


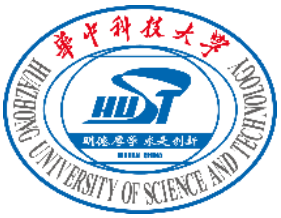

中国研究生创新实践系列大赛

中国光谷·“华为杯”第十九届中国研究生数学建模竞赛

学　　校	华中科技大学
参赛队号	2210487014**3**
队员姓名	1.徐锐
	2.李辰
	3.张世杰

中国研究生创新实践系列大赛

中国光谷·“华为杯”第十九届中国研究生数学建模竞赛

题　目　基于整数规划的方形件排样和组批优化问题研究

摘　　要：

常见的板式产品如玻璃，PCB 板，铝合金门窗等产品因其结构定制化程度高的特点，相关生产制造的企业往往采用“多品种小批量”的个性化生产模式。通过对客户订单的组批，预先规划好各个产品的排样方式，由于所有的产品均由原板材切割生成，因此这种生产模式直接引起生产效率同生产成本之间的矛盾，若能协调好生产订单组批与排样优化难题可以为企业计划排产的顺利进行提供有效的决策支持。针对此类问题，本文从组批与排样过程中的实际约束入手，建立了整数规划模型，合理配置原材料资源和订单组批，从时间和空间两个维度提高原材料利用率和生产效率。

针对问题一，首先在满足各个子集**生产订单需求**和**三阶段齐头切**等约束的条件下建立**排样整数规划模型**，在尽可能少的母板上切割出要求的产品。基于所建模型，设计了一种**三阶段启发式搜索树算法(3-Stage Heuristic Search Tree,3-SHST)**对模型进行求解。基于**宽度非递增原则**，采用**贪婪式排列算法**对原始数据进行排列组合，在 **FFDH 算法**的基础上利用**后序遍历**查找产品项可能插入的位置节点，并生成**切割树**，遍历所有树节点找到最优插入位置。最后在所给数据集 dataA1~dataA4 中进行算法验证，最终求得四个数据集上所需原板材的个数分别为 **89 块、89 块、89 块和 87 块**，板材利用率分别为 **93.87%、93.12%、94.08%和 94.08%**，排样算法运行时间分别为 **0.1064s、0.0917s、0.1289s 和 0.1112s**，所有数据集的排样示意图和相关排样信息表格分别以.jpg 和.csv 文件格式输出。

针对问题二，首先在问题一所建整数规划模型的基础上，增加对每个批次**产品项总数**和**面积总和**的约束及相关决策变量，建立**组批整数规划模型**，在材料利用率最大化的优化目标下平衡生产效率和材料利用率的关系。设计了一种**基于生产条件约束的订单层次聚类算法**，以**生产条件约束**为前提，利用**杰卡德距离**定义各个订单簇之间材料种类的区别，最后利用**凝聚层次聚类算法**得到最优的组批结果。针对各个批次的排样问题，采用问题一中的**排样优化算法**得到最优的组批排样方案。最后在所给数据集 dataB1~dataB5 中进行算法验证，最终求得五个数据集上组批次个数分别为 **55 个、41 个、51 个、44 个**和 **70 个**，使用板材个数分别为**3726 块、2498 块、2497 块、2565 块**和 **4004 块**，板材利用率分别为**79.93%、77.14%、77.44%、79.15%**和 **77.17%**，组批算法运行时间分别为 **33.441s、13.2627s、13.7917s、11.3077s** 和 **47.0652s**。所有数据集的排样示意图和相关组批信息表格分别以.jpg 和.csv 文件格式输出。

