及启发式求解，并对比两种模型在此问题上的优劣。然而由于优化目标的变化，需要对近似策略进行调整：虽然任意两墨盒之间的切换时间已知，但墨盒切换的策略事先未知。本文定义墨盒切换的代价矩阵，两墨盒之间切换的代价不止与时间有关，也与被切换对象未来出现的频率有关。不同于问题一的是，若需要更换墨盒时，优先更换掉那些切换代价大的墨盒，否则保留原插槽内墨盒组合。最后，本文对上述算法的结果与表现进行简单分析。

2.3 问题三的分析

对于问题三，优化问题的限制条件发生改变，每种包装印刷任务新增内部顺序限制，因此必须增加切换墨盒时的顺序约束。同时，由于此时单个包装任务所需墨盒的组合不能任意排布，需要找到潜在的紧前约束来对任务内部结构进行限制。基于上述改动，提出问题三的整数规划模型，并同样利用 Gurobi 进行精确求解。

2.4 问题四、五的分析

对于问题四与问题五，与问题一到问题三有本质区别：包装种类印刷顺序（即任务顺序）并不知道。问题四与问题五分别考虑用 1 台和 2 台喷雾清洗机（机器），完成（访问）所有任务，需要确定机器访问任务的顺序，使得所有任务均被完成，结合前三问一些限制的基础上，寻求切换时间最小的插槽内墨盒动态组合方式。这与 TSP 问题有很多相似之处。

为此，针对问题四，本文提出基于 TSP 问题的整数规划模型，并借助 Gurobi 求解。而对于问题五，本文尝试考虑更一般的 M 台机器的插槽墨盒动态组合方式问题，提出基于 M-TSP 的整数规划模型，并在 M 取 2 时对问题 5 求解。

三、模型假设

- 忽略将墨盒放入空的插槽中的时间，只考虑墨盒清洗的时间
- 忽略切换印刷包装种类所需的时间
- 假设每种包装所需的墨盒数不超过插槽数，否则此类包装无法印刷

四、符号说明

以下符号仅适用于问题一以及问题二方案一的模型建立：

符号	定义	单位
i	包装种类中的i个包装 $i \in I$	/
j	插槽序号中的j个序号 $j \in J$	/
k	墨盒编号中的k个编号 $k \in K$	/
I	需要加工的包装种类的集合	/
J	可以使用的插槽的集合	/
K	需要使用的墨盒的集合	/
X_{ikj}	表示包装i的墨盒k是否在插槽j上	/
$Y_{k_1k_2j}$	表示是否在插槽j上先执行k_1，后一步执行k_2	/

以下符号适用于问题一以及问题二方案二的模型建立，以及问题三、四、五的模型建立：

符号	定义	单位
i	包装种类中的i个包装 $i \in N$	/
u	墨盒编号中的u个编号 $u \in U$	/
k	插槽序号中的k个序号 $k \in K$	/
N	包装种类的集合(将印刷一种包装看作一次任务)	/
U	需要使用的墨盒的集合	/
K	可以使用的插槽的集合	/
X_{iku}	执行 task i 时，插槽k内是否装着墨盒u	/
Y_{ik}	执行 task i 时，插槽k是否需要更换墨盒	/
S_i	第i个任务所需墨盒的集合	/
$Z_{iku_1u_2}$	task i 的插槽k，是否从墨盒u_1换成墨盒u_2	/

五、模型的建立与求解

5.1 问题一的求解

5.1.1 模型建立

对于问题一，此类优化问题往往是 NP-难的，本文构建两种基于不同决策变量设置的 0-1 整数规划模型，并分析其对应的约束条件。在求解时首先考虑设计一种基于贪婪策略的近似算法，但往往无法保证最优性。因此，对问题一的两种整数规划模型，分别调用 Gurobi 求解器精确求解、设计遗传算法进行启发式求解，并对比分析两种整数规划模型之间，以及各种求解方法之间得到结果的优劣。

- 方案 1

参数说明：$C_{k_1k_2}$为 0-1 变量，表示若k_1和k_2不是同一种墨盒则为 1，否则为 0。

目标函数：总切换次数尽可能少。.

$$min\sum_{i_1}\sum_{i_2}\sum_{k_1\in K_{i_1}}\sum_{k_2\in K_{i_2}}\sum_{j}C_{k_1k_2}Y_{k_1k_2j}$$

约束 1：任务i上的墨盒k必须插在其中一个插槽上。

$$\sum_{j}X_{ikj}=1 \quad \forall i\in I,k\in K_i$$

约束 2：任务i上的墨盒k必须插在不同的插槽上。

$$\sum_{k}X_{ikj}=1 \quad \forall i\in I,j\in J$$

约束 3：

$$Y_{k_1k_2j}\le X_{i_1k_1j}$$

$$\forall i_1\in I,i_2\in I\text{且}i_1\ne i_2,k_1\in K_{i_1},k_2\in K_{i_2},j\in J$$

表示若$X_{i_1k_1j}$为 1，则$Y_{k_1k_2j}$可能为 0 或 1；若$X_{i_1k_1j}$为 0，则$Y_{k_1k_2j}$只能为 0。

约束 4：

$$Y_{k_1k_2j}\le X_{i_2k_2j}$$

$$\forall i_1\in I,i_2\in I\text{且}i_1\ne i_2,k_1\in K_{i_1},k_2\in K_{i_2},j\in J$$

表示若$X_{i_1k_2j}$为 1，则$Y_{k_1k_2j}$可能为 0 或 1；若$X_{i_1k_2j}$为 0，则$Y_{k_1k_2j}$只能为 0。

约束 5：

$$\sum_{k_2}Y_{k_1k_2j}\le 1$$

$$\forall i_1 \in I, i_2 \in I 且 i_1 \neq i_2, k_1 \in K_{i_1}, k_2 \in K_{i_2}, j \in J$$

表示每次墨盒切换时，一种墨盒只能切换到与之不同的最多一种墨盒。

约束 6：保证任务持续进行。

$$X_{i_2k_2j} \cdot X_{i_1k_1j} \cdot Y_{k_1k_2j} \le Y_{k_2k_3j} \cdot X_{i_2k_2j} \cdot X_{i_3k_3j}$$

$$\forall i_1 \in I, i_2 \in I, i_3 \in I 且 i_2 - i_1 = 1, i_3 - i_2 = 1$$

综上，可得到问题一的 0-1 整数规划模型，如下所示：

$$min \sum_{i_1} \sum_{i_2} \sum_{k_1 \in K_{i_1}} \sum_{k_2 \in K_{i_2}} \sum_{j} C_{k_1k_2} Y_{k_1k_2j}$$

$$s.t. \begin{cases} \sum_j X_{ikj} = 1 \quad \forall i \in I, k \in K_i \\ \sum_k X_{ikj} = 1 \quad \forall i \in I, j \in J \\ Y_{k_1k_2j} \le X_{i_1k_1j} \quad \forall i_1 \in I, i_2 \in I 且 i_1 \neq i_2, k_1 \in K_{i_1}, k_2 \in K_{i_2}, j \in J \\ Y_{k_1k_2j} \le X_{i_2k_2j} \quad \forall i_1 \in I, i_2 \in I 且 i_1 \neq i_2, k_1 \in K_{i_1}, k_2 \in K_{i_2}, j \in J \\ \sum_{k_2} Y_{k_1k_2j} \le 1 \quad \forall i_1 \in I, i_2 \in I 且 i_1 \neq i_2, k_1 \in K_{i_1}, k_2 \in K_{i_2}, j \in J \\ X_{i_2k_2j} \cdot X_{i_1k_1j} \cdot Y_{k_1k_2j} \le Y_{k_2k_3j} \cdot X_{i_2k_2j} \cdot X_{i_3k_3j} \quad \forall i_1 \in I, i_2 \in I, i_3 \in I 且 i_2 - i_1 = 1, i_3 - i_2 = 1 \\ X_{ikj} \in \{0,1\}, \quad Y_{k_1k_2j} \in \{0,1\} \end{cases}$$

- **方案二**

目标函数：总切换次数尽可能少。

$$min \sum_{k \in K} \sum_{i \in N} Y_{ik}$$

约束 1：每个 $task\ i$所要求的墨盒集合$S[i]$中的墨盒$= 1$

$$\sum_{k \in K} X_{iks} = 1 \quad \forall i \in N, s \in S_i$$

约束 2：执行每个 $task\ i$时，每种墨盒u最多只装一个。

$$\sum_{k \in K} X_{iku} \le 1 \quad \forall i \in N, u \in U$$

约束 3：执行每个 $task\ i$时，每个插槽k，只装一个墨盒。

$$\sum_{u \in U} X_{iku} \le 1 \quad \forall i \in N, k \in K$$

约束 4：对每个插槽而言，如果当前 $task\ i$的墨盒u和 $task\ i+1$ 的墨盒u相同 右边相减为 0, 否则绝对值为 1,代表更换了墨盒。

$$Y_{ik} \geq X_{i+1,k,u} - X_{iku} \qquad Y_{ik} \geq X_{iku} - X_{i+1,k,u}$$

$$\forall\, u \in U, k \in K, i \in N \text{且} i+1 \in N$$

综上，可得到问题一的模型规划，如下所示：

$$min \sum_{k \in K} \sum_{i \in N} Y_{ik}$$

$$s.t. \begin{cases} \sum_{k \in K} X_{iks} = 1 \quad \forall i \in N, s \in S_i \\ \sum_{k \in K} X_{iku} \leq 1 \quad \forall i \in N, u \in U \\ \sum_{u \in U} X_{iku} \leq 1 \quad \forall i \in N, k \in K \\ Y_{ik} \geq X_{i+1,k,u} - X_{iku} \quad Y_{ik} \geq X_{iku} - X_{i+1,k,u} \quad \forall\, u \in U, k \in K, i \in N \text{且} i+1 \in N \\ X_{iku} \in \{0,1\}, \quad Y_{ik} \in \{0,1\} \end{cases}$$

5.1.2 模型求解

（1）贪婪策略的设计与求解

针对问题一考虑设计一种基于贪婪策略的近似算法：首先，初始化印刷机上的墨盒插槽，并建立一个记录墨盒使用频次的数据结构。然后，从待印刷新任务中选择一个任务。接着，检查当前任务需要的墨盒组合是否已经在插槽中。如果是，则继续印刷；否则，执行下一步。在选择需要替换的墨盒时，优先考虑未来出现频次最低的墨盒，以最小化未来墨盒更换的次数，替换墨盒并更新墨盒使用频次的记录。重复以上步骤，直到所有印刷新任务完成。下面是该算法的流程图：

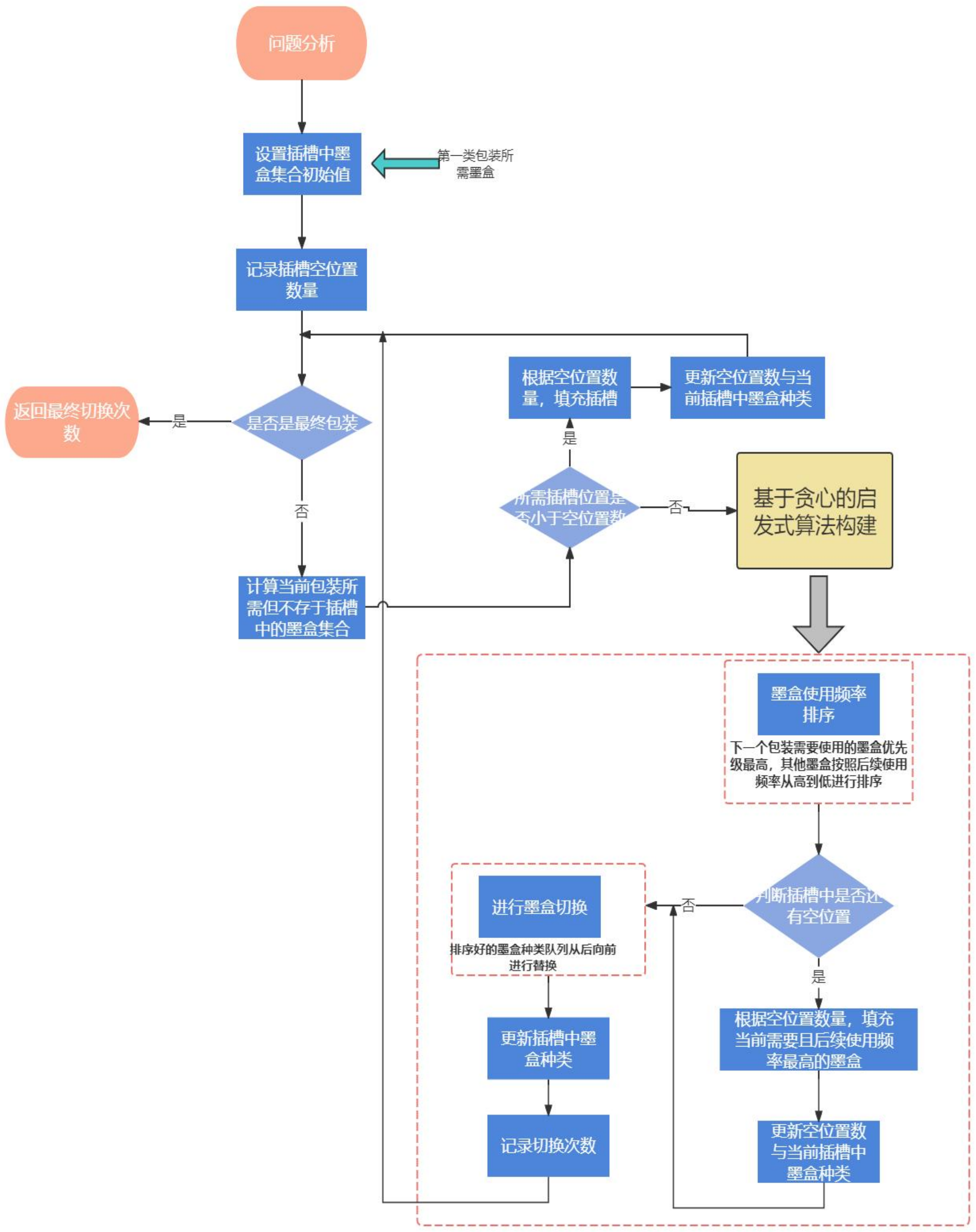

问题分析
设置插槽中墨盒集合初始值
第一类包装所需墨盒
记录插槽空位置数量
是否是最终包装
是
返回最终切换次数
否
计算当前包装所需但不存在于插槽中的墨盒集合
所需插槽位置是否小于空位置数
是
根据空位置数量，填充插槽
更新空位置数与当前插槽中墨盒种类
否
基于贪心的启发式算法构建
墨盒使用频率排序
下一个包装需要使用的墨盒优先级最高，其他墨盒按照后续使用频率从高到低进行排序
判断插槽中是否还有空位置
否
进行墨盒切换
排序好的墨盒种类队列从后向前进行替换
更新插槽中墨盒种类
记录切换次数
是
根据空位置数量，填充当前需要且后续使用频率最高的墨盒
更新空位置数与当前插槽中墨盒种类

图 1 问题一贪婪算法流程图

Step1：将第一个包装所需墨盒集合作为插槽中墨盒的初始值，插槽可以不装满，记录下插槽中的空位置数量。

Step2：因为包装顺序固定，从第二个包装开始进行遍历。

Step3：遍历开始时，首先计算出当前包装所需要的墨盒有哪些不存在在插槽中，设置一个数组来存储这些需要但不存在的墨盒。

Step4：判断这些需要但不存在的墨盒的个数是否小于当前插槽中空位置数量。

如果小于，则直接将这些需要的墨盒插入插槽，更新插槽中空位置的数量以及更新当前插槽中墨盒的种类，直接进入下一次的循环。

如果大于，则进行进一步判断。

Step5：如果大于，先根据空位的数量将插槽装满，这时还剩一些当前包装所需但不存在于插槽的墨盒，这就需要涉及到墨盒的切换问题。

Step6：进行墨盒切换时，首先对插槽中的墨盒按照后续使用频率的高低进行排序，这里是从高到低进行排序。

Step7：将当前包装所需但不存在于插槽的墨盒与插槽中后续使用频率最低的墨盒进行切换，记录下每一次印刷一种包装时，切换的次数。

通过该算法，得到的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

表 1 问题一贪婪算法结果

	实例 1	实例 2	实例 3	实例 4	实例 5
运行结果/次	5	8	48	58	133
运行时间/秒	0.001	0.001	0.003	0.005	0.009

（2）遗传算法的求解

遗传算法是一种基于全局优化的搜索算法，为本文解决复杂组合优化问题以及其他难解优化为题一个有效途径。本文根据问题特点设计遗传算法进行启发式求解，具体代码放至附录，下面仅对最为通用的遗传算法的求解思想做简要介绍：

1，首先，由约束条件可以确定优化问题的可行解域，约束条件越准确，其可行解域就越准确。用字符串表示满足可行域约束的可行解，全体可行解集合构成解空间。

2，构建一个合适的非负函数（遗传算法中往往叫做适应度函数），用来比较每一个可行解的优劣，通常函数值越大的，表示这个可行解越好。

3，设计可迭代策略（遗传算法三大策略为：选择、交叉、变异）不断改变可行解，使得可行解改变后依然在可行域内，改变后的可行解越好，则以越大概率接受这个可行解。

4，设定合适的函数值或者迭代次数作为终止条件。

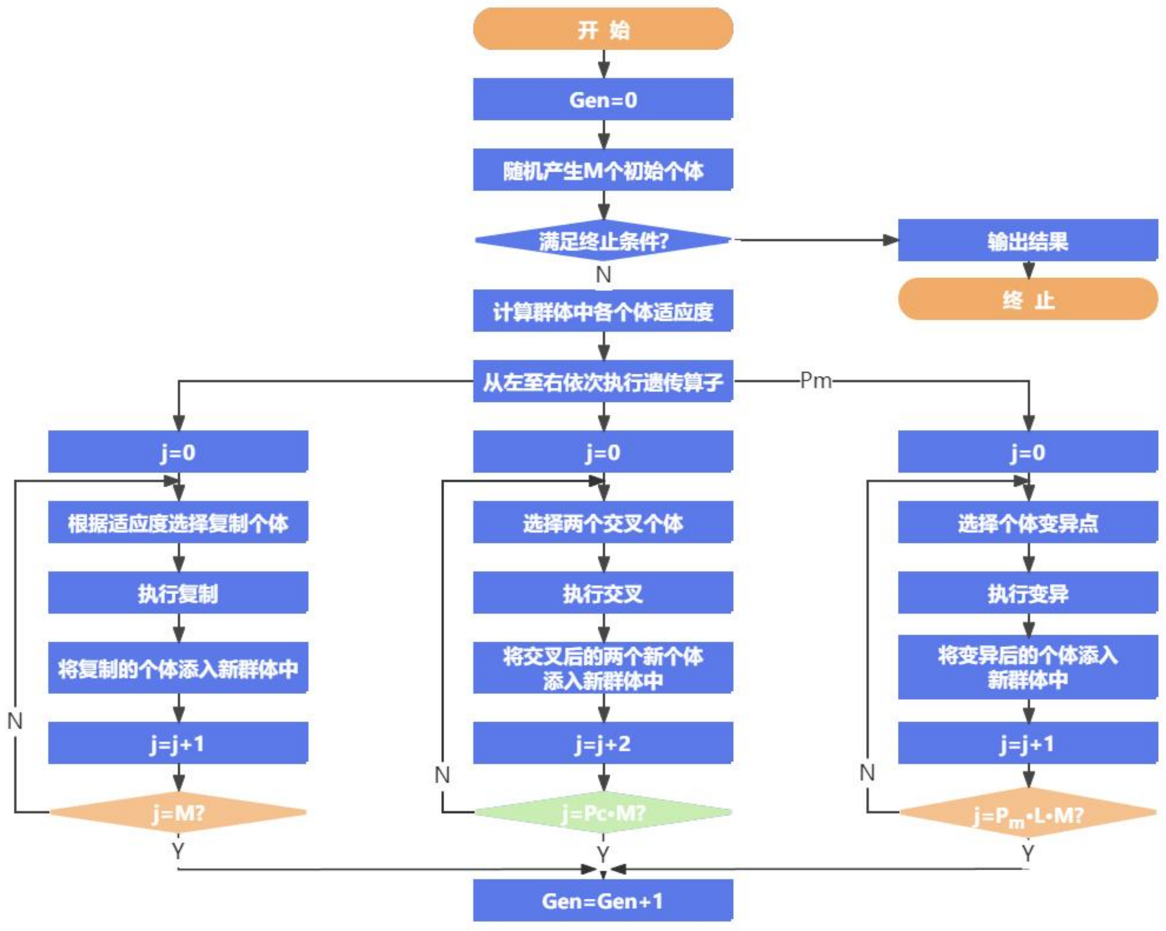


图 2 遗传算法流程图

基于方案一，通过该算法，得到的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

表 2 问题一遗传算法结果

	实例 1	实例 2	实例 3	实例 4	实例 5
运行结果/次	5	8	68	77	158
运行时间/秒	0.452	0.568	0.962	1.368	1.943

（3）Gurobi 求解

Gurobi 是一种大规模数学规划优化器，可以求解的优化类型有：LP、MIP、QP、QCQP、SOCP 等，均表现出强大的优化速度与求解精度。本文使用 Python 调用 Gurobi 11.0.0 版本对模型进行求解，步骤为：

Step1：创建模型对象

Step2：创建变量对象

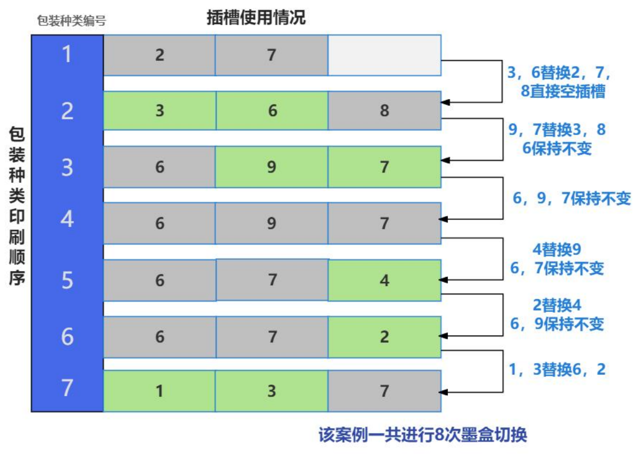
Step3：创建目标函数与约束条件

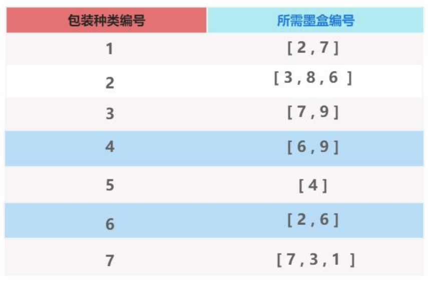
Step4：求解最优化模型并获取相关信息

Gurobi 求解 MIP 问题时算法的主体框架为 B&C。在对模型进行求解前，首先进行预求解，预求解过程可以删除冗余的变量和约束，从而简化求解过程。在完成预求解后，进入框架主体部分，即 B&C：首先进行 Node Selection，并对选取的 Node 进行预求解。之后对选择的 Node 线性松弛，求解松弛问题的解，在这个过程中，Gurobi 不断采用割平面法生成 Cuts，并将 Cuts 返回到当前节点的线性松弛中。对加入 Cuts 的问题线性松弛，并求解。若得到的解为整数解，则剪枝；否则执行启发式算法，利用启发式算法迅速寻求整数可行解，更新 Bound。最后，开始进行分支，生成分支节点，完成一轮操作。不断循环上述操作，直至找到最优解。

其中 Gurobi 进行 Node Selection 时需要设置 BranchDir 参数，其决定了 Gurobi 以何种方式决定下一个节点在 B&C 树中的位置。常见的方式有三种：1，Gurobi 自动选取；2，DFS 方法；3，BFS 方法。本文这里采取 BFS 即广度优先方法选取下一节点位置。

以附件 1 中的实例 2 为例，下图展示了插槽使用情况及墨盒切换情况：





包装种类编号	所需墨盒编号
1	[2,7]
2	[3,8,6]
3	[7,9]
4	[6,9]
5	[4]
6	[2,6]
7	[7,3,1]

图 3 附件 1 实例 2

基于方案二，调用 Gurobi，得到的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

表 3 问题一 Gurobi 结果

	实例 1	实例 2	实例 3	实例 4	实例 5
运行结果/次	5	8	48	58	130
运行时间/秒	0.0	0.0	1190	660	1257

5.1.3 结果分析

表 4 问题一结果汇总

		实例 1	实例 2	实例 3	实例 4	实例 5
贪婪策略	运行结果/次	5	8	48	58	133
	运行时间/秒	0.001	0.001	0.003	0.005	0.009
遗传算法	运行结果/次	5	8	68	77	158
	运行时间/秒	0.452	0.568	0.962	1.368	1.943
求解器	运行结果/次	5	8	48	58	130
	运行时间/秒	0.0	0.0	1190	660	1257

一、运行结果对比

贪婪策略、遗传算法和 Gurobi 求解器在实例 1 和实例 2 中的运行结果大致相近，均为 5 次和 8 次。这表明在小规模问题上，三种方法均能达到理想效果。

在实例 3、实例 4 和实例 5 中，贪婪策略和 Gurobi 求解器的运行结果相近，而遗传算法的运行结果明显更差。这表明在处理更复杂或更大规模的问题时，遗传算法可能需要更多的迭代次数来找到满意的解。

二、运行时间对比

贪婪策略的运行时间极短，从 0.001 秒至 0.009 秒不等。相比之下，遗传算法和 Gurobi 求解器的运行时间明显较长。

遗传算法的运行时间虽然比贪婪策略长，但相对于 Gurobi 求解器在实例 3、4、5 中的运行时间，它仍是较短的。

Gurobi 求解器在实例 3、4、5 中的运行时间分别为 1190 秒、660 秒和 1257 秒，显著长于其他两种算法。这表明在处理大规模或复杂性更高的问题时，Gurobi 求解器需要更多的计算资源来找到高精度解。

5.2 问题二的求解

5.2.1 模型建立

在问题二的求解中，继承问题一的思想，对两种 0-1 整数规划模型分别调用 Gurobi 精确求解以及启发式求解，并对比两种模型在此问题上的优劣。然而由于优化目标的变化，需要对近似策略进行调整，定义墨盒切换的代价矩阵。不同于问题一的是，若需要更换墨盒时，优先更换掉那些切换代价大的墨盒，否则保留原插槽内墨盒组合。最后，本文对上述算法的结果与表现进行简单分析。

- **方案一**

对于问题二，目标由最小化总切换次数变为总切换时间，因此参数和目标函数需要发生改变，而约束条件并没有发生改变。继承问题一的思想，对贪婪策略进行调整。

参数说明：$D_{k_1k_2}$为墨盒之间的切换时间，切换时间单位为分钟。例如，将放置在插槽中的墨盒 1 切换为墨盒 2 的切换时间为 10，则此时$D_{k_1k_2}=10$。

目标函数：总切换时间尽可能少。

$$min\sum_{i_1}\sum_{i_2}\sum_{k_1\in K_{i_1}}\sum_{k_2\in K_{i_2}}\sum_{j}D_{k_1k_2}Y_{k_1k_2j}$$

约束条件：同问题一方案一。

综上，可得到问题二的 0-1 整数规划模型，如下所示：

$$min\sum_{i_1}\sum_{i_2}\sum_{k_1\in K_{i_1}}\sum_{k_2\in K_{i_2}}\sum_{j}D_{k_1k_2}Y_{k_1k_2j}$$

$$s.t.\begin{cases}\sum_{j}X_{ikj}=1 \quad \forall i\in I,k\in K_i\\ \sum_{k}X_{ikj}=1 \quad \forall i\in I,j\in J\\ Y_{k_1k_2j}\le X_{i_1k_1j} \quad \forall i_1\in I,i_2\in I\text{且}i_1\neq i_2,k_1\in K_{i_1},k_2\in K_{i_2},j\in J\\ Y_{k_1k_2j}\le X_{i_2k_2j} \quad \forall i_1\in I,i_2\in I\text{且}i_1\neq i_2,k_1\in K_{i_1},k_2\in K_{i_2},j\in J\\ \sum_{k_2}Y_{k_1k_2j}\le 1 \quad \forall i_1\in I,i_2\in I\text{且}i_1\neq i_2,k_1\in K_{i_1},k_2\in K_{i_2},j\in J\\ X_{i_2k_2j}\cdot X_{i_1k_1j}\cdot Y_{k_1k_2j}\le Y_{k_2k_3j}\cdot X_{i_2k_2j}\cdot X_{i_3k_3j} \quad \forall i_1\in I,i_2\in I,i_3\in I\text{且}i_2-i_1=1,i_3-i_2=1\\ X_{ikj}\in\{0,1\}, \quad Y_{k_1k_2j}\in\{0,1\}\end{cases}$$

- **方案二**

目标函数：总切换时间尽可能少。

$$min\sum_{u_2\in U}\sum_{u_1\in U}\sum_{k\in K}\sum_{i\in N}T_{[u_1][u_2]}\cdot Z_{iku_1u_2}$$

约束 1-3：同问题一方案二。

约束 4： 如果当前$task\ i$选中墨盒u_1和 $task\ i+1$ 选中墨盒u_2, $X_{iku_1}+X_{i+1,k,u_2}=2$, 那么减 1 等于 1，说明$u_1$变换为$u_2$，若$u_1$和$u_2$同色, 则$T_{[u_1][u_2]}=0$，否则需要花费时间更换。

$$Z_{iku_1u_2}\ge X_{iku_1}+X_{i+1,k,u_2}-1$$

$$\forall\ u_2\in U,u_1\in U,k\in K,i\in N\text{且}i+1\in N$$

综上，可得到问题二的模型规划，如下所示：

$$min\sum_{u_2\in U}\sum_{u_1\in U}\sum_{k\in K}\sum_{i\in N}T_{[u_1][u_2]}\cdot Z_{iku_1u_2}$$

$$
s.t.\begin{cases}
\sum_{k\in K} X_{iks} = 1 \qquad \forall i\in N, s\in S_i \\
\sum_{k\in K} X_{iku} \le 1 \qquad \forall i\in N, u\in U \\
\sum_{u\in U} X_{iku} \le 1 \qquad \forall i\in N, k\in K \\
Z_{iku_1u_2} \ge X_{iku_1} + X_{i+1,k,u_2} - 1 \qquad \forall\, u_2\in U, u_1\in U, k\in K, i\in N \text{且} i+1\in N \\
X_{iku}\in\{0,1\}, \qquad Y_{ik}\in\{0,1\}
\end{cases}
$$

5.2.2 模型求解

（1）贪婪策略的设计与求解

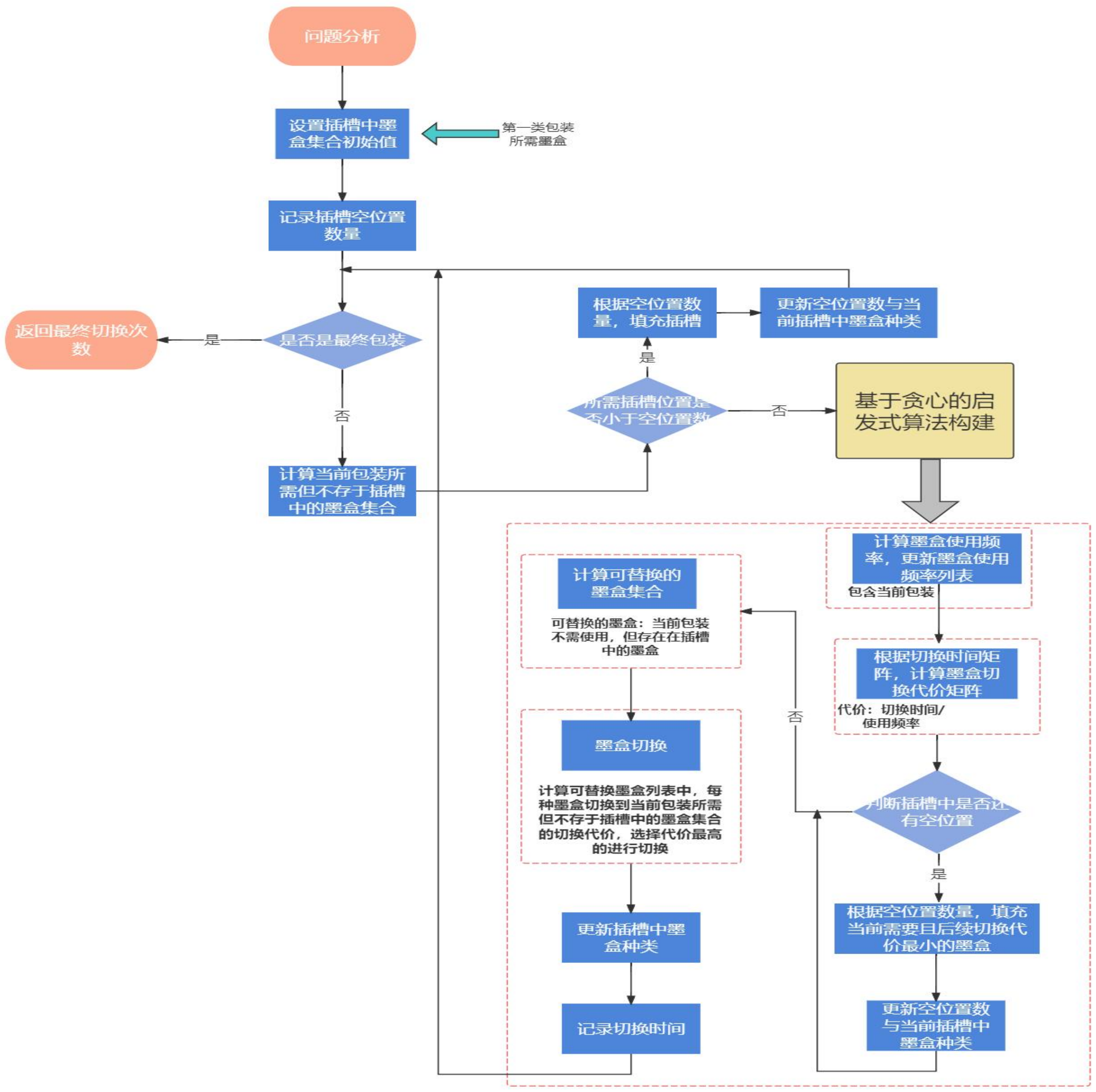

问题分析
设置插槽中墨盒集合初始值
第一类包装所需墨盒
记录插槽空位置数量
是否是最终包装
是
返回最终切换次数
否
计算当前包装所需但不存于插槽中的墨盒集合
所需插槽位置是否小于空位置数
是
根据空位置数量，填充插槽
更新空位置数与当前插槽中墨盒种类
否
基于贪心的启发式算法构建
计算墨盒使用频率，更新墨盒使用频率列表
包含当前包装
根据切换时间矩阵，计算墨盒切换代价矩阵
代价：切换时间/使用频率
判断插槽中是否还有空位置
是
根据空位置数量，填充当前需要且后续切换代价最小的墨盒
更新空位置数与当前插槽中墨盒种类
否
计算可替换的墨盒集合
可替换的墨盒：当前包装不需使用，但存在在插槽中的墨盒
墨盒切换
计算可替换墨盒列表中，每种墨盒切换到当前包装所需但不存于插槽中的墨盒集合的切换代价，选择代价最高的进行切换
更新插槽中墨盒种类
记录切换时间