关键词：二维库存切割；排样优化；组批优化；整数规划

目录

一、问题背景与重述

1.1 问题背景

随着数字化、网络化、智能化的工业浪潮席卷全球，“智能制造”已被列为“中国制造 2025”的主攻方向。在机械制造、纺织加工、印刷作业等领域经常会遇到这样的问题：从一组大的矩形原料上按需求切割下一定尺寸一定需求量的小矩形，按照最朴素的节约资源降低生产成本的想法，希望在使用原材料最少的前提下完成订单需求，这就是典型的二维切削库存（2DCS，two-dimensional cutting stock）问题[1]，其变体问题为二维装箱（2DBP，two-dimensional bin packing）问题[2]。题干中介绍的三阶段的齐头切模式常被用于作为解决上述问题的限制条件之一，这是因为这种切割方式在材料利用率和切割过程的复杂性之间取得了很好的平衡[3]。整数线性规划（ILP，Integer linear programming）常被用于二阶段和三阶段的齐头切 2DCS 或 2DBP 问题[4]，文献[2]将 Lodi 等人的混合整数规划（MILP，The Mixed-Integer Linear Programming）模型[5]扩展到三阶段模型并用于列生成以解决 2DBP 问题。同时若在原材料利用率最大的要求上加上生产批次与交货时间约束、切割次数最少约束等附加条件则会使优化目标增多，约束条件也更加复杂，求解也变得更为困难。本赛题正是在此背景下要解决小批次多种类的订单的二维切削库存问题的排样优化和订单组批问题。

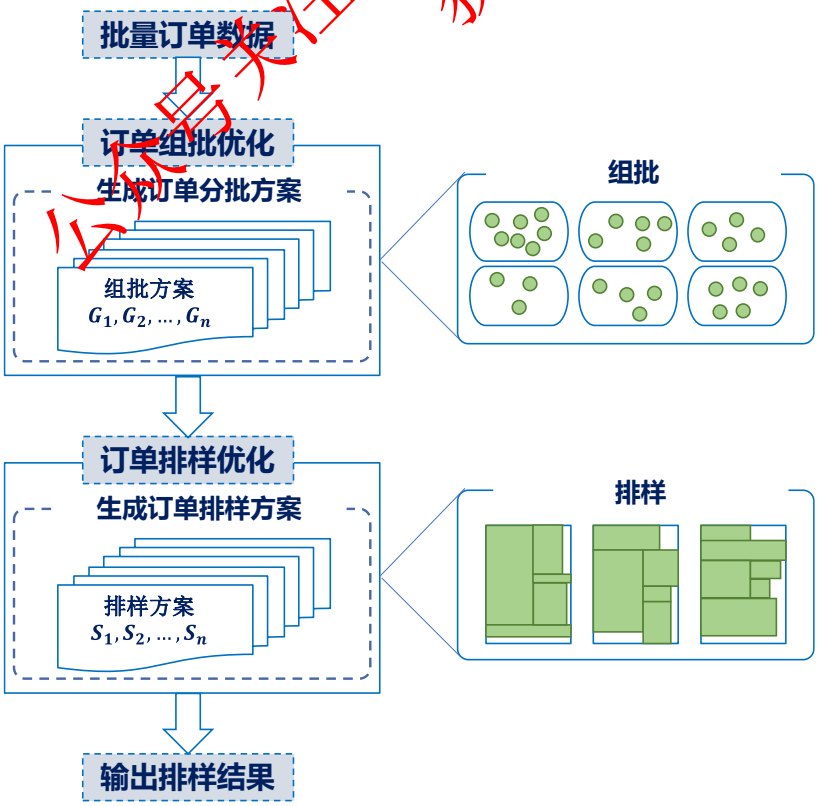

图 1 组批与排样流程

1.2 问题重述

基于上述研究背景，题目提供了两个独立的数据集 A 和 B，分别包含产品项数量、需求、尺寸、产品 id、订单号等信息，本文需要解决以下两个问题：

问题 1：排样优化问题。对数据集 A 中的每个子集，在满足各子集生产订单需求和三阶段齐头切约束条件下，建立相关的模型并设计求解算法，使用最少的相同尺寸的原板材来完成订单生产需求，使板材利用率达到最大化，并按格式输出排样图和排样信息表格。此问题中未对订单号作相关要求，且同一子集中产品的材料也相同，因此这两项数据可以不做考虑。这是一个典型的三阶段齐头切 2DCS 或 2DBP 问题。

问题 2：订单组批问题。对数据集 B 中的每个子集，在问题一约束的基础上，增加以下进一步约束：

1) 每份订单当且仅当出现在一个批次中；
2) 每个批次中的相同材质的产品项（item）才能使用同一块板材原片进行排样；
3) 为保证加工环节快速流转，每个批次产品项（item）总数不能超过 1000；
4) 因工厂产能限制，每个批次产品项（item）的面积总和不能超过 250㎡；