图 4 问题二贪婪算法流程图

通过该算法，得到的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

表 5 问题二贪婪算法结果

	实例 1	实例 2	实例 3	实例 4	实例 5
运行结果/分钟	40	68	130	315	407
运行时间/秒	0. 001	0. 001	0. 023	0. 047	0. 116

（2）遗传算法的求解

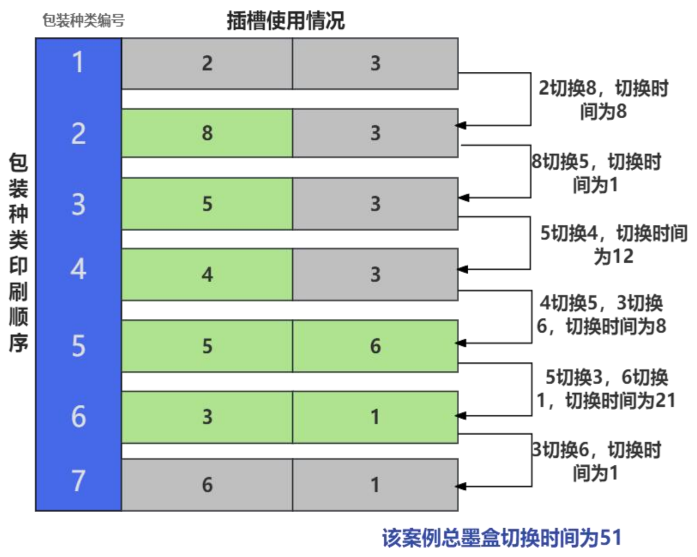
基于方案一，通过该算法，得到的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

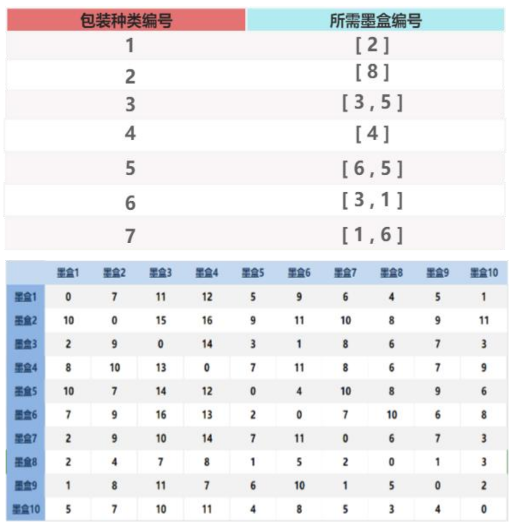
表 6 问题二遗传算法结果

	实例 1	实例 2	实例 3	实例 4	实例 5
运行结果/分钟	32	51	256	419	639
运行时间/秒	0. 472	0. 528	0. 83	1. 438	2. 152

（3）Gurobi 求解

以附件 2 中的实例 2 为例，下图展示了插槽使用情况及墨盒切换时间情况：





包装种类编号	所需墨盒编号
1	[2]
2	[8]
3	[3,5]
4	[4]
5	[6,5]
6	[3,1]
7	[1,6]

	墨盒1	墨盒2	墨盒3	墨盒4	墨盒5	墨盒6	墨盒7	墨盒8	墨盒9	墨盒10
墨盒1	0	7	11	12	5	9	6	4	5	1
墨盒2	10	0	15	16	9	11	10	8	9	11
墨盒3	2	9	0	14	3	1	8	6	7	3
墨盒4	8	10	13	0	7	11	8	6	7	9
墨盒5	10	7	14	12	0	4	10	8	9	6
墨盒6	7	9	16	13	2	0	7	10	6	8
墨盒7	2	9	10	14	7	11	0	6	7	3
墨盒8	2	4	7	8	1	5	2	0	1	3
墨盒9	1	8	11	7	6	10	1	5	0	2
墨盒10	5	7	10	11	4	8	5	3	4	0

图 5 附件 2 实例 2

基于方案二，通过调用 Gurobi，得到的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

表 7 问题二 Gurobi 结果

	实例 1	实例 2	实例 3	实例 4
运行结果/分钟	32	51	111	上界 209
运行时间/秒	0.0	0.0	645	2h

5.2.3 结果分析

表 8 问题二结果汇总

		实例 1	实例 2	实例 3	实例 4	实例 5
贪婪策略	运行结果/分钟	40	68	130	315	407
	运行时间/秒	0.001	0.001	0.023	0.047	0.116
遗传算法	运行结果/分钟	32	51	256	419	639
	运行时间/秒	0.472	0.528	0.83	1.438	2.152
Gurobi	运行结果/分钟	32	51	111	上界 209	/
	运行时间/秒	0.0	0.0	645	2h	/

一、运行结果对比

在实例 1 和实例 2 中，贪婪策略、遗传算法和 Gurobi 的运行结果相差不大。这表明在小规模问题上，三种方法都能得到理想效果。

在实例 3 中，贪婪策略和 Gurobi 的运行结果出现显著差异。贪婪策略的结果为 130 分钟，而 Gurobi 的结果为 111 分钟，略优于贪婪策略。遗传算法的结果为 256 分钟，表现最差。

在实例 4 中，贪婪策略的结果为 315 分钟，而遗传算法和 Gurobi 的结果分别为 419 分钟和上界 209 分钟。这表明在处理更复杂的问题时，遗传算法和 Gurobi 可能难以找到最优解，而贪婪策略依然能保持相对较好的性能。

二、运行时间对比

贪婪策略的运行时间极短，从 0.001 秒至 0.116 秒不等，几乎可以忽略不计。这表明贪婪策略在计算效率上具有显著优势。

遗传算法的运行时间相对较长，从 0.472 秒至 2.152 秒不等。虽然比贪婪策略长，但在可接受范围内，表明遗传算法在处理中等规模问题时仍具有一定的效率。

Gurobi 在实例 1 和实例 2 中的运行时间为 0 秒，这可能是由于问题规模极小，Gurobi 能够迅速找到最优解。然而，在实例 3 中，Gurobi 的运行时间显著增加至 645 秒，这表明在处理更大规模的问题时，Gurobi 需要更多的计算资源。在实例 4 中，Gurobi 的运行时间更是长达 2 小时（即 7200 秒），进一步证实了其在处理复杂问题时的计算负担。

5.3 问题三的求解

5.3.1 模型建立

对于问题三，每种包装印刷任务新增内部顺序限制，因此必须增加切换墨盒时的顺序约束。同时，由于此时单个包装任务所需墨盒的组合不能任意排布，需要找到潜在的紧前约束来对任务内部结构进行限制。

目标函数：总切换时间尽可能少。

$$min\sum_{u_2\in U}\sum_{u_1\in U}\sum_{k\in K}\sum_{i\in N}T_{[u_1][u_2]}\cdot Z_{iku_1u_2}$$

约束 1-3：同问题一方案二。

约束 4：紧前约束：对每个$task\ i$内部，假设是[3,2,1]，那么说明墨盒 3 必须在墨盒 2 选定之前, 插在插槽上, (3,2), (2,1) 两对紧前约束。

$$\sum_{k_i=1}^{k}X_{ik_1S_{i,idx-1}}\geq X_{ikS_{i,idx}}\qquad \forall k\in K, idx\in[2,|S_i|]\cap Z^+$$

其中，$|S_i|$表示S_i的长度。

约束 5：换墨盒约束。

$$Z_{iku_1u_2}\geq X_{iku_1}+X_{i+1,k,u_2}-1$$

$$\forall\ u_2\in U,u_1\in U,k\in K,i\in N\text{且}i+1\in N$$

综上，可得到问题三的模型规划，如下所示：

$$min\sum_{u_2\in U}\sum_{u_1\in U}\sum_{k\in K}\sum_{i\in N}T_{[u_1][u_2]}\cdot Z_{iku_1u_2}$$

$$s.t.\begin{cases}\sum_{k\in K}X_{iks}=1 \qquad \forall i\in N,s\in S_i\\ \sum_{k\in K}X_{iku}\leq 1 \qquad \forall i\in N,u\in U\\ \sum_{u\in U}X_{iku}\leq 1 \qquad \forall i\in N,k\in K\\ \sum_{k_i=1}^{k}X_{ik_1S_{i,idx-1}}\geq X_{ikS_{i,idx}} \qquad \forall k\in K, idx\in[2,|S_i|]\cap Z^+\\ Z_{iku_1u_2}\geq X_{iku_1}+X_{i+1,k,u_2}-1 \qquad \forall\ u_2\in U,u_1\in U,k\in K,i\in N\text{且}i+1\in N\\ X_{iku}\in\{0,1\},\qquad Y_{ik}\in\{0,1\}\end{cases}$$

5.3.2 模型求解

（1）遗传算法求解

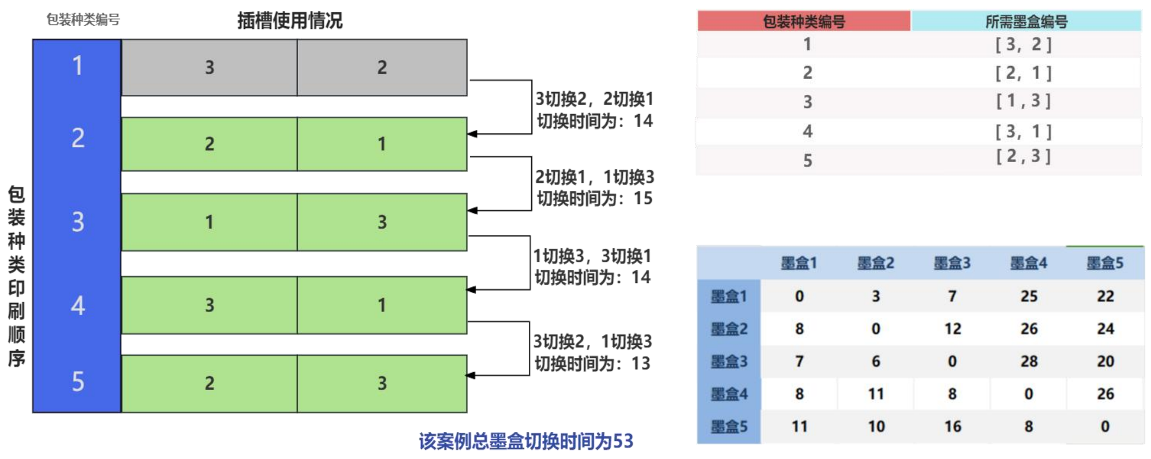
基于方案一，设计该算法，得到问题三的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

表 9 问题三遗传算法结果

	实例 1	实例 2	实例 3	实例 4
运行结果/分钟	56	288	396	438
运行时间/秒	0.470	0.826	1.431	1.955

（2）Gurobi 求解

基于方案二，以附件 3 中的实例 1 为例，下图展示了插槽使用情况及墨盒切换时间情况：



包装种类编号	所需墨盒编号
1	[3, 2]
2	[2, 1]
3	[1 , 3]
4	[3, 1]
5	[2 , 3]

	墨盒1	墨盒2	墨盒3	墨盒4	墨盒5
墨盒1	0	3	7	25	22
墨盒2	8	0	12	26	24
墨盒3	7	6	0	28	20
墨盒4	8	11	8	0	26
墨盒5	11	10	16	8	0

图 6 附件 3 实例 1

5.4 问题四的求解

5.4.1 模型建立

问题四考虑用 1 台喷雾清洗机（机器），完成（访问）所有任务，需要确定机器访问任务的顺序，使得所有任务均被完成。为此，针对问题四，本文借助求解 TSP 问题的建模思想构建整数规划模型。

本问题中，N用链表的形式存储，n表示$listN$的长度。Y_{ij}为 0-1 变量，表示是否执行完i再执行j：若值为 1，表示执行完 i 后再执行 j 变量，否则为 0。构造虚拟任务起点 0 以及虚拟任务终点$n+1$。统计到达$task\ i$时累计完成任务数V_i。

目标函数：总切换时间尽可能少。

$$min\sum_{u_2\in U}\sum_{u_1\in U}\sum_{k\in K}\sum_{i\in N}T_{[u_1][u_2]}\cdot Z_{iku_1u_2}$$

约束 1-3：同问题一方案二。

约束 4：起点约束，从 0 点出发。

$$\sum_{j=N}Y_{0j}=1$$

约束 5：终点约束，回到$n+1$ 点。

$$\sum_{j=N}Y_{j,n+1}=1$$

约束 6：每个印刷任务i入度等于出度为 1，等于每个任务必须且只能执行一次。

$$\sum_{\substack{j\neq i\\ j=N\cup\{0\}}}Y_{ji}=\sum_{\substack{j\neq i\\ j=N\cup\{n+1\}}}Y_{ij}$$

$$\sum_{\substack{j\neq i\\ j=N\cup\{0\}}}Y_{ji}=1$$

约束 7：

$$V_j\geq V_i+1-n(1-Y_{ij})$$

$$\forall j\in N\cup\{n+1\},\forall i\in N\cup\{0\},且i\neq j$$

约束 8：虚拟任务起点 0 的次数为 0。

$$V_0=0$$

约束 9：紧前约束

$$\sum_{k_i=1}^{k}X_{ik_1S_{i,idx-1}}\geq X_{ikS_{i,idx}}\qquad\forall k\in K,idx\in[2,|S_i|]\cap Z^+$$

其中，$|S_i|$表示S_i的长度。

约束 10：换墨盒约束。

$$Z_{iku_1u_2}\geq X_{iku_1}+X_{i+1,k,u_2}-1$$

$$\forall\, u_2\in U,u_1\in U,k\in K,i\in N且i+1\in N$$

综上，可得到问题四的模型规划，如下所示：

$$min\sum_{u_2\in U}\sum_{u_1\in U}\sum_{k\in K}\sum_{i\in N}T_{[u_1][u_2]}\cdot Z_{iku_1u_2}$$

$$
s.t.\begin{cases}
\sum\limits_{k\in K} X_{iks}=1 \quad \forall i\in N, s\in S_i \\
\sum\limits_{k\in K} X_{iku}\le 1 \quad \forall i\in N, u\in U \\
\sum\limits_{u\in U} X_{iku}\le 1 \quad \forall i\in N, k\in K \\
\sum\limits_{j=N} Y_{0j}=1 \\
\sum\limits_{j=N} Y_{j,n+1}=1 \\
\sum\limits_{\substack{j\neq i\\ j=N\cup\{0\}}} Y_{ji} = \sum\limits_{\substack{j\neq i\\ j=N\cup\{n+1\}}} Y_{ij} \qquad \sum\limits_{\substack{j\neq i\\ j=N\cup\{0\}}} Y_{ji}=1 \\
V_j\ge V_i+1-n(1-Y_{ij}) \quad \forall j\in N\cup\{n+1\}, \forall i\in N\cup\{0\}, 且 i\neq j \\
V_0=0 \\
\sum\limits_{k_i=1}^{k} X_{ik_1S_{i,idx-1}}\ge X_{ikS_{i,idx}} \quad \forall k\in K, idx\in[2,|S_i|]\cap Z^+ \\
Z_{iku_1u_2}\ge X_{iku_1}+X_{i+1,k,u_2}-1 \quad \forall\, u_2\in U, u_1\in U, k\in K, i\in N 且 i+1\in N \\
X_{iku}\in\{0,1\}, \qquad Y_{ik}\in\{0,1\}
\end{cases}
$$

5.4.2 模型求解

（1）遗传算法求解

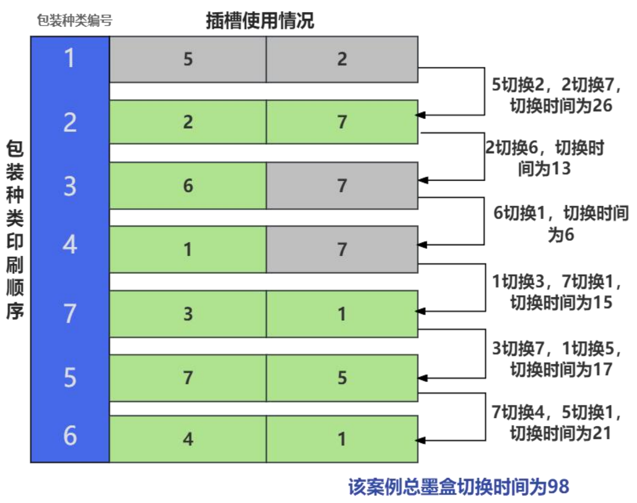
基于方案一，设计该算法，得到问题四的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

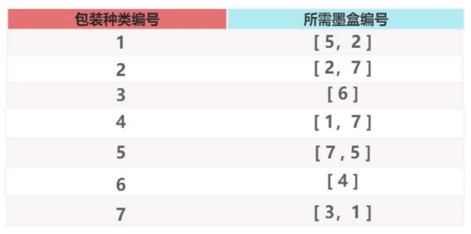
表 10 问题四遗传算法结果

	实例 1	实例 2	实例 3	实例 4
运行结果/分钟	21	37	70	367
运行时间/秒	0.477	0.529	0.545	1.433

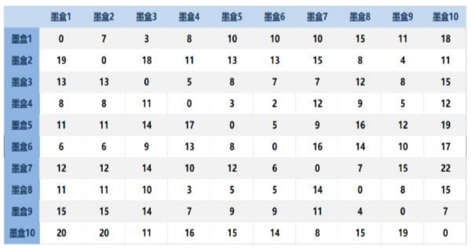
（2）Gurobi 求解

基于方案二，以附件 4 中的实例 3 为例，下图展示了插槽使用情况及墨盒切换时间情况：





包装种类编号	所需墨盒编号
1	[5, 2]
2	[2, 7]
3	[6]
4	[1, 7]
5	[7,5]
6	[4]
7	[3, 1]



	墨盒1	墨盒2	墨盒3	墨盒4	墨盒5	墨盒6	墨盒7	墨盒8	墨盒9	墨盒10
墨盒1	0	7	3	8	10	10	10	15	11	18
墨盒2	19	0	18	11	13	13	15	8	4	11
墨盒3	13	13	0	5	8	7	7	12	8	15
墨盒4	8	8	11	0	3	2	12	9	5	12
墨盒5	11	11	14	17	0	5	9	16	12	19
墨盒6	6	6	9	13	8	0	16	14	10	17
墨盒7	12	12	14	10	12	6	0	7	15	22
墨盒8	11	11	10	3	5	5	14	0	8	15
墨盒9	15	15	14	7	9	9	11	4	0	7
墨盒10	20	20	11	16	15	14	8	15	19	0

图 7 附件 4 实例 3

基于方案二，通过调用 Gurobi，得到的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

表 11 问题四 Gurobi 结果

	实例 1	实例 2	实例 3	实例 4
运行结果/分钟	37	64	98	上界 264
运行时间/秒	0.0	0.0	0.0	1h

5.4.3 结果分析

表 12 问题四结果汇总

		实例 1	实例 2	实例 3	实例 4
遗传算法	运行结果/分钟	21	37	70	367
	运行时间/秒	0.477	0.529	0.545	1.433
Gurobi	运行结果/分钟	37	64	98	上界 264
	运行时间/秒	0.0	0.0	0.0	1h

5.5 问题五的求解

5.5.1 模型建立

问题五考虑用 2 台喷雾清洗机，完成所有任务，需要确定机器访问任务的顺序，使得所有任务均被完成。为此，针对问题五，本文尝试考虑更一般的M台机器的插槽墨盒动态组合方式问题，提出基于 M-TSP 的整数规划模型，并在M取 2 时对问题 5 求解。

目标函数：总切换时间尽可能少。

$$min \sum_{u_2 \in U} \sum_{u_1 \in U} \sum_{k \in K} \sum_{i \in N} T_{[u_1][u_2]} \cdot Z_{iku_1u_2}$$

约束 1-3：同问题一方案二。

约束 4：从起点出发，回到终点$n+1$点的机器数 2 改成机器数$\leq M$。

$$\sum_{j=N} Y_{0j} = \sum_{j=N} Y_{j,n+1}$$

$$\sum_{j=N} Y_{0j} \leq M$$

约束 5-9：同问题四约束 6-10。

综上，可得到问题五的模型规划，如下所示：

$$min \sum_{u_2 \in U} \sum_{u_1 \in U} \sum_{k \in K} \sum_{i \in N} T_{[u_1][u_2]} \cdot Z_{iku_1u_2}$$

$$s.t.\left\{\begin{array}{c}
\sum_{k \in K} X_{iks} = 1 \quad \forall i \in N, s \in S_i \\
\sum_{k \in K} X_{iku} \leq 1 \quad \forall i \in N, u \in U \\
\sum_{u \in U} X_{iku} \leq 1 \quad \forall i \in N, k \in K \\
\sum_{j=N} Y_{0j} = \sum_{j=N} Y_{j,n+1} \qquad \sum_{j=N} Y_{0j} \leq M \\
\sum_{\substack{j \neq i \\ j=N\cup\{0\}}} Y_{ji} = \sum_{\substack{j \neq i \\ j=N\cup\{n+1\}}} Y_{ij} \qquad \sum_{\substack{j \neq i \\ j=N\cup\{0\}}} Y_{ji} = 1 \\
V_j \geq V_i + 1 - n(1 - Y_{ij}) \quad \forall j \in N \cup \{n+1\}, \forall i \in N \cup \{0\}, 且 i \neq j \\
V_0 = 0 \\
\sum_{k_i=1}^{k} X_{ik_1S_{i,idx-1}} \geq X_{ikS_{i,idx}} \quad \forall k \in K, idx \in [2, |S_i|] \cap Z^+ \\
Z_{iku_1u_2} \geq X_{iku_1} + X_{i+1,k,u_2} - 1 \quad \forall\, u_2 \in U, u_1 \in U, k \in K, i \in N 且 i+1 \in N \\
X_{iku} \in \{0,1\}, \qquad Y_{ik} \in \{0,1\}
\end{array}\right.$$

5.5.2 模型求解

（1）遗传算法求解

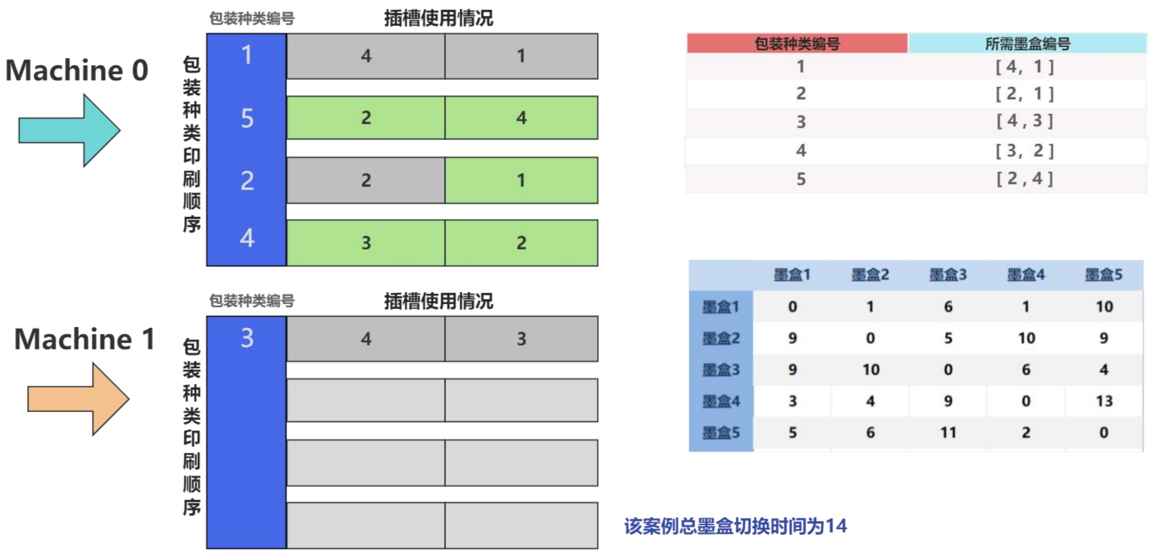
基于方案一，设计该算法，得到问题五的运行结果及运行时间如下表所示，其中运行时间保留至小数点后三位：

表 13 问题五遗传算法结果

	实例 1	实例 2	实例 3	实例 4
运行结果/分钟	16	158	237	285
运行时间/秒	0.716	0.996	1.472	2.071

（2）Gurobi 求解

基于方案二，以附件 5 中的实例 1 为例，下图展示了插槽使用情况及墨盒切换时间情况：



包装种类编号	所需墨盒编号
1	[4, 1]
2	[2, 1]
3	[4,3]
4	[3, 2]
5	[2,4]

	墨盒1	墨盒2	墨盒3	墨盒4	墨盒5
墨盒1	0	1	6	1	10
墨盒2	9	0	5	10	9
墨盒3	9	10	0	6	4
墨盒4	3	4	9	0	13
墨盒5	5	6	11	2	0

该案例总墨盒切换时间为14

图 8 附件 5 实例 1

六、模型的评价

6.1 优点

基于对本文问题的分析，构建了两种不同的整数规划模型，提供了本文问题丰富的建模方法。同时，采用多种模型求解方案：包含近似算法、启发式算法以及调用求解器求解的方法。本文借助了对 TSP 问题的分析方法，考虑了更一般的 M 台清洗机的墨盒切换问题，对本问题有了进一步的思考。

6.2 缺点

然而，本文也有一些不足，比如：考虑近似算法时，本文只给出了问题一和问题二的策略，且并未分析其近似比；另外，本文在调用 Gurobi 前并未利用精确求解算法对

模型求解难度进行简化，而是直接调用求解。未来如果对模型进一步完善，设计好的近似算法或精确求解算法，或许会对本问题的认识更加深化。

参考文献

[1]Ge, B., & Chan, W. L. (2017). "Optimal design of color ink-switching system for multi-color flexographic printing machine." Computers & Industrial Engineering, 111, 224-233.

[2]Elango, S., & Senthilkumar, M. (2017). "Performance Improvement of Flexographic Printing Machine by Quick Changeover Techniques." Procedia Engineering, 174, 457-464.

[3]Yung C. Shin, Shikui Chen, Ali Mehrabi. "An optimization model for printing ink usage in roll-to-roll printing process." Journal of Manufacturing Processes, Volume 24, 2016, Pages 215-223.

[4]张卫东，李喜丹，赵俊义，樊希元."柔性版印刷机自动调墨系统研究." 中国造纸学报, 2015(2), 42-45.

附录

本部分共有三部分组成，分别为贪婪策略、遗传算法以及调用 Gurobi 所需的全部代码：

Part1：基于贪婪策略的近似算法

问题一：

```
# 初值设置为第一件包装需要的墨盒

import time
total_switches = 0
current_config = []
def greedy_min_switch_operations(packages, slot_count):
    global total_switches
    global current_config
    n = len(packages)
    total_switches = 0
    current_config = [''] * slot_count
    min_switches = float('inf')
    # 用于记录每一件包装的墨盒切换次数
    switches=0

#   初始化墨盒==>第一包装需要的墨盒
    initial_config = packages[0]
    # 用于记录最新的墨盒队列（可以不满）
    current_config[:] = initial_config
    empty_ink = slot_count-len(initial_config)
    print("所有包装信息为:",packages)
    print("初始化墨盒为:",initial_config)
    print("剩余墨盒数:",empty_ink)



    # 从第二层开始
    for i in range(1,n):
        # print(i)
        # print(packages[i])
        # 计算出当前需要的墨盒有哪几个不存在在插槽中
        need_change = number_of_noexist_ink(packages[i])
        # 处理可以直接插入的情况
        if len(need_change) <= empty_ink:
            empty_ink = next_fill(need_change, empty_ink)
            print("目前剩余:", empty_ink)
            print("最新墨盒信息:", current_config)
            continue
        # 处理不能直接插入的情况：(1)还有空位，只是不够 （2）没有空位，需要选择切换
```

```
            switches = update_ink_cartridges(current_config, packages[i], packages[i+1:],empty_ink)
            print(f"目前处理到第{i + 1}个包装,墨盒切换次数为：{switches}")
            empty_ink=0;
            total_switches += switches
            print(f"第{i + 1}件包装交换的次数为：{switches}")
        print(f"总的交换次数为：{total_switches}")

        if total_switches < min_switches:
            min_switches = total_switches

        return min_switches


def next_fill(list,empty_ink):
        for ink in list:
            current_config.append(ink)
        empty_ink -= len(list)
        return empty_ink


#      计算出当前需要的墨盒有哪几个不存在在插槽中
def number_of_noexist_ink(list):
        no_exist_ink = []
        for ink in list:
            if ink not in current_config:
                no_exist_ink.append(ink)
        return no_exist_ink


def update_ink_cartridges(current_config, required_ink, packages,empty_ink):
        print("最初墨盒为：", current_config)
        # 找到所有需要但是不存在于插槽中的墨盒
        need_ink = number_of_noexist_ink(required_ink)

        # 还有空位 则先插空位
        if empty_ink>0:
            for i in range(empty_ink):
                current_config.append(need_ink[i])
        print("lallala 剩余空插槽数：",empty_ink)
        # 选择要切换的墨盒：1.当前要用的不能切换 2.切换频率低的
        # 对剩下的所有墨盒排序：从高到低
        # sorted_config = set_initial_config(packages, slot_count)
        sorted_config = ink_sort(packages)
        print("排序啦！！：",sorted_config)
```

```
    # 先对目前墨盒中的元素进行排序 从高到低
    new_config = []
    for ink in required_ink:
        if ink in current_config:
            new_config.append(ink)
    for ink in sorted_config:
        if ink in current_config and ink not in required_ink:
            new_config.append(ink)

    # 处理长度不够的情况
    if len(new_config)!=slot_count:
        for ink in current_config:
            if ink not in new_config:
                new_config.append(ink)
    print("新频率列表为:",new_config)

    # 计算需要替换的数量
    need_replace_count = len(need_ink) - empty_ink
    for i in range(need_replace_count):
        if new_config[len(new_config) - 1 - i] not in required_ink:
            new_config[len(new_config) - 1 - i] = need_ink[i+empty_ink]

    print("交换后的墨盒为：", new_config)
    current_config[:] = new_config
    return need_replace_count


"""
设置初始墨盒配置
:param packages: list of list of str, 各个包装的墨盒配置
:param slot_count: int, 插槽数量
:return: list of str, 初始墨盒配置
"""
def set_initial_config(packages, slot_count):
    sorted_colors = ink_sort(packages)
    initial_config = sorted_colors[:slot_count]
    if len(initial_config) < slot_count:
        initial_config += [''] * (slot_count - len(initial_config))
    return initial_config

# 颜色频率排序
def ink_sort(packages):
    from collections import Counter
    color_frequency = Counter(color for package in packages for color in package)
```

```
        # 给颜色安装频率高低进行排序
        sorted_colors = [color for color, _ in color_frequency.most_common()]
        return sorted_colors

# Example usage
# 1_5_10_2
# packages = [
#         ['7', '5'],    # Example package ink configurations
#         ['2', '5'],
#         ['4', '2'],
#         ['8'],
#         ['7','3']
# ]
#
# slot_count = 2

# Example usage
# 2_7_10_3
packages = [
        ['2', '7'],    # Example package ink configurations
        ['3', '8','6'],
        ['7', '9'],
        ['6','9'],
        ['4'],
            ['2','6'],
            ['7','3','1']
]
slot_count = 3

# Example usage
# 3_10_50_15
# packages = [
#         ['11', '7'],    # Example package ink configurations
#         ['31', '6','40','21','10','17','35','42','23','24','33'],
#         ['11', '16','30','27','1','10','19','8','29','14','23','43','15','4'],
#         ['23','19','30','48','12','10','44','31'],
#         ['4','38','41','10','26','6'],
#         ['5', '42','25'],
#         ['25', '27','13','31','30','22','43','28','16','48','7','47','17'],
#         ['4','40','16','36','33','19','18','23','26','6','15','27','49','7'],
#         ['26', '8','20','41','6','5','18','3','4','39','47','12','1','34','28'],
#         ['2', '24','38','6','39','9','4']
# ]
# slot_count = 15
# Example usage
# 4_20_50_10
# packages = [
#         ['30', '25', '11', '28'],
#         ['36', '2'],
#         ['41', '1', '28'],
#         ['13', '22', '12', '25'],
#         ['7', '37', '31', '46', '40'],
#         ['38'],
#         ['46', '14'],
#         ['34', '15', '27', '44', '35', '31', '3', '11', '21', '28'],
#         ['21', '26'],
#         ['18', '2', '4', '20', '42'],
#         ['18', '10', '4'],
#         ['21', '8', '23', '5', '4', '32', '30', '40', '10'],
#         ['25', '16', '2', '48', '20', '21', '39', '23', '3', '34'],
#         ['7', '45', '41'],
#         ['17', '18'],
#         ['45', '7', '3', '28', '10', '9'],
#         ['33', '43', '36', '49', '10', '40'],
#         ['22', '6', '17', '44', '33'],
#         ['15', '13', '23', '12', '6', '22', '44', '4'],
#         ['13', '35', '28', '48', '5', '20']
# ]
# slot_count = 10


# Example usage
# 5_30_60_10
# packages = [
#         ['56', '20', '26', '44', '14', '9', '33', '5', '11', '55'],
#         ['56', '49', '4'],
#         ['3', '38', '10'],
#         ['54', '8', '4', '19', '35', '36', '56', '46', '2'],
#         ['24', '33', '34'],
#         ['11', '3', '56', '48', '12', '40', '10'],
#         ['56', '36'],
#         ['51', '18', '5', '1', '14', '12', '40', '37', '30', '20'],
#         ['11', '10', '32'],
#         ['26', '43'],
#         ['10', '38', '56', '44', '57', '16', '52', '22', '6', '50'],
#         ['24', '52', '10'],
#         ['10', '59', '28', '46'],
#         ['37', '1', '30', '49', '3', '34', '15', '20', '10'],
#         ['19', '15', '22', '23', '36', '30'],
#         ['18', '58', '38', '7', '14', '28', '33', '41'],
#         ['14', '29', '18', '11', '8', '41'],
#         ['15', '46', '56', '50', '10', '3', '48', '31'],
#         ['42', '45', '29', '26', '8', '46'],
#         ['40', '57', '10', '51', '54', '20', '24', '48', '37'],
#         ['35', '20'],
#         ['56', '16', '23', '25'],
#         ['3', '40', '9', '12', '10', '49', '14', '43'],
#         ['4', '32', '46', '25', '24', '22', '33', '13'],
#         ['21', '40', '23', '3'],
#         ['30', '3', '35', '16', '41', '54', '48', '29', '17', '50'],
#         ['59', '34'],
#         ['38', '3', '36', '47', '51', '49', '10', '6'],
#         ['42', '1', '44', '13', '41', '59'],
#         ['33', '13', '9', '39', '30', '16', '24']
# ]
# slot_count = 10

start_time = time.time()
result = greedy_min_switch_operations(packages, slot_count)
end_time = time.time()

print("Minimum switch operations (Greedy):", result)
print(f"Execution time: {end_time - start_time:.4f} seconds")
```