要根据上述所有条件，建立规划模型并设计相应的求解算法，对数据集 B 中的全部订单进行组批，然后对每个批次进行独立排样，使得板材原片的用量最少。本问题虽然最终的优化目标与问题一相同，但由于加上了同批次 item 数量、同批次 item 面积和以及同材质 item 排样三个条件，实质上是在问题一的基础上升了一个维度，提出了一个子问题，即如何组批才能平衡生产效率和材料利用率的关系，使得材料利用率最大化。同时数据集 B 的数据量比 A 要大的多，这也就对模型和算法的求解效率提出了需求。

1.3 技术路线图

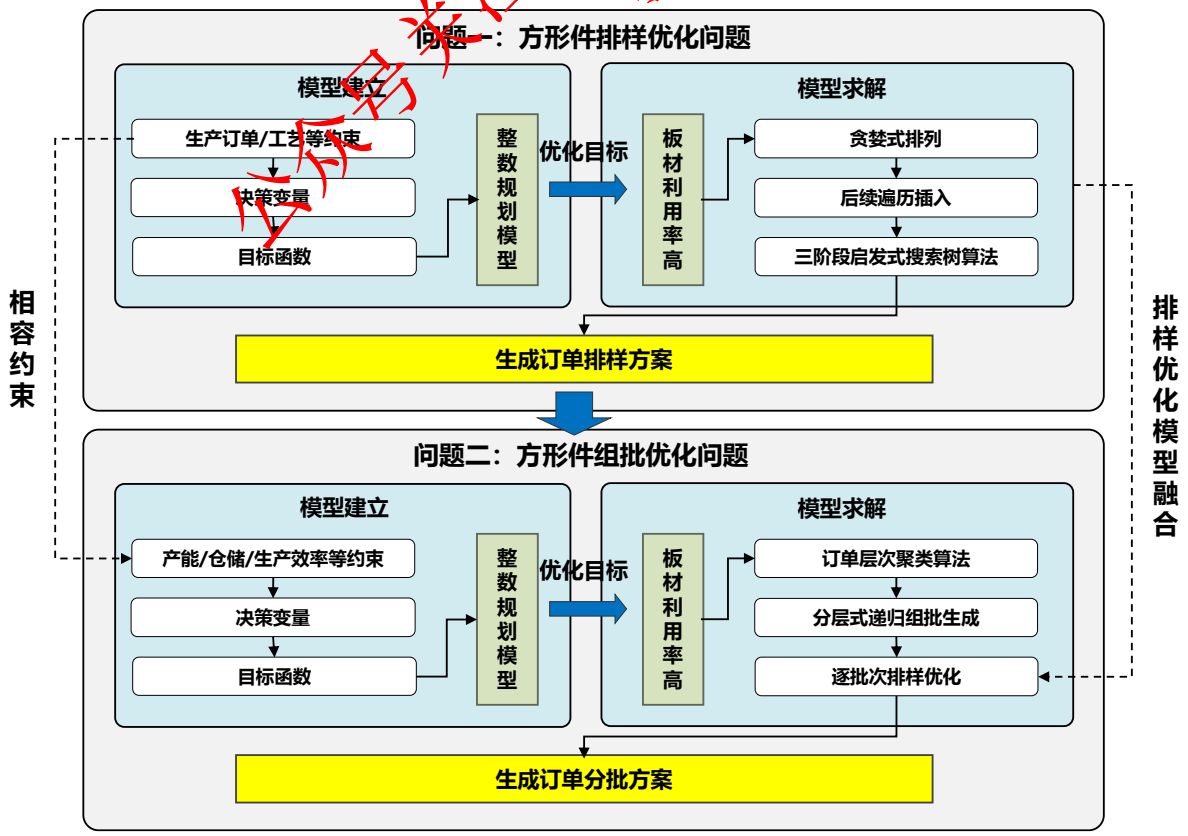

图 2 本文技术路线图

二、基本假设和符号说明

2.1 基本假设

基于题干和实际求解中可能考虑到的因素，在求解之前我们需做出以下假设以保证模型和算法的逻辑完备：

（1）本次赛题所有订单的交货期均相同，不做区分；

（2）只考虑齐头切的切割方式（直线切割、切割方向垂直于板材一条边，并保证每次直线切割板材可分离成两块）；

（3）切割阶段数不超过 3，同一个阶段切割方向相同；

（4）排样方式为精确排样；

（5）假定板材原片仅有一种规格且数量充足；

（6）排样方案不用考虑锯缝宽度（即切割的缝隙宽度）影响；

（7）数据集 A 和 B 中的各子集数据是独立的，应分别处理。

2.2 符号说明

针对贯穿全文的一些主要变量及其符号含义列表如下。

表 1 符号及变量定义说明

符号	符号含义	单位
$Area_i$	数据集 A_i 中 *item* 面积总和	m^2
L	所使用板材原片的长	*mm*
W	所使用板材原片的宽	*mm*
l_i	*item* 的长	*mm*
w_i	*item* 的宽	*mm*
d	所有 *stripe* 的尺寸类别数	个
t	能生成 *stripe* 的 *stack* 的个数	个
m	所有 *stripe* 的最大顺序编号	
q_{k_a}	0/1 变量，顺序编号为 k 的 *stack* 是否可以被编号为 a 的 *stripe* 包含	
y_{i_a}	0/1 变量，顺序编号为 i 的 *item* 是否可以被编号为 a 的 *stack* 包含	
x_a	正整数变量，在顺序编号为 a 的 *stack* 中包含 *item* 的个数	个
z_a	正整数变量，在顺序编号为 a 的 *stripe* 中包含 *stack* 的个数	个