问题二：

```
import numpy as np
import pandas as pd
from copy import deepcopy
import time
current_config = []
color_list=[]

def ink_sort_frequency(packages):
    from collections import Counter
    #  统计每种颜色的频次
    color_frequency = Counter(color for package in packages for color in package)
    #  根据频次对颜色进行排序，并返回颜色及其频次的元组
    sorted_colors_with_frequency = color_frequency.most_common()
    # 将频次转换为百分比
    total_count = sum(color_frequency.values())
    sorted_colors_with_frequency = [(color, count / total_count) for color, count in
sorted_colors_with_frequency]
    return sorted_colors_with_frequency

def ink_sort_color(packages):
    from collections import Counter
    # 统计每种颜色的频次
    color_frequency = Counter(color for package in packages for color in package)
    # 根据颜色进行排序，并返回颜色及其频次的元组
    sorted_colors = sorted(color_frequency.keys())
    # 计算每种颜色的百分比
    total_count = sum(color_frequency.values())
    sorted_colors_with_frequency = [(color, color_frequency[color] / total_count) for color in
sorted_colors]
    return sorted_colors_with_frequency


def set_initial_config(packages, slot_count):
    sorted_colors_with_frequency = ink_sort_frequency(packages)
    # 提取颜色，按频次排序
    sorted_colors = [color for color, _ in sorted_colors_with_frequency]
    initial_config = sorted_colors[:slot_count]
    if len(initial_config) < slot_count:
        initial_config += [''] * (slot_count - len(initial_config))
    return initial_config


def min_switch_time(T, packages, slot_count):
    totol_time = 0
    global current_config
    global color_list
    current_config = [''] * slot_count
    color_list=[]
    P = len(packages)

    color_set = set(color for package in packages for color in package)
    color_set_origin = sorted(color_set)

    # 根据切换时间与频率，进行排序（时间/使用频率）值越大 代价越高，需要切换
    # 首先找到频率的排名
    frequency_sort = ink_sort_color(packages)
    print("墨盒频率排序:",frequency_sort)
    #  计算每个包装切换到任意一个包装的代价==>时间花费矩阵进行预处理


#    获取当前需要用到的墨盒时间列表与颜色坐标
    new_T,color_list = data_Pretreatment(T, packages)
    print("lalalal:",color_list)
    for i in range(len(new_T)):#  行
        for j in range(len(new_T[i])):#  列
            if j!=i:
                probability = frequency_sort[j][1]
                new_T[i][j] = str(float(T[i][j]) / probability)
    #    求得当前所有包装的代价
    print("切换代价计算:",new_T)

    #  第一件包装所需墨盒列表,设为初值
    initial_config = packages[0]
    current_config = initial_config
    print("第一包装所需：",initial_config)

    #  记录空插槽的值
    empty_ink= slot_count - len(current_config)

    #  当前可用的颜色坐标



    for i in range(1,P):
        # 1.计算要切换的数量
        need_change = number_of_noexist_ink(packages[i],current_config)
        #  重新更新 new_T
        #       1.获得新频次排名，包含本次
        new_frequency_sort = ink_sort_color(packages[i:])
        print("最新频率排行:",new_frequency_sort)

        # #  初始化更新后的频率列表
        updated_frequencies = [list(frequency) for frequency in frequency_sort]

        for h in range(len(frequency_sort)):
            color = frequency_sort[h][0]
            color_pro = frequency_sort[h]
            for j in range(len(new_frequency_sort)):
                new_color = new_frequency_sort[j][0]
                new_color_pro = new_frequency_sort[j]
                if color == new_color:
                        updated_frequencies[h] = list(new_color_pro)
                        break
                else:
                    updated_frequencies[h][0] = color
                    updated_frequencies[h][1] = float('1000')
        print("更新后的频率：", updated_frequencies)

        # 更新代价表
        for k in range(len(new_T)):   # 行
            for j in range(len(new_T[k])):   # 列
                if j != k:
                    probability = updated_frequencies[j][1]
                    new_T[k][j] = str(float(T[k][j]) / probability)
        print("最新 new_T:",new_T)

        # 查看是否还有空插槽
        if len(need_change) <= empty_ink:
            empty_ink = next_fill(need_change, empty_ink)
            print("目前剩余:", empty_ink)
            print("最新墨盒信息:", current_config)
            continue


        # 找到可以替换的列表
        change_list = get_change_list(current_config,packages[i])
        print("可交换列表:",change_list)
        print("需要交换的墨盒编号:",need_change)
        switch_time = change_worth(change_list, need_change, new_T,T)
        print("该次切换时间为:", switch_time)
        totol_time = totol_time + switch_time

        print("总的最小切换时间:", totol_time)
        print("当前墨盒信息为:",current_config)


def empty_check(need_list,empty_len):
    if(empty_len>len(need_list)):
        empty_len=empty_len-len(need_list)
        print("剩余空插槽:",empty_len)
        return 1
    else:
        return 0

def next_fill(list,empty_ink):
    for ink in list:
        current_config.append(ink)
    empty_ink -= len(list)
    return empty_ink


def find_location(ink):
    for i, color in enumerate(color_list):
        if int(color) == ink:
            return i
    return -1   # 如果没有找到对应的墨盒颜色，返回-1

def find_max_switch(T, change_list, need_list):
    max_switch = float('-inf')
    max_ink = None
    max_bin = None
    for ink in change_list:
        x_lab = find_location(int(ink))
        for bin in need_list:
            y_lab = find_location(int(bin))
            switch_time = float(T[x_lab][y_lab])
            if switch_time > max_switch:
                max_switch = switch_time
                max_ink = ink
                max_bin = bin

    return max_ink, max_bin


def update_lists(change_list, need_list, used_ink, used_bin):
    change_list.remove(used_ink)
    need_list.remove(used_bin)
    return change_list, need_list

def change_worth(change_list, need_list, new_T,T):
    total_switch_time = 0
    while change_list and need_list:
        # 找切换点 用 new_T
        min_ink, min_bin = find_max_switch(new_T, change_list, need_list)
        # 切换时间，看 T
        min_switch = float(T[int(min_ink)-1][int(min_bin)-1])
        print(f"选择了墨盒 {min_ink} 切换到墨盒 {min_bin}，切换时间为 {min_switch}")
        for i in range(len(current_config)):
            if current_config[i] == min_ink:
                    current_config[i] = min_bin

            total_switch_time += min_switch
            change_list, need_list = update_lists(change_list, need_list, min_ink, min_bin)
    return total_switch_time


def get_change_list(old_config,need_config):
    change_list = []
    for ink in old_config:
        if ink not in need_config:
            change_list.append(ink)
    return    change_list

def number_of_noexist_ink(list,current_config):
    no_exist_ink = []
    for ink in list:
        if ink not in current_config:
            no_exist_ink.append(ink)
    return no_exist_ink


# 数据预处理
def data_Pretreatment(T,packages):
    # 找到所有使用到的墨盒颜色
    color_set = set(color for package in packages for color in package)
    color_set_origin = sorted(color_set)
    # 将集合转换为列表，对列表中的每个元素执行减法操作，然后再转换回集合
    # 对 str 进行减法，需要类型转换
    color_set = set(int(x) - 1 for x in color_set_origin)
    color_list = [str(x) for x in color_set]
    color_list = [int(x) for x in color_list]
    # { }  是集合    [",")是列表
    # 将 color_list 转换为整数集合
    print("目前用到的颜色:",color_list)
    need_T = remove_rows_and_columns(T,color_list,color_list)
    print(need_T)
    print("长度",len(need_T))
    return need_T,color_set_origin

def remove_rows_and_columns(matrix, row_indices, col_indices):
    # 删除指定的行
    matrix = [row for i, row in enumerate(matrix) if i in row_indices]
    # 删除指定的列
```

```
        matrix = [[elem for j, elem in enumerate(row) if j in col_indices] for row in matrix]
        return matrix


# 示例数据
# 读取 excel 表中数据
spend_time = pd.read_excel('附件 2/Ins1_5_10_3.xlsx',sheet_name="墨盒切换时间")
pacakages_info = pd.read_excel('附件 2/Ins1_5_10_3.xlsx',sheet_name="包装种类及其所需墨盒")
#
# # 将 DataFrame 转换为列表数组，并进行切片和转换操作
data_list = spend_time.values.tolist()
T = [[str(item) for item in row[1:]] for row in data_list]

# 例 1：5_10_3
packages = [
        ['3', '6', '4'],
        ['6', '5'],
        ['1', '5'],
        ['9'],
        ['2', '1'],
]

# 例 2：2_7_10_2
# spend_time = pd.read_excel('附件 2/Ins2_7_10_2.xlsx',sheet_name="墨盒切换时间")
# # 将 DataFrame 转换为列表数组，并进行切片和转换操作
# data_list = spend_time.values.tolist()
# T = [[str(item) for item in row[1:]] for row in data_list]
#
# packages = [
#         [2],
#         [8],
#         [3, 5],
#         [4],
#         [6, 5],
#         [3, 1],
#         [1, 6]
# ]


# 例 3：3_10_30_10
# spend_time = pd.read_excel('附件 2/Ins3_10_30_10.xlsx',sheet_name="墨盒切换时间")
# # 将 DataFrame 转换为列表数组，并进行切片和转换操作
# data_list = spend_time.values.tolist()
# T = [[str(item) for item in row[1:]] for row in data_list]
```

```
#
# packages = [
#         [3, 19, 22, 12, 10, 11, 8, 21, 16],
#         [20, 2, 14, 5, 11, 19, 12, 18, 23, 26],
#         [24, 11, 14, 8, 1, 20, 29, 25, 12],
#         [21, 27],
#         [1, 4, 22, 10, 17, 9, 27, 8, 29, 7],
#         [29, 20],
#         [20, 27, 4],
#         [8, 3, 1, 7],
#         [14, 23, 18, 24, 4, 22, 27, 9],
#         [3, 18, 12, 20, 16]
# ]


# 例 4：4_20_40_10
# spend_time = pd.read_excel('附件 2/Ins4_20_40_10.xlsx',sheet_name="墨盒切换时间")
# # 将 DataFrame 转换为列表数组，并进行切片和转换操作
# data_list = spend_time.values.tolist()
# T = [[str(item) for item in row[1:]] for row in data_list]
#
# packages = [
#         [25, 6, 20, 29, 28, 2, 24, 9, 15, 4],
#         [19, 9, 4, 36, 20, 18, 31, 39, 26],
#         [30, 8, 7],
#         [34, 36, 21, 9, 14],
#         [17, 1, 27, 5, 19],
#         [29, 25, 14, 12, 28, 10, 27, 34, 23, 17],
#         [36, 32, 23, 1, 26, 9, 2, 12, 24],
#         [22, 11, 1, 21],
#         [19, 26, 10, 20, 11, 13, 12, 5, 34],
#         [1, 20, 35, 30],
#         [30, 18, 19, 15, 5, 16, 33, 23],
#         [6, 32, 18, 19],
#         [38, 10, 36, 19, 3, 24, 21, 26, 6],
#         [11, 31, 21, 4, 39, 1, 26, 20, 33, 6],
#         [12, 33, 23],
#         [29, 33, 36, 30, 37, 9],
#         [26, 8, 20, 13, 32, 29, 27, 36, 39, 24],
#         [8, 22, 1, 26, 11, 31, 19],
#         [34, 24],
#         [33, 20, 2]
# ]
# 例 5：5_30_60_10
# spend_time = pd.read_excel('附件 2/Ins5_30_60_10.xlsx',sheet_name="墨盒切换时间")
# # 将 DataFrame 转换为列表数组，并进行切片和转换操作
# data_list = spend_time.values.tolist()
# T = [[str(item) for item in row[1:]] for row in data_list]

# packages = [
#         ['46', '47', '44', '4', '49'],
#         ['32', '56', '28', '18', '52', '7'],
#         ['15', '55', '16', '28', '22', '23', '46', '52', '53'],
#         ['8', '51', '19', '59', '24'],
#         ['51', '30', '18', '44', '41', '10'],
#         ['26', '5', '19'],
#         ['5', '20', '30', '7', '15', '4', '14'],
#         ['15', '33', '40', '23', '17', '55', '25', '28'],
#         ['14', '2', '8', '25', '20', '12', '3'],
#         ['7', '55', '21', '53', '13', '1'],
#         ['11', '22', '25', '17', '33', '52'],
#         ['4', '36', '29', '56', '17', '24', '11', '39'],
#         ['4', '10', '44', '5', '17', '51', '32', '21', '3', '24'],
#         ['56', '36', '24', '32', '33', '25', '44', '53'],
#         ['36', '59', '46'],
#         ['33', '6'],
#         ['45', '41', '7', '46', '40', '17', '9', '56'],
#         ['31', '38', '33'],
#         ['56', '34', '35', '26', '43', '21', '37', '2', '3'],
#         ['40', '59', '23', '14', '43'],
#         ['33', '25', '17'],
#         ['37', '35', '11', '24', '45', '26', '27', '7', '53'],
#         ['9', '19', '8', '22'],
#         ['24', '9', '36', '23', '48'],
#         ['53', '21', '2', '11', '12', '49', '28', '14', '27', '13'],
#         ['29', '32', '36', '46', '14', '17', '11'],
#         ['28', '54'],
#         ['50', '33', '20'],
#         ['40', '24', '59', '1', '54', '15', '52', '53', '34', '30'],
#         ['55', '43', '28']
# ]


slot_count = 3
start_time = time.time()
result = min_switch_time(T, packages, slot_count)
end_time = time.time()
```

```
print("Minimum switch time (Greedy):", result)
print(f"Execution time: {end_time - start_time:.4f} seconds")
```