$J(A,B)$	杰卡德相似系数	
$J_\delta(A,B)$	杰卡德距离	

三、问题一：方形件排样优化问题求解

3.1 数据分析与思考

首先对给出的 dataA1-dataA4 四组数据进行粗略的观察和分析，可分别计算出每组数据中 item 的面积之和，以 dataA1 数据为例，item 面积之和为：

$$Area_1 = \sum_{i=0}^{752} length_i * width_i = 2.4868561455 \times 10^2 (m^2) \tag{3.1}$$

用面积除以原板材面积可得极限情况下最小使用的板材数：

$$Area_1 \div (1220 \times 2440 \times 10^{-6}) = 83.54 \approx 84 \tag{3.2}$$

以 item 面积之和除以极限情况下使用板材面积之和可得 dataA1 所提供数据的原材料极限利用率：

$$Area_1 / (84 * 2440 * 1220 \times 10^{-6}) = 99.45\% \tag{3.3}$$

当然，由于题干中给出的不能拼接等条件以及三阶段齐头切方式所带来的必然的材料浪费，实际的原材料利用率不肯可能达到以上计算值，同样的，如果模型及算法的计算结果显示利用率超过此值那说明结果也是错误的。对其他三组数据进行同样的处理得出的结果如表 2 所示：

表 2 极限情况下材料利用率分析

数据集	dataA1	dataA2	dataA3	dataA4
极限情况下所需原材料块数	84	83	84	82
极限利用率	99.45%	98.66%	99.68%	97.44%

继续对数据进行观察，发现每个数据集中的方形件的需求量都是 1 且很少有长宽完全一样的两个方形件，但是若以长度或者宽度来进行排序，可以发现有不少的方形件具有同样的长度或者宽度，这也符合题干中所说的同一个 stack 中 item 的长或宽需相等的条件。为了更加直观的看出方形件的外观尺寸分布情况，利用数据集中的数据在 origin 软件中画出如图 3 所示散点频数统计图，X、Y 坐标分别是方形件的长和宽，XY 平面内是方形件的长宽散点图，在 XZ 和 YZ 平面内我们作出了某一长度或宽度区间段内方形件的频数统计直方图，可以看出在某些区间内方形件的长或宽特征体现出高度集中的特征。