Part2：问题一到问题五基于整数规划的遗传算法

文件 1：

```
import numpy as np
import pandas as pd
from config import Config
from copy import deepcopy
from jmetal.core.problem import Problem, S
from jmetal.core.solution import CompositeSolution, FloatSolution, IntegerSolution, PermutationSolution
from fitness import Fitness
import random
from task import Task, SubTask
from station import Station

random.seed(0)


class AlgoProblem(Problem):
    def number_of_variables(self) -> int:
        return len(self.sub_task_id_list)

    def number_of_objectives(self) -> int:
        return 1

    def number_of_constraints(self) -> int:
        return 1

    def name(self) -> str:
        pass

    def __init__(self):
        super().__init__()
        # course_id + type
        """包装 id 列表"""
        self.task_id_list = []
        self.task_list = []
        self.sub_task_id_list = []
        self.sub_task_dict = {}
        self.task_dict = {}
        """"插槽 id 列表"""
        self.station_id_list = []
        self.station_dict = {}

        """墨盒切换时间"""
        self.distance_matrix = {}
        self.number_of_objectives = 1
        self.number_of_constraints = 1
        self.number_of_variables = None

        self.build()
        self.fitness = Fitness()
        self.obj_directions = [self.MINIMIZE]

    def get_name(self) -> str:
        pass

    def evaluate(self, solution: S):
        self.fitness.evaluate_fitness(solution)

    def create_solution(self) -> CompositeSolution:
        solution_list = []
        #  给每个包装对应的墨盒分配插槽
        for task_id in self.task_id_list:
            task_obj = self.task_dict[task_id]
            per_solution_1 = PermutationSolution(len(self.station_id_list), self.number_of_objectives,
                                                 self.number_of_constraints)

            per_solution_1.variables          =          random.sample(range(len(self.station_id_list)),
k=len(self.station_id_list))
            solution_list.append(per_solution_1)

        #  包装的执行任务顺序
        float_solution = FloatSolution([0.0 for _ in range(len(self.task_id_list))],
                                       [1.0 for _ in range(len(self.task_id_list))],
                                       self.number_of_objectives,
                                       self.number_of_constraints)
        float_solution.variables = [random.uniform(0.0, 1.0) for _ in self.task_id_list]
        # float_solution1 = FloatSolution([0.0 for _ in range(len(self.sub_task_dict))],
        #                                [1.0 for _ in range(len(self.sub_task_dict))],
        #                                self.number_of_objectives,
        #                                self.number_of_constraints)
        # float_solution1.variables = [random.uniform(0.0, 1.0) for _ in self.sub_task_dict]
        solution_list.append(float_solution)
        # solution_list.append(float_solution1)
        return CompositeSolution(solution_list)

    def build(self):
        df = pd.read_excel(Config.data_file_path, sheet_name=None)
        real_sub_id_list = []
        for index, row in df["包装-墨盒-插槽"].iterrows():
            if not np.isnan(row["包装种类编号"]):
                task_id = int(row["包装种类编号"])
                self.task_id_list.append(task_id)
                task_obj = Task()
                task_obj.id = task_id
                self.task_list.append(task_obj)
                self.task_dict[task_id] = task_obj
            if not np.isnan(row["插槽序号"]):
                s_id = int(row["插槽序号"])
                self.station_id_list.append(s_id)
                s_obj = Station()
                s_obj.id = s_id
                self.station_dict[s_id] = s_obj
            if not np.isnan(row["墨盒编号"]):
                real_sub_id_list.append(int(row["墨盒编号"]))

        for index, row in df["包装种类及其所需墨盒"].iterrows():
            task_id = int(row["包装种类编号"])
            task_obj = self.task_dict[task_id]
            sub_id_str = row["所需墨盒编号"]
            sub_id_str = sub_id_str[1: len(sub_id_str) - 1]
            sub_id_str_list = sub_id_str.split(", ")
            sub_id_list = [int(s) for s in sub_id_str_list]
            for sub_id in sub_id_list:
                sub_task = SubTask()
                sub_task.id = str(task_id) + "-" + str(sub_id)
                sub_task.real_id = sub_id
                sub_task.task_id = task_id
                self.sub_task_id_list.append(sub_task.id)
                self.sub_task_dict[sub_task.id] = sub_task
                task_obj.sub_tasks.append(sub_task)

        if Config.solver_version == 1:
            for sub_id1 in real_sub_id_list:
                for sub_id2 in real_sub_id_list:
                    if sub_id1 != sub_id2:
                        self.distance_matrix[sub_id1, sub_id2] = 1
                    else:
                        self.distance_matrix[sub_id1, sub_id2] = 0
        else:
            df_1 = df["墨盒切换时间"]
            matrix = df_1.values.tolist()
            for index, values in enumerate(matrix):
                for index2, value in enumerate(values):
                    sub_id = real_sub_id_list[index]
                    sub_id2 = real_sub_id_list[index2]
                    self.distance_matrix[sub_id, sub_id2] = value
```

文件 2：

```
import os
class Config(object):
    """求解版本  1-问题 1 2-问题 2 3-问题 3 4-问题 4 5-问题 5"""
    solver_version = 1

    """ga  相关参数"""
    population_size = 100
    offspring_population_size = 100
    mutation_probability = 0.2
    crossover_probability = 0.8
    max_evaluations = 2000
    mento_caro_iter_num = 10
    """文件路径"""
    root_folder_path = os.path.dirname(os.path.abspath(__file__))
    data_folder_path = os.path.join(root_folder_path, "data")
    """工件文件"""
    data_file_path = os.path.join(data_folder_path, "Ins5_30_60_10.xlsx")
```

文件 3：

```
import copy
import random
from typing import List

from jmetal.core.operator import Crossover
from jmetal.core.solution import Solution, FloatSolution, BinarySolution, PermutationSolution, \
    CompositeSolution, IntegerSolution
from jmetal.util.ckecking import Check

"""
```

```
.. module:: crossover
   :platform: Unix, Windows
   :synopsis: Module implementing crossover operators.

.. moduleauthor:: Antonio J. Nebro <antonio@lcc.uma.es>, Antonio Benítez-Hidalgo <antonio.b@uma.es>
"""


class NullCrossover(Crossover[Solution, Solution]):
    def __init__(self):
        super(NullCrossover, self).__init__(probability=0.0)

    def execute(self, parents: List[Solution]) -> List[Solution]:
        if len(parents) != 2:
            raise Exception('The number of parents is not two: {}'.format(len(parents)))

        return parents

    def get_number_of_parents(self) -> int:
        return 2

    def get_number_of_children(self) -> int:
        return 2

    def get_name(self):
        return 'Null crossover'


class PMXCrossover(Crossover[PermutationSolution, PermutationSolution]):
    def __init__(self, probability: float):
        super(PMXCrossover, self).__init__(probability=probability)

    def execute(self, parents: List[PermutationSolution]) -> List[PermutationSolution]:
        if len(parents) != 2:
            raise Exception('The number of parents is not two: {}'.format(len(parents)))

        offspring = [copy.deepcopy(parents[0]), copy.deepcopy(parents[1])]
        permutation_length = offspring[0].number_of_variables

        rand = random.random()
        if rand <= self.probability:
            cross_points = sorted([random.randint(0, permutation_length) for _ in range(2)])

            def _repeated(element, collection):
                c = 0
                for e in collection:
                    if e == element:
                        c += 1
                return c > 1

            def _swap(data_a, data_b, cross_points):
                c1, c2 = cross_points
                new_a = data_a[:c1] + data_b[c1:c2] + data_a[c2:]
                new_b = data_b[:c1] + data_a[c1:c2] + data_b[c2:]
                return new_a, new_b

            def _map(swapped, cross_points):
                n = len(swapped[0])
                c1, c2 = cross_points
                s1, s2 = swapped
                map_ = s1[c1:c2], s2[c1:c2]
                for i_chromosome in range(n):
                    if not c1 < i_chromosome < c2:
                        for i_son in range(2):
                            while _repeated(swapped[i_son][i_chromosome], swapped[i_son]):
                                map_index = map_[i_son].index(swapped[i_son][i_chromosome])
                                swapped[i_son][i_chromosome] = map_[1 - i_son][map_index]
                return s1, s2

            swapped = _swap(parents[0].variables, parents[1].variables, cross_points)
            mapped = _map(swapped, cross_points)

            offspring[0].variables, offspring[1].variables = mapped

        return offspring

    def get_number_of_parents(self) -> int:
        return 2

    def get_number_of_children(self) -> int:
        return 2

    def get_name(self):
        return 'Partially Matched crossover'


class CXCrossover(Crossover[PermutationSolution, PermutationSolution]):
    def __init__(self, probability: float):
        super(CXCrossover, self).__init__(probability=probability)

    def execute(self, parents: List[PermutationSolution]) -> List[PermutationSolution]:
        if len(parents) != 2:
            raise Exception('The number of parents is not two: {}'.format(len(parents)))

        offspring = [copy.deepcopy(parents[1]), copy.deepcopy(parents[0])]
        rand = random.random()

        if rand <= self.probability:
            for i in range(parents[0].number_of_variables):
                idx = random.randint(0, len(parents[0].variables[i]) - 1)
                curr_idx = idx
                cycle = []

                while True:
                    cycle.append(curr_idx)
                    curr_idx = parents[0].variables[i].index(parents[1].variables[i][curr_idx])

                    if curr_idx == idx:
                        break

                for j in range(len(parents[0].variables[i])):
                    if j in cycle:
                        offspring[0].variables[i][j] = parents[0].variables[i][j]
                        offspring[1].variables[i][j] = parents[0].variables[i][j]

        return offspring

    def get_number_of_parents(self) -> int:
        return 2

    def get_number_of_children(self) -> int:
        return 2

    def get_name(self):
        return 'Cycle crossover'


class SBXCrossover(Crossover[FloatSolution, FloatSolution]):
    __EPS = 1.0e-14

    def __init__(self, probability: float, distribution_index: float = 20.0):
        super(SBXCrossover, self).__init__(probability=probability)
        self.distribution_index = distribution_index
        if distribution_index < 0:
            raise Exception("The distribution index is negative: " + str(distribution_index))

    def execute(self, parents: List[FloatSolution]) -> List[FloatSolution]:
        Check.that(issubclass(type(parents[0]), FloatSolution), "Solution type invalid: " +
str(type(parents[0])))
        Check.that(issubclass(type(parents[1]), FloatSolution), "Solution type invalid")
        Check.that(len(parents) == 2, 'The number of parents is not two: {}'.format(len(parents)))

        offspring = [copy.deepcopy(parents[0]), copy.deepcopy(parents[1])]
        rand = random.random()

        if rand <= self.probability:
            for i in range(parents[0].number_of_variables):
                value_x1, value_x2 = parents[0].variables[i], parents[1].variables[i]

                if random.random() <= 0.5:
                    if abs(value_x1 - value_x2) > self.__EPS:
                        if value_x1 < value_x2:
                            y1, y2 = value_x1, value_x2
                        else:
                            y1, y2 = value_x2, value_x1

                        lower_bound, upper_bound = parents[0].lower_bound[i],
parents[1].upper_bound[i]

                        beta = 1.0 + (2.0 * (y1 - lower_bound) / (y2 - y1))
                        alpha = 2.0 - pow(beta, -(self.distribution_index + 1.0))

                        rand = random.random()
                        if rand <= (1.0 / alpha):
                            betaq = pow(rand * alpha, (1.0 / (self.distribution_index + 1.0)))
                        else:
                            betaq = pow(1.0 / (2.0 - rand * alpha), 1.0 / (self.distribution_index +
1.0))

                        c1 = 0.5 * (y1 + y2 - betaq * (y2 - y1))
                        beta = 1.0 + (2.0 * (upper_bound - y2) / (y2 - y1))
                        alpha = 2.0 - pow(beta, -(self.distribution_index + 1.0))

                        if rand <= (1.0 / alpha):
                            betaq = pow((rand * alpha), (1.0 / (self.distribution_index + 1.0)))
                        else:
                            betaq = pow(1.0 / (2.0 - rand * alpha), 1.0 / (self.distribution_index +
1.0))

                        c2 = 0.5 * (y1 + y2 + betaq * (y2 - y1))

                        if c1 < lower_bound:
                            c1 = lower_bound
                        if c2 < lower_bound:
                            c2 = lower_bound
                        if c1 > upper_bound:
                            c1 = upper_bound
                        if c2 > upper_bound:
                            c2 = upper_bound

                        if random.random() <= 0.5:
                            offspring[0].variables[i] = c2
                            offspring[1].variables[i] = c1
                        else:
                            offspring[0].variables[i] = c1
                            offspring[1].variables[i] = c2
                    else:
                        offspring[0].variables[i] = value_x1
                        offspring[1].variables[i] = value_x2
                else:
                    offspring[0].variables[i] = value_x1
                    offspring[1].variables[i] = value_x2
        return offspring

    def get_number_of_parents(self) -> int:
        return 2

    def get_number_of_children(self) -> int:
        return 2

    def get_name(self) -> str:
        return 'SBX crossover'


class IntegerSBXCrossover(Crossover[IntegerSolution, IntegerSolution]):
    __EPS = 1.0e-14

    def __init__(self, probability: float, distribution_index: float = 20.0):
        super(IntegerSBXCrossover, self).__init__(probability=probability)
        self.distribution_index = distribution_index
    def execute(self, parents: List[IntegerSolution]) -> List[IntegerSolution]:
        Check.that(issubclass(type(parents[0]), IntegerSolution), "Solution type invalid")
        Check.that(issubclass(type(parents[1]), IntegerSolution), "Solution type invalid")
        Check.that(len(parents) == 2, 'The number of parents is not two: {}'.format(len(parents)))

        offspring = [copy.deepcopy(parents[0]), copy.deepcopy(parents[1])]
        rand = random.random()

        if rand <= self.probability:
            for i in range(parents[0].number_of_variables):
                value_x1, value_x2 = parents[0].variables[i], parents[1].variables[i]

                if random.random() <= 0.5:
                    if abs(value_x1 - value_x2) > self.__EPS:
                        if value_x1 < value_x2:
                            y1, y2 = value_x1, value_x2
                        else:
                            y1, y2 = value_x2, value_x1

                        lower_bound, upper_bound = parents[0].lower_bound[i],
parents[1].upper_bound[i]

                        beta = 1.0 + (2.0 * (y1 - lower_bound) / (y2 - y1))
                        alpha = 2.0 - pow(beta, -(self.distribution_index + 1.0))

                        rand = random.random()
                        if rand <= (1.0 / alpha):
                            betaq = pow(rand * alpha, (1.0 / (self.distribution_index + 1.0)))
                        else:
                            betaq = pow(1.0 / max(0.001, (2.0 - rand * alpha)), 1.0 /
(self.distribution_index + 1.0))

                        c1 = 0.5 * (y1 + y2 - betaq * (y2 - y1))
                        beta = 1.0 + (2.0 * (upper_bound - y2) / (y2 - y1))
                        alpha = 2.0 - pow(beta, -(self.distribution_index + 1.0))

                        if rand <= (1.0 / alpha):
                            betaq = pow((rand * alpha), (1.0 / (self.distribution_index + 1.0)))
                        else:
                            betaq = pow(1.0 / max(0.001, (2.0 - rand * alpha)), 1.0 /
(self.distribution_index + 1.0))

                        c2 = 0.5 * (y1 + y2 + betaq * (y2 - y1))
                        if c1 < lower_bound:
                            c1 = lower_bound
                        if c2 < lower_bound:
                            c2 = lower_bound
                        if c1 > upper_bound:
                            c1 = upper_bound
                        if c2 > upper_bound:
                            c2 = upper_bound

                        if random.random() <= 0.5:
                            offspring[0].variables[i] = int(c2)
                            offspring[1].variables[i] = int(c1)
                        else:
                            offspring[0].variables[i] = int(c1)
                            offspring[1].variables[i] = int(c2)
                    else:
                        offspring[0].variables[i] = value_x1
                        offspring[1].variables[i] = value_x2
                else:
                    offspring[0].variables[i] = value_x1
                    offspring[1].variables[i] = value_x2
        return offspring

    def get_number_of_parents(self) -> int:
        return 2

    def get_number_of_children(self) -> int:
        return 2

    def get_name(self) -> str:
        return 'Integer SBX crossover'


class SPXCrossover(Crossover[BinarySolution, BinarySolution]):

    def __init__(self, probability: float):
        super(SPXCrossover, self).__init__(probability=probability)

    def execute(self, parents: List[BinarySolution]) -> List[BinarySolution]:
        Check.that(type(parents[0]) is BinarySolution, "Solution type invalid")
        Check.that(type(parents[1]) is BinarySolution, "Solution type invalid")
        Check.that(len(parents) == 2, 'The number of parents is not two: {}'.format(len(parents)))
        offspring = [copy.deepcopy(parents[0]), copy.deepcopy(parents[1])]
        rand = random.random()

        if rand <= self.probability:
            # 1. Get the total number of bits
            total_number_of_bits = parents[0].get_total_number_of_bits()

            # 2. Calculate the point to make the crossover
            crossover_point = random.randrange(0, total_number_of_bits)

            # 3. Compute the variable containing the crossover bit
            variable_to_cut = 0
            bits_count = len(parents[1].variables[variable_to_cut])
            while bits_count < (crossover_point + 1):
                variable_to_cut += 1
                bits_count += len(parents[1].variables[variable_to_cut])

            # 4. Compute the bit into the selected variable
            diff = bits_count - crossover_point
            crossover_point_in_variable = len(parents[1].variables[variable_to_cut]) - diff

            # 5. Apply the crossover to the variable
            bitset1 = copy.copy(parents[0].variables[variable_to_cut])
            bitset2 = copy.copy(parents[1].variables[variable_to_cut])

            for i in range(crossover_point_in_variable, len(bitset1)):
                swap = bitset1[i]
                bitset1[i] = bitset2[i]
                bitset2[i] = swap

            offspring[0].variables[variable_to_cut] = bitset1
            offspring[1].variables[variable_to_cut] = bitset2

            # 6. Apply the crossover to the other variables
            for i in range(variable_to_cut + 1, parents[0].number_of_variables):
                offspring[0].variables[i] = copy.deepcopy(parents[1].variables[i])
                offspring[1].variables[i] = copy.deepcopy(parents[0].variables[i])

        return offspring

    def get_number_of_parents(self) -> int:
        return 2

    def get_number_of_children(self) -> int:
        return 2

    def get_name(self) -> str:
        return 'Single point crossover'

class DifferentialEvolutionCrossover(Crossover[FloatSolution, FloatSolution]):
    """ This operator receives two parameters: the current individual and an array of three parent
individuals. The
    best and rand variants depends on the third parent, according whether it represents the current of
the "best"
    individual or a random_search one. The implementation of both variants are the same, due to that
the parent selection is
    external to the crossover operator.
    """

    def __init__(self, CR: float, F: float, K: float = 0.5):
        super(DifferentialEvolutionCrossover, self).__init__(probability=1.0)
        self.CR = CR
        self.F = F
        self.K = K

        self.current_individual: FloatSolution = None

    def execute(self, parents: List[FloatSolution]) -> List[FloatSolution]:
        """ Execute the differential evolution crossover ('best/1/bin' variant in jMetal).
        """
        if len(parents) != self.get_number_of_parents():
            raise Exception('The number of parents is not {}: {}'.format(self.get_number_of_parents(),
len(parents)))

        child = copy.deepcopy(self.current_individual)

        number_of_variables = parents[0].number_of_variables
        rand = random.randint(0, number_of_variables - 1)

        for i in range(number_of_variables):
            if random.random() < self.CR or i == rand:
                value = parents[2].variables[i] + self.F * (parents[0].variables[i] - parents[1].variables[i])

                if value < child.lower_bound[i]:
                    value = child.lower_bound[i]
                if value > child.upper_bound[i]:
                    value = child.upper_bound[i]
            else:
                value = child.variables[i]

            child.variables[i] = value

        return [child]

    def get_number_of_parents(self) -> int:
        return 3

    def get_number_of_children(self) -> int:
        return 1

    def get_name(self) -> str:
        return 'Differential Evolution crossover'


class CompositeCrossover(Crossover[CompositeSolution, CompositeSolution]):
    __EPS = 1.0e-14

    def __init__(self, crossover_operator_list:[Crossover]):
        super(CompositeCrossover, self).__init__(probability=1.0)

        Check.is_not_none(crossover_operator_list)
        Check.collection_is_not_empty(crossover_operator_list)

        self.crossover_operators_list = []
        for operator in crossover_operator_list:
            Check.that(issubclass(operator.__class__, Crossover), "Object is not a subclass of 
Crossover")
            self.crossover_operators_list.append(operator)

    def execute(self, solutions: List[CompositeSolution]) -> List[CompositeSolution]:
        Check.is_not_none(solutions)
        Check.that(len(solutions) == 2, "The number of parents is not two: " + str(len(solutions)))

        offspring1 = []
        offspring2 = []
        number_of_solutions_in_composite_solution = solutions[0].number_of_variables

        for i in range(number_of_solutions_in_composite_solution):
            parents = [solutions[0].variables[i], solutions[1].variables[i]]
            children = self.crossover_operators_list[i].execute(parents)
            offspring1.append(children[0])
            offspring2.append(children[1])
        return [CompositeSolution(offspring1), CompositeSolution(offspring2)]

    def get_number_of_parents(self) -> int:
        return 2

    def get_number_of_children(self) -> int:
        return 2

    def get_name(self) -> str:
        return 'Composite crossover'
```

文件 4：

```
import math
# from algo_problem import AlgoProblem
from jmetal.core.solution import Solution
from config import Config
import numpy as np
import itertools
# from algo_problem import AlgoProblem


class Fitness(object):
    def __init__(self):
        self.problem = None

    def reset(self):
        for task in self.problem.task_list:
            task.reset()
        for station_id, station_obj in self.problem.station_dict.items():
            station_obj.reset()

    def evaluate_fitness(self, solution: Solution):
        #  解码
        self.reset()
        solutions = solution.variables
        for index, task_id in enumerate(self.problem.task_id_list):
            per_solu = solutions[index]
            station_id_list = [self.problem.station_id_list[index1] for index1 in per_solu.variables]
            task_obj = self.problem.task_dict[task_id]
            if Config.solver_version == 3 or Config.solver_version == 4 or Config.solver_version == 5:
                station_id_list = sorted(station_id_list)
            for index2, sub_task in enumerate(task_obj.sub_tasks):
                station_id = station_id_list[index2]
                sub_task.assign_station_id = station_id
                station_obj = self.problem.station_dict[station_id]
                station_obj.sub_tasks.append(sub_task)
        if Config.solver_version == 4 or Config.solver_version == 5:
            for index, value in enumerate(solutions[-1].variables):
                task_obj = self.problem.task_dict[self.problem.task_id_list[index]]
                task_obj.value = value
        else:
            for index, task in enumerate(self.problem.task_list):
                task.value = index

        self.problem.task_list = sorted(self.problem.task_list)
        # 计算目标函数
        total_time = 0
        for index, task in enumerate(self.problem.task_list):
            change_times = []
            for sub_task in task.sub_tasks:
                station_obj = self.problem.station_dict[sub_task.assign_station_id]
                if station_obj.last_sub_task_id is not None and station_obj.last_sub_task_id != sub_task.real_id:
                    change_times.append(self.problem.distance_matrix[station_obj.last_sub_task_id, sub_task.real_id])
                station_obj.last_sub_task_id = sub_task.real_id
            change_times = sorted(change_times)
            if Config.solver_version == 5:
                # 两台清洗机
                total_1 = 0
                total_2 = 0
                for index3, time in enumerate(change_times):
                    if index3 % 2 == 0:
                        total_1 += time
                    else:
                        total_2 += time
                total_time += max(total_1, total_2)
            else:
                # 一台清洗机
                for index3, time in enumerate(change_times):
                    total_time += time
        solution.objectives[0] = total_time
```

文件 5：

```
from typing import TypeVar, List

from jmetal.config import store
from jmetal.core.algorithm import EvolutionaryAlgorithm
from jmetal.core.operator import Mutation, Crossover, Selection
from jmetal.core.problem import Problem
from jmetal.util.evaluator import Evaluator
from jmetal.util.generator import Generator
from jmetal.util.termination_criterion import TerminationCriterion

S = TypeVar('S')
R = TypeVar('R')


class GeneticAlgorithm(EvolutionaryAlgorithm[S, R]):

    def __init__(self,
                 problem: Problem,
                 population_size: int,
                 offspring_population_size: int,
                 mutation: Mutation,
                 crossover: Crossover,
                 selection: Selection,
                 termination_criterion: TerminationCriterion = store.default_termination_criteria,
                 population_generator: Generator = store.default_generator,
                 population_evaluator: Evaluator = store.default_evaluator):
        super(GeneticAlgorithm, self).__init__(
            problem=problem,
            population_size=population_size,
            offspring_population_size=offspring_population_size)
        self.mutation_operator = mutation
        self.crossover_operator = crossover
        self.selection_operator = selection

        self.population_generator = population_generator
        self.population_evaluator = population_evaluator

        self.termination_criterion = termination_criterion
        self.observable.register(termination_criterion)

        self.mating_pool_size = \
            self.offspring_population_size * \
            self.crossover_operator.get_number_of_parents()                                    //
self.crossover_operator.get_number_of_children()
```

```
        if self.mating_pool_size < self.crossover_operator.get_number_of_children():
            self.mating_pool_size = self.crossover_operator.get_number_of_children()

    def create_initial_solutions(self) -> List[S]:
        return [self.population_generator.new(self.problem)
                for _ in range(self.population_size)]

    def create_initial_solutions_by_fcfs(self) -> List[S]:
        pass

    def evaluate(self, population: List[S]):
        return self.population_evaluator.evaluate(population, self.problem)

    def stopping_condition_is_met(self) -> bool:
        return self.termination_criterion.is_met

    def selection(self, population: List[S]):
        mating_population = []

        for i in range(self.mating_pool_size):
            solution = self.selection_operator.execute(population)
            mating_population.append(solution)

        return mating_population

    def reproduction(self, mating_population: List[S]) -> List[S]:
        number_of_parents_to_combine = self.crossover_operator.get_number_of_parents()
        # print(self.evaluations)
        if len(mating_population) % number_of_parents_to_combine != 0:
            raise Exception('Wrong number of parents')

        offspring_population = []
        for i in range(0, self.offspring_population_size, number_of_parents_to_combine):
            parents = []
            for j in range(number_of_parents_to_combine):
                parents.append(mating_population[i + j])

            offspring = self.crossover_operator.execute(parents)

            for solution in offspring:
                self.mutation_operator.execute(solution)
                offspring_population.append(solution)
                if len(offspring_population) >= self.offspring_population_size:
                break

        return offspring_population

    def replacement(self, population: List[S], offspring_population: List[S]) -> List[S]:
        population.extend(offspring_population)

        population.sort(key=lambda s: s.objectives[0])

        return population[:self.population_size]

    def get_result(self) -> R:
        return self.solutions[0]

    def get_name(self) -> str:
        return 'Genetic algorithm'
```

文件 6：

```
from solver import Solver
import random
from config import Config
import time


if __name__ == '__main__':
    start = time.time()
    s = Solver()
    # s.solve()
    s.heuristic_solve()
    print("求解耗时：", time.time() - start, "秒")
```

文件 7：

```
import random

from jmetal.core.operator import Mutation
from jmetal.core.solution import BinarySolution, Solution, FloatSolution, PermutationSolution, \
    CompositeSolution, IntegerSolution
from jmetal.util.ckecking import Check

"""
.. module:: mutation
   :platform: Unix, Windows
```

```
    :synopsis: Module implementing mutation operators.

.. moduleauthor:: Antonio J. Nebro <antonio@lcc.uma.es>, Antonio Benítez-Hidalgo <antonio.b@uma.es>
"""


class NullMutation(Mutation[Solution]):

    def __init__(self):
        super(NullMutation, self).__init__(probability=0)

    def execute(self, solution: Solution) -> Solution:
        return solution

    def get_name(self):
        return 'Null mutation'


class BitFlipMutation(Mutation[BinarySolution]):

    def __init__(self, probability: float):
        super(BitFlipMutation, self).__init__(probability=probability)

    def execute(self, solution: BinarySolution) -> BinarySolution:
        Check.that(type(solution) is BinarySolution, "Solution type invalid")

        for i in range(solution.number_of_variables):
            for j in range(len(solution.variables[i])):
                rand = random.random()
                if rand <= self.probability:
                    solution.variables[i][j] = True if solution.variables[i][j] is False else False

        return solution

    def get_name(self):
        return 'BitFlip mutation'


class PolynomialMutation(Mutation[FloatSolution]):

    def __init__(self, probability: float, distribution_index: float = 0.20):
        super(PolynomialMutation, self).__init__(probability=probability)
        self.distribution_index = distribution_index
    def execute(self, solution: FloatSolution) -> FloatSolution:
        Check.that(issubclass(type(solution), FloatSolution), "Solution type invalid")
        for i in range(solution.number_of_variables):
            rand = random.random()

            if rand <= self.probability:
                y = solution.variables[i]
                yl, yu = solution.lower_bound[i], solution.upper_bound[i]

                if yl == yu:
                    y = yl
                else:
                    delta1 = (y - yl) / (yu - yl)
                    delta2 = (yu - y) / (yu - yl)
                    rnd = random.random()
                    mut_pow = 1.0 / (self.distribution_index + 1.0)
                    if rnd <= 0.5:
                        xy = 1.0 - delta1
                        val = 2.0 * rnd + (1.0 - 2.0 * rnd) * (pow(xy, self.distribution_index + 1.0))
                        deltaq = pow(val, mut_pow) - 1.0
                    else:
                        xy = 1.0 - delta2
                        val = 2.0 * (1.0 - rnd) + 2.0 * (rnd - 0.5) * (pow(xy, self.distribution_index +
1.0))
                        deltaq = 1.0 - pow(val, mut_pow)

                    y += deltaq * (yu - yl)
                    if y < solution.lower_bound[i]:
                        y = solution.lower_bound[i]
                    if y > solution.upper_bound[i]:
                        y = solution.upper_bound[i]

                solution.variables[i] = y

        return solution

    def get_name(self):
        return 'Polynomial mutation'


class IntegerPolynomialMutation(Mutation[IntegerSolution]):

    def __init__(self, probability: float, distribution_index: float = 0.20):
        super(IntegerPolynomialMutation, self).__init__(probability=probability)
        self.distribution_index = distribution_index

    def execute(self, solution: IntegerSolution) -> IntegerSolution:
        Check.that(issubclass(type(solution), IntegerSolution), "Solution type invalid")

        for i in range(solution.number_of_variables):
            if random.random() <= self.probability:
                y = solution.variables[i]
                yl, yu = solution.lower_bound[i], solution.upper_bound[i]

                if yl == yu:
                    y = yl
                else:
                    delta1 = (y - yl) / (yu - yl)
                    delta2 = (yu - y) / (yu - yl)
                    mut_pow = 1.0 / (self.distribution_index + 1.0)
                    rnd = random.random()
                    if rnd <= 0.5:
                        xy = max(0, 1.0 - delta1)
                        val = 2.0 * rnd + (1.0 - 2.0 * rnd) * (xy ** (self.distribution_index + 1.0))
                        deltaq = val ** mut_pow - 1.0
                    else:
                        xy = max(0, 1.0 - delta2)
                        val = 2.0 * (1.0 - rnd) + 2.0 * (rnd - 0.5) * (xy ** (self.distribution_index +
1.0))

                        deltaq = 1.0 - val ** mut_pow
                    # print(xy, (self.distribution_index + 1.0))
                    y += deltaq * (yu - yl)
                    if y < solution.lower_bound[i]:
                        y = solution.lower_bound[i]
                    if y > solution.upper_bound[i]:
                        y = solution.upper_bound[i]

                solution.variables[i] = int(round(y))
        return solution

    def get_name(self):
        return 'Polynomial mutation (Integer)'


class SimpleRandomMutation(Mutation[FloatSolution]):

    def __init__(self, probability: float):
        super(SimpleRandomMutation, self).__init__(probability=probability)
    def execute(self, solution: FloatSolution) -> FloatSolution:
        Check.that(type(solution) is FloatSolution, "Solution type invalid")

        for i in range(solution.number_of_variables):
            rand = random.random()
            if rand <= self.probability:
                solution.variables[i] = solution.lower_bound[i] + \
                                                  (solution.upper_bound[i]  -  solution.lower_bound[i])  *
random.random()
        return solution

    def get_name(self):
        return 'Simple random_search mutation'


class UniformMutation(Mutation[FloatSolution]):

    def __init__(self, probability: float, perturbation: float = 0.5):
        super(UniformMutation, self).__init__(probability=probability)
        self.perturbation = perturbation

    def execute(self, solution: FloatSolution) -> FloatSolution:
        Check.that(type(solution) is FloatSolution, "Solution type invalid")

        for i in range(solution.number_of_variables):
            rand = random.random()

            if rand <= self.probability:
                tmp = (random.random() - 0.5) * self.perturbation
                tmp += solution.variables[i]

                if tmp < solution.lower_bound[i]:
                    tmp = solution.lower_bound[i]
                elif tmp > solution.upper_bound[i]:
                    tmp = solution.upper_bound[i]

                solution.variables[i] = tmp

        return solution

    def get_name(self):
        return 'Uniform mutation'
```

```
class NonUniformMutation(Mutation[FloatSolution]):

    def __init__(self, probability: float, perturbation: float = 0.5, max_iterations: int = 0.5):
        super(NonUniformMutation, self).__init__(probability=probability)
        self.perturbation = perturbation
        self.max_iterations = max_iterations
        self.current_iteration = 0

    def execute(self, solution: FloatSolution) -> FloatSolution:
        Check.that(type(solution) is FloatSolution, "Solution type invalid")

        for i in range(solution.number_of_variables):
            if random.random() <= self.probability:
                rand = random.random()

                if rand <= 0.5:
                    tmp = self.__delta(solution.upper_bound[i] - solution.variables[i],
self.perturbation)
                else:
                    tmp = self.__delta(solution.lower_bound[i] - solution.variables[i],
self.perturbation)

                tmp += solution.variables[i]

                if tmp < solution.lower_bound[i]:
                    tmp = solution.lower_bound[i]
                elif tmp > solution.upper_bound[i]:
                    tmp = solution.upper_bound[i]

                solution.variables[i] = tmp

        return solution

    def set_current_iteration(self, current_iteration: int):
        self.current_iteration = current_iteration

    def __delta(self, y: float, b_mutation_parameter: float):
        return (y * (1.0 - pow(random.random(),
                               pow((1.0 - 1.0 * self.current_iteration / self.max_iterations),
b_mutation_parameter))))

    def get_name(self):
        return 'Uniform mutation'
```

```
class PermutationSwapMutation(Mutation[PermutationSolution]):

    def execute(self, solution: PermutationSolution) -> PermutationSolution:
        Check.that(type(solution) is PermutationSolution, "Solution type invalid")

        rand = random.random()

        if solution.number_of_variables - 1 < 2:
            return solution

        if rand <= self.probability:
            pos_one, pos_two = random.sample(range(solution.number_of_variables - 1), 2)
            solution.variables[pos_one], solution.variables[pos_two] = \
                solution.variables[pos_two], solution.variables[pos_one]

        return solution

    def get_name(self):
        return 'Permutation Swap mutation'


class CompositeMutation(Mutation[Solution]):
    def __init__(self, mutation_operator_list:[Mutation]):
        super(CompositeMutation,self).__init__(probability=1.0)

        Check.is_not_none(mutation_operator_list)
        Check.collection_is_not_empty(mutation_operator_list)

        self.mutation_operators_list = []
        for operator in mutation_operator_list:
            Check.that(issubclass(operator.__class__, Mutation), "Object is not a subclass of Mutation")
            self.mutation_operators_list.append(operator)

    def execute(self, solution: CompositeSolution) -> CompositeSolution:
        Check.is_not_none(solution)

        mutated_solution_components = []
        for i in range(solution.number_of_variables):

mutated_solution_components.append(self.mutation_operators_list[i].execute(solution.variables[i]))

        return CompositeSolution(mutated_solution_components)
    def get_name(self) -> str:
        return "Composite mutation operator"


class ScrambleMutation(Mutation[PermutationSolution]):

    def execute(self, solution: PermutationSolution) -> PermutationSolution:
        for i in range(solution.number_of_variables):
            rand = random.random()

            if rand <= self.probability:
                point1 = random.randint(0, len(solution.variables[i]))
                point2 = random.randint(0, len(solution.variables[i]) - 1)

                if point2 >= point1:
                    point2 += 1
                else:
                    point1, point2 = point2, point1

                if point2 - point1 >= 20:
                    point2 = point1 + 20

                values = solution.variables[i][point1:point2]
                solution.variables[i][point1:point2] = random.sample(values, len(values))

        return solution

    def get_name(self):
        return 'Scramble'
```