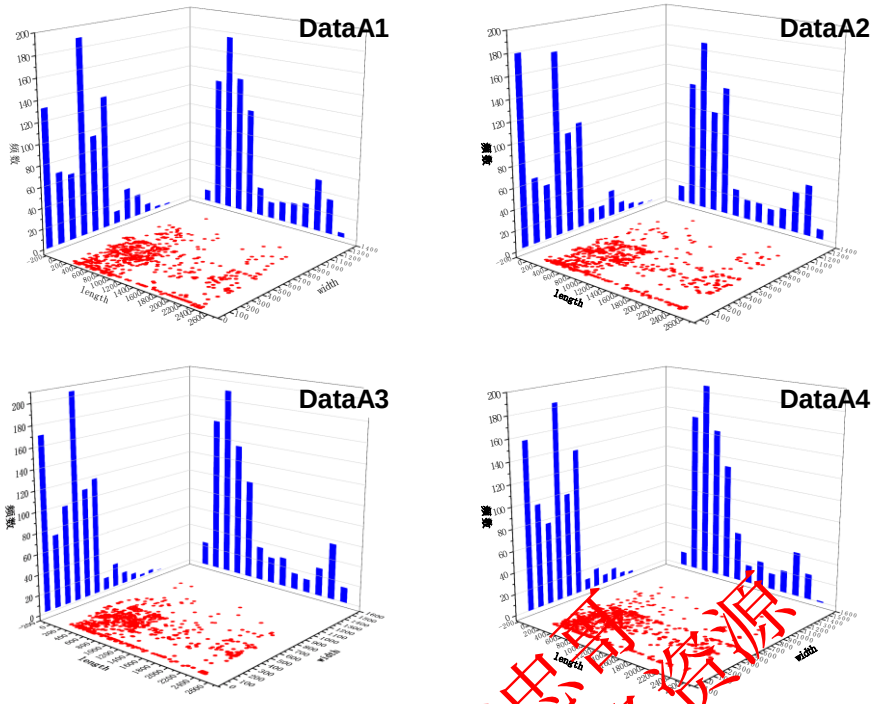

图 3 四个数据集的散点频数统计图

利用上述特征分析的结果呈现了数据的整体概貌，根据以上分析的结果，本文提出了二维库存切割问题的整数规划模型并设计了相应的求解算法，并对模型和算法进行了相应的解释与评价，最终输出了相应格式的结果。问题一求解思路如图 4 所示。

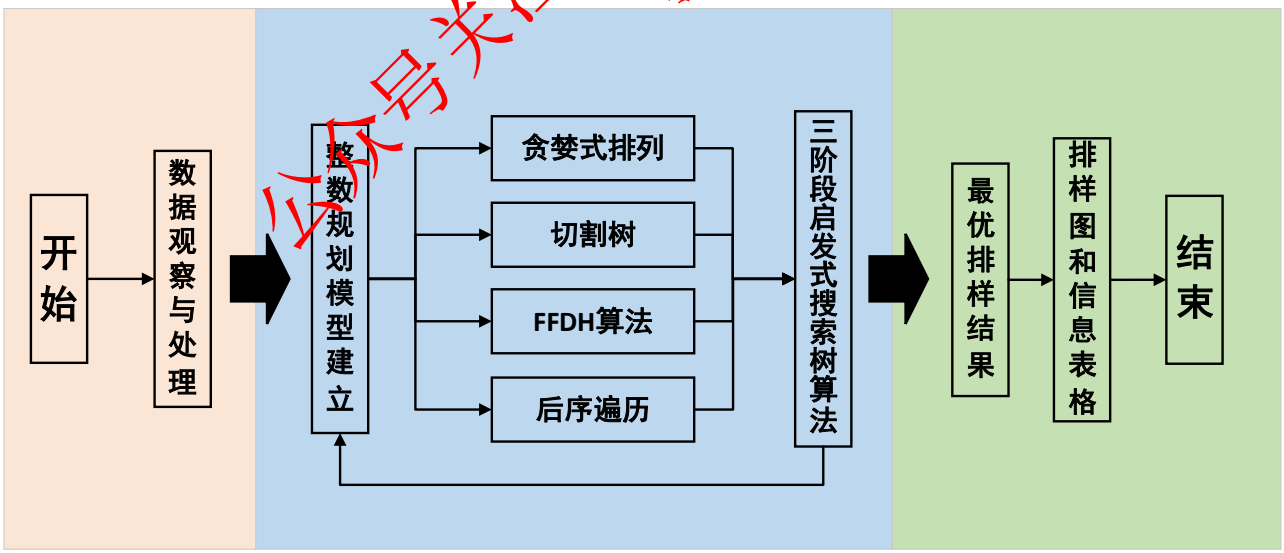

图 4 问题一求解思路

3.2 排样优化问题模型建立与求解

3.2.1 排样问题整数规划模型建立

针对问题一，优化模型要求根据同一生产批次内方形件的尺寸与数量,选择原片的规格和数量,进行下料排样优化,以最大化板材原片利用率。并需要满足以下两个约束：

（1）在相同栈（stack）里的产品项（item）的宽度（或长度）应该相同；
（2）最终切割生成的产品项是完整的，非拼接而成。

优化的目标是要满足尽可能少的板材原片数量，由于本问题为典型的 NP-hard 问题，一般没有 polynomal 复杂度的算法，算法复杂度随着变量的增加呈指数级增长（指数爆炸），因此在上述题干约束的基础上引申做出以下约束：