文件 8：

```
from algo_problem import AlgoProblem
from config import Config
from crossover import CompositeCrossover, SBXCrossover, PMXCrossover
from mutation import CompositeMutation, UniformMutation, PermutationSwapMutation
from jmetal.util.termination_criterion import StoppingByEvaluations
from genetic_algorithm import GeneticAlgorithm
from jmetal.operator.selection import RouletteWheelSelection
import pandas as pd
import os
from jmetal.util.solution import get_non_dominated_solutions, read_solutions, print_function_values_to_file, \
    print_variables_to_file
import matplotlib.pyplot as plt


class Solver(object):
    def __init__(self):
        self.problem = AlgoProblem()
        self.problem.fitness.problem = self.problem
        # self.problem.build()

    @staticmethod
    def cal_transfer_time(task, p_id):
        return max(abs(task.bay - p_id), abs(task.column - task.end_column))

    @staticmethod
    def cal_transfer_time1(pos1, pos2):
        return max(abs(pos1[0] - pos2[1]), abs(pos1[1] - pos2[1]))

    def heuristic_solve(self):
        crossover_list = []
        mutation_list = []
        for _ in self.problem.task_id_list:
            crossover_list.append(PMXCrossover(Config.crossover_probability))
            mutation_list.append(PermutationSwapMutation(Config.mutation_probability))
        crossover_list.append(SBXCrossover(Config.crossover_probability, distribution_index=20))
        mutation_list.append(UniformMutation(Config.mutation_probability))
        front = []
        algorithm = GeneticAlgorithm(
            problem=self.problem,
            population_size=Config.population_size,
            offspring_population_size=Config.offspring_population_size,
            mutation=CompositeMutation(mutation_list),
            crossover=CompositeCrossover(crossover_list),
            termination_criterion=StoppingByEvaluations(max_evaluations=Config.max_evaluations),
            selection=RouletteWheelSelection()
        )

        algorithm.run()
        front += [algorithm.get_result()]
        print("最优值：", algorithm.get_result().objectives[0])
        print_function_values_to_file(front, 'FUN.' + "'NSGAII")
        self.out_put_solution(front)

    def out_put_solution(self, front):
        writer = pd.ExcelWriter(os.path.join(Config.data_folder_path, "output.xlsx"))
        index = 0
        self.problem.fitness.evaluate_fitness(front[0])
        df = []
        columns = ["包装种类编号", "墨盒编号", "分配的插槽编号"]
        for task in self.problem.task_list:
            for sub_task in task.sub_tasks:
                df.append([task.id, sub_task.real_id, sub_task.assign_station_id])
        df = pd.DataFrame(df, columns=columns)
        df.to_excel(writer, index=False, sheet_name=str(index))
        writer._save()
```

文件 9：

```
"""插槽"""
class Station(object):
    def __init__(self):
        self.id = None

        self.sub_tasks = []
        self.distance_matrix = {}
        self.last_sub_task_id = None


    def reset(self):
        self.sub_tasks = []
        self.last_sub_task_id = None
```

文件 10：

```
"""墨盒类"""
class SubTask(object):
    def __init__(self):
        """real_id + task_id"""
        self.id = None
        self.real_id = None
        self.task_id = None
        self.sort_index = 1

        self.assign_station_id = None
        self.rank_level = self.task_id
        self.value = None
    def reset(self):
        self.assign_station_id = None
        self.sort_index = 1
        self.value = None

    def __lt__(self, other):
        if self.sort_index == 1:
            return self.value < other.value
        else:
            return self.rank_level < other.rank_level


"""包装类"""
class Task(object):
    def __init__(self):
        self.id = None
        """所需墨盒列表"""
        self.value = None
        self.sub_tasks = []


    def __lt__(self, other):
        return self.value < other.value


    def reset(self):
        for sub_task in self.sub_tasks:
            sub_task.reset()
        self.value = None
```

Part3：问题一到问题五调用 Gurobi 求解器求解

文件 1：read

```
import pandas as pd
def read_data(filename, no_time_matrix=False):
    df = pd.read_excel(filename, sheet_name=0)
    U = []
    K = []
    N = []
    for r, row in df.iterrows():
        if row['包装种类编号'] == row['包装种类编号']:
            N.append(int(row['包装种类编号']))
        if row['墨盒编号'] == row['墨盒编号']:
            U.append(int(row['墨盒编号']))

        if row['插槽序号'] == row['插槽序号']:
            K.append(int(row['插槽序号']))
    df = pd.read_excel(filename, sheet_name=1)
    S = []
    for r, row in df.iterrows():
        S.append(eval(row['所需墨盒编号']))

    if no_time_matrix:
        return range(U[0], U[-1]+1), range(N[0], N[-1]+1), range(K[0], K[-1]+1), S, None
    df = pd.read_excel(filename, sheet_name=2, index_col=0)
    time_matrix = df.values
    return range(U[0], U[-1]+1), range(N[0], N[-1]+1), range(K[0], K[-1]+1), S, time_matrix
```

文件 2：问题一 model1

```
from gurobipy import Model, quicksum

from read import read_data

def main():
    filename = '附件 1/Ins2_7_10_3.xlsx'
    U, N, K, S, time_matrix = read_data(filename, True)

    m = Model()
    # 执行 task i 时  插槽  k  是否装着墨盒 u
    x = m.addVars(N, K, U, vtype='B', name='x')
    # 执行 task i 时  插槽  k  是否需要更换墨盒
    y = m.addVars(N, K, vtype='B', name='y')
    # 每个  task i 所要求的墨盒集合 S[i]中的墨盒=1
    for i in N:
        # 这里 S[i-1]因为 python 从 0 开始计算索引
        for s in S[i-1]:
            m.addConstr(quicksum(x[i, k, s] for k in K) == 1)
    # 执行每个 task i 时,每种墨盒 u 最多只装一个
    for i in N:
        for u in U:
            m.addConstr(quicksum(x[i, k, u] for k in K) <= 1)
    # 执行每个 task i 时,每个插槽 k,只装一个墨盒
    for i in N:
        for k in K:
            m.addConstr(quicksum(x[i, k, u] for u in U) <= 1)


    # 每次 task i, 计算其更换墨盒的次数
    for i in N:
        if i + 1 not in N:
            continue
        # 对每个插槽
        for k in K:
            # 如果当前 task i 的墨盒 u 和 task i+1 的墨盒 u 相同 右边相减=0, 否则绝对值=1,代表
更换了墨盒
            for u in U:
                m.addConstr(y[i, k] >= x[i+1, k, u] - x[i, k, u])
                m.addConstr(y[i, k] >= x[i, k, u] - x[i+1, k, u])


    m.setObjective(quicksum(y[i, k] for i in N for k in K))

    m.optimize()
    for i in N:
        tmp = []
        for k in K:
            for u in U:
                if x[i, k, u].x > 0.5:
                    tmp.append(u)
        print(f'task {i}: ', tmp)
    print('目标函数:', m.objVal)
if __name__ == "__main__":
    main()
```

文件 3：问题二 model2

```
    from gurobipy import Model, quicksum

    from read import read_data


    def main():
        filename = '附件 1/Ins2_7_10_3.xlsx'
        U, N, K, S, time_matrix = read_data(filename, True)

        m = Model()
        # 执行 task i 时 插槽 k 是否装着墨盒 u
        x = m.addVars(N, K, U, vtype='B', name='x')
        #  执行 task i 时  插槽  k  是否需要更换墨盒
        y = m.addVars(N, K, vtype='B', name='y')
        #  每个  task i 所要求的墨盒集合 S[i]中的墨盒=1
        for i in N:
            #  这里 S[i-1]因为 python 从 0 开始计算索引
            for s in S[i-1]:
                m.addConstr(quicksum(x[i, k, s] for k in K) == 1)
        #  执行每个 task i 时,每种墨盒 u 最多只装一个
        for i in N:
            for u in U:
                m.addConstr(quicksum(x[i, k, u] for k in K) <= 1)
        #  执行每个 task i 时,每个插槽 k,只装一个墨盒
        for i in N:
            for k in K:
                m.addConstr(quicksum(x[i, k, u] for u in U) <= 1)


        #  每次 task i,  计算其更换墨盒的次数
        for i in N:
            if i + 1 not in N:
                continue
            #  对每个插槽
            for k in K:
                #  如果当前 task i  的墨盒 u  和  task i+1 的墨盒 u 相同  右边相减=0,  否则绝对值=1,
    代表更换了墨盒
                for u in U:
                    m.addConstr(y[i, k] >= x[i+1, k, u] - x[i, k, u])
                    m.addConstr(y[i, k] >= x[i, k, u] - x[i+1, k, u])


        m.setObjective(quicksum(y[i, k] for i in N for k in K))

        m.optimize()
        for i in N:
            tmp = []
            for k in K:
                for u in U:
                    if x[i, k, u].x > 0.5:
                        tmp.append(u)
            print(f'task {i}: ', tmp)
        print('目标函数:', m.objVal)
    if __name__ == "__main__":
        main()
```

文件 4：问题三 model3

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

from read import read_data


def main():
    filename = '附件 3/Ins1_5_5_2.xlsx'
    U, N, K, S, time_matrix = read_data(filename)

    m = Model()
    x = m.addVars(N, K, U, vtype='B', name='x')
    z = m.addVars(N, K, U, U,    vtype='B', name='z')

    # 每个印刷任务,
    for i in N:
        for s in S[i-1]:
            m.addConstr(quicksum(x[i, k, s] for k in K) == 1)
    # 每个印刷任务,每个魔盒只装一个
    for i in N:
        for u in U:
            m.addConstr(quicksum(x[i, k, u] for k in K) <= 1)
    # 每个印刷任务,每个插槽,只装一个
    for i in N:
        for k in K:
            m.addConstr(quicksum(x[i, k, u] for u in U) == 1)

    # 紧前约束: 对每个 task i 内部
    # 假设是  [3,2,1]
    # 那么说明 墨盒 3 必须在墨盒 2 选定之前,插在插槽上,  (3,2), (2,1) 两对紧前约束
    for i in N:
        # 从 墨盒序列的第二个开始 (2)
        for idx in range(1, len(S[i-1])):
            pre_box = S[i-1][idx-1] # 前一个墨盒 3
            cur_box = S[i-1][idx]    # 当前墨盒 2
            # 如果想把墨盒 2 插入插槽 k, 那么插槽 k 之前的插槽 k1 中,必须已经差了墨盒 3
            for k in K:
                m.addConstr(quicksum(x[i, k1, pre_box] for k1 in range(1, k)) >= x[i, k, cur_box])
    # 换墨盒约束
    for i in N:
        for j in N:
            if i == j :
                continue
            for k in K:
                for u1 in U:
                    for u2 in U:
                        m.addConstr(z[i, k, u1, u2] >= x[i, k, u1] + x[i + 1, k, u2] - 1)

    m.setObjective(quicksum(time_matrix[u1-1][u2-1] * z[i, k, u1, u2] for i in N for k in K for u1 in U for
u2 in U))
    m.optimize()
    for i in N:
        tmp = []
        for k in K:
            for u in U:
                if x[i, k, u].x > 0.5:
                    tmp.append(u)
        print(f'task {i}: ', tmp)
    print('目标函数:', m.objVal)
if __name__ == "__main__":
    main()
```

文件 5：问题四 model4

```
from gurobipy import Model, quicksum
from read import read_data


def main():
    filename = '附件 4/Ins4_20_40_10.xlsx'
    #filename = '附件 4/Ins2_10_30_10.xlsx'
    U, N, K, S, time_matrix = read_data(filename)
    N_list = [i for i in N]
    n = len(N_list)
    m = Model()
    x = m.addVars(N, K, U, vtype='B', name='x')
    # 是否执行完 task i  再执行 task j
    y = {(i, j): m.addVar(vtype='B', name=f'y_{i}_{j}') for i in [0] + N_list for j in N_list + [n+1] if i != j}
    z = m.addVars(N, K, U, U,    vtype='B', name='z')
    # 到达 task i 是累积完成任务数
    # 包括虚拟任务起点 0 和 虚拟任务终点 n+1
    v = {i: m.addVar(vtype='C', name=f'v_{i}') for i in [0] + N_list + [n+1]}

    # 从 0 点出发
    m.addConstr(quicksum(y[0, j] for j in N_list) == 1)
    # 回到 n+1 点
    m.addConstr(quicksum(y[j, n+1] for j in N_list) == 1)
    # 每个印刷任务 i 入度等于出度=1, 等于每个任务必须且只能执行一次
    for i in N:
        m.addConstr(quicksum(y[j, i] for j in N_list+[0] if j != i) == quicksum(y[i, j] for j in N_list+[n+1] if
j != i))
        m.addConstr(quicksum(y[j, i] for j in N_list+[0] if j != i) == 1)
    # 计算执行到 task i 的累积完成任务数
    for i in [0] + N_list:
        for j in N_list +[n+1]:
            if i == j:
                continue
            m.addConstr(v[j] >= v[i] + 1 - n*(1-y[i, j]))
    # 虚拟任务起点 0 的次数=0
    m.addConstr(v[0] == 0)
    # 每个印刷任务,
    for i in N:
        for s in S[i-1]:
            m.addConstr(quicksum(x[i, k, s] for k in K) == 1)
    # 每个印刷任务,每个魔盒只装一个
    for i in N:
        for u in U:
            m.addConstr(quicksum(x[i, k, u] for k in K) <= 1)
    # 每个印刷任务,每个插槽,只装一个
    for i in N:
        for k in K:
            m.addConstr(quicksum(x[i, k, u] for u in U) == 1)

    # 紧前约束
    for i in N:
        for idx in range(1, len(S[i-1])):
            pre_box = S[i-1][idx-1]
            cur_box = S[i-1][idx]
            for k in K:
                m.addConstr(quicksum(x[i, k1, pre_box] for k1 in range(1, k)) >= x[i, k, cur_box])

    # 墨盒更换约束
    for i in N:
        if i + 1 not in N:
            continue
        for k in K:
            for u1 in U:
                for u2 in U:
```

文件六：问题五 model5

```
from gurobipy import Model, quicksum
from read import read_data


def main():
    filename = '附件 4/Ins4_20_40_10.xlsx'
    #filename = '附件 4/Ins2_10_30_10.xlsx'
    U, N, K, S, time_matrix = read_data(filename)
    N_list = [i for i in N]
    n = len(N_list)
    m = Model()
    x = m.addVars(N, K, U, vtype='B', name='x')
    # 是否执行完 task i 再执行 task j
    y = {(i, j): m.addVar(vtype='B', name=f'y_{i}_{j}') for i in [0] + N_list for j in N_list + [n+1] if i != j}
    z = m.addVars(N, K, U, U,    vtype='B', name='z')
    # 到达 task i 是累积完成任务数
    # 包括虚拟任务起点 0 和 虚拟任务终点 n+1
    v = {i: m.addVar(vtype='C', name=f'v_{i}') for i in [0] + N_list + [n+1]}

    # 从 0 点出发
    m.addConstr(quicksum(y[0, j] for j in N_list) == 1)
    # 回到 n+1 点
    m.addConstr(quicksum(y[j, n+1] for j in N_list) == 1)
    # 每个印刷任务 i 入度等于出度=1, 等于每个任务必须且只能执行一次
    for i in N:
        m.addConstr(quicksum(y[j, i] for j in N_list+[0] if j != i) == quicksum(y[i, j] for j in N_list+[n+1] if
j != i))
        m.addConstr(quicksum(y[j, i] for j in N_list+[0] if j != i) == 1)
    # 计算执行到 task i 的累积完成任务数
    for i in [0] + N_list:
        for j in N_list +[n+1]:
            if i == j:
                continue
            m.addConstr(v[j] >= v[i] + 1 - n*(1-y[i, j]))
    # 虚拟任务起点 0 的次数=0
    m.addConstr(v[0] == 0)
    # 每个印刷任务,
for i in N:
    for s in S[i-1]:
        m.addConstr(quicksum(x[i, k, s] for k in K) == 1)
# 每个印刷任务,每个魔盒只装一个
for i in N:
    for u in U:
        m.addConstr(quicksum(x[i, k, u] for k in K) <= 1)
# 每个印刷任务,每个插槽,只装一个
for i in N:
    for k in K:
        m.addConstr(quicksum(x[i, k, u] for u in U) == 1)

# 紧前约束
for i in N:
    for idx in range(1, len(S[i-1])):
        pre_box = S[i-1][idx-1]
        cur_box = S[i-1][idx]
        for k in K:
            m.addConstr(quicksum(x[i, k1, pre_box] for k1 in range(1, k)) >= x[i, k, cur_box])

# 墨盒更换约束
for i in N:
    if i + 1 not in N:
        continue
    for k in K:
        for u1 in U:
            for u2 in U:
```