（1）切割方式固定为第 1 阶段采用横向切割生成 stripe（条带），第 2 阶段采用纵向切割生成 stack（栈），第三阶段采用横向切割生成 item（产品项）；
（2）每个 stack 中的第一个(最上面)元素是 stack 中最宽的元素；
（3）每个 stripe 中的第一个(最左边)元素是 stripe 中最宽的元素；
（4）所有被选择的 item 的宽度按照不递减排列；

为解释模型需要，对问题一涉及的符号作进一步说明：

（1）t 表示不同 item 的类型编号，其中 $t \in \{1,\ldots,m\}$，m 表示所有 item 的类别数。用 e_j 表示每种 item 的需求数量，其中 $j \in \{1,\ldots,t\}$；

（2）假设最终可能会产生 d 种不同尺寸的 stripe，对于每一个 stripe，都可以通过他们的尺寸唯一确定其类型，用符号 l 表示，其中 $l \in \{1,\ldots,d\}$；

（3）最终产生的 stack 个数为 m，每个 stack 的产生都源于最初插入了一个顺序编号为 i 的 item，记此 stack 的顺序编号为 a；与此类似，假定最终可能会产生 m 个 stripe，每一个 stripe 的产生都源于最初插入了一个顺序编号为 k 的 stack，记此 stripe 的顺序编号为 a。这里符号不进行区分只用来表明元素的产生与其组成之间存在上述关系。

基于上述所有约束条件，我们选择通过整数规划对该问题进行建模，模型如下：

（一）决策变量：

(1) $q_{k_a}^l \in \{0,1\}$：当且仅当顺序编号为 k 的 stack 可以被顺序编号为 a，尺寸类型为 l 的 stripe 包含时 $q_{k_a}^l = 1$，否则 $q_{k_a}^l = 0$，其中 $k = 1,\ldots,t; l = 1,\ldots,d; a = 1,\ldots,m$。

(2) $y_{i_a}^l \in \{0,1\}$：当且仅当顺序编号为 i 的 item 可以被顺序编号为 a，尺寸类型为 l 的 stack 包含时 $y_{i_a}^l = 1$，否则 $y_{i_a}^l = 0$，其中 $i = 1,\ldots,t; l = 1,\ldots,d; a = 1,\ldots,m$。

(3) $x_{i_a j}^l \in \mathbb{Z}^+$：表示在顺序编号为 a，尺寸类型为 l 的 stack 中，包含尺寸类型为 j 的 item 的个数。此 stack 因尺寸类型为 i 的 item 的加入而产生。其中 $i = 1,\ldots,t;\ j \geq i; l = 1,\ldots,d; a = 1,\ldots,m$。

(4) $z_{k_a i}^l \in \mathbb{Z}^+$：表示在顺序编号为 a，尺寸类型为 l 的 stripe 中，包含尺寸类型 i 的 stack 的个数，此 stripe 因尺寸类型为 k 的 stack 的加入而产生，其中 $k = 1,\ldots,t; i \geq k; l = 1,\ldots,d; a = 1,\ldots,m$。

（二）目标函数：

为了保证所使用的板材数量尽量少，我们考虑将目标函数定义为最小化所有被使用的

stripe 的和。$q_{k_a}^l$ 保证了当且仅当顺序编号为 k 的 stack 可以被顺序编号为 a，尺寸类型为 l 的 stripe 包含时该高度才能累加。

$$Z = \min \sum_{l=1}^{d} \sum_{k=1}^{t} \sum_{a=1}^{m} q_{k_a}^l L_l \tag{3.4}$$

（三）约束条件：

1.采用等式约束使所有 item 均按照需求合理安排进原片，确保了每个尺寸类型为 j 的 item 都能够被安排 e_j 次，如式(3.5)所示：

$$\sum_{l=1}^{d} \sum_{a=1}^{m} \left(\sum_{i=1}^{j} x_{i_a j}^l + y_{j_a}^l \right) = e_j \qquad j = 1, \ldots, t \tag{3.5}$$

2.同理，式(3.6)确保每个被安排的 stack 都在已使用的 stripe 内：

$$\sum_{a=1}^{m} \left(\sum_{k=1}^{i} z_{k_a i}^l + q_{i_a}^l \right) = \sum_{a=1}^{m} \left(y_{i_a}^l \right) \qquad i = 1, \ldots, t; l = 1, \ldots, d \tag{3.6}$$

3.每个 stack 在插入 stripe 时需考虑其宽度是否超出当前 stripe 的剩余宽度，式(3.7)确保每个被插入的 stack 的宽度不超过当前尺寸类型为 l 的 stripe 的宽度。

$$\sum_{a=1}^{m} w_i z_{k_a i}^l \le \left(W_l - w_k \right) q_{k_a}^l \qquad k = 1, \ldots, t; a = 1, \ldots, m; l = 1, \ldots, d \tag{3.7}$$

4.每个 item 在插入 stack 时需考虑其高度是否超出当前 stack 的剩余高度，式(3.8)确保每个被插入的 item 的高度不超过当前尺寸类型为 l 的 stack 的高度。

$$\sum_{j=i}^{m} h_j x_{i_a j}^l \le \left(L_l - h_i \right) y_{i_a}^l \qquad i = 1, \ldots, t; a = 1, \ldots, m; l = 1, \ldots, d \tag{3.8}$$

5.式(3.9)确保顺序编号为 k 的 stack 不可包含于多种尺寸类型的 stripe，具有唯一性。

$$\sum_{l=1}^{d} q_{i_a}^l \le 1 \qquad k = 1, \ldots, t; a = 1, \ldots, m \tag{3.9}$$

6.式(3.10)确保顺序编号为 i 的 item 不可包含于多种尺寸类型的 stack，具有唯一性。

$$\sum_{l=1}^{d} y_{i_a}^l \le 1 \qquad i = 1, \ldots, t; a = 1, \ldots, m \tag{3.10}$$

综上所述，根据问题一所建立的整数规划模型总结如下：

$$Z=\min\sum_{l=1}^{d}\sum_{k=1}^{t}\sum_{a=1}^{m}q_{k_a}^{l}L_l$$

$$s.t.\begin{cases}\sum_{l=1}^{d}\sum_{a=1}^{m}\left(\sum_{i=1}^{j}x_{i_a j}^{l}+y_{j_a}^{l}\right)=e_j & j=1,\ldots,t\\ \sum_{a=1}^{m}\left(\sum_{k=1}^{i}z_{k_a i}^{l}+q_{i_a}^{l}\right)=\sum_{a=1}^{m}\left(y_{i_a}^{l}\right) & i=1,\ldots,t;\ l=1,\ldots,d\\ \sum_{a=1}^{m}w_i z_{k_a i}^{l}\le\left(W_l-w_k\right)q_{k_a}^{l} & k=1,\ldots,t;\ a=1,\ldots,m;\ l=1,\ldots,d\\ \sum_{j=i}^{m}h_j x_{i_a j}^{l}\le\left(L_l-h_i\right)y_{i_a}^{l} & i=1,\ldots,t;\ a=1,\ldots,m;\ l=1,\ldots,d\\ \sum_{l=1}^{d}q_{i_a}^{l}\le 1 & k=1,\ldots,t;\ a=1,\ldots,m\\ \sum_{l=1}^{d}y_{i_a}^{l}\le 1 & i=1,\ldots,t;\ a=1,\ldots,m\end{cases}\tag{3.11}$$

3.2.2 三阶段启发式搜索树算法设计

上节中本文提出了针对问题一即二维三阶段库存切割问题的混合整数规划模型，在本节中提出了一种求解该模型的三阶段启发式搜索树算法（3-SHST）。目前针对混合整数规划模型的解决方式主要有三种：一是精确算法（Exact Algorithm），如割平面法、分支定界法等，这类算法通常针对的是规模比较小的问题，求到的解也常常是全局最优解；二是近似算法（Approximate Algorithm），针对特定问题，使用一些分解问题难点的技巧如降维、贪婪、松弛等，来给出一个近似最优解，这种算法比较受场景限制；第三种方法就是启发式算法（Heuristic Algorithm），启发式算法的理论严谨性没有前两种强，往往是设计者根据经验或自然现象观察的结论，但是在求解复杂问题时可能有意向不到的效果[6]。针对问题一各数据集 item 长宽特征各不相同、需求量小但是总数多的特点，本文结合相关参考文献[7]提出一种三阶段启发式搜索树算法。

首先我们针对三段式切割问题的特点我们引入切割树的概念。一个切割树代表一种切割方式，它的每个节点代表通过某种齐头切的方式切割出的一个矩形，如图 5 所示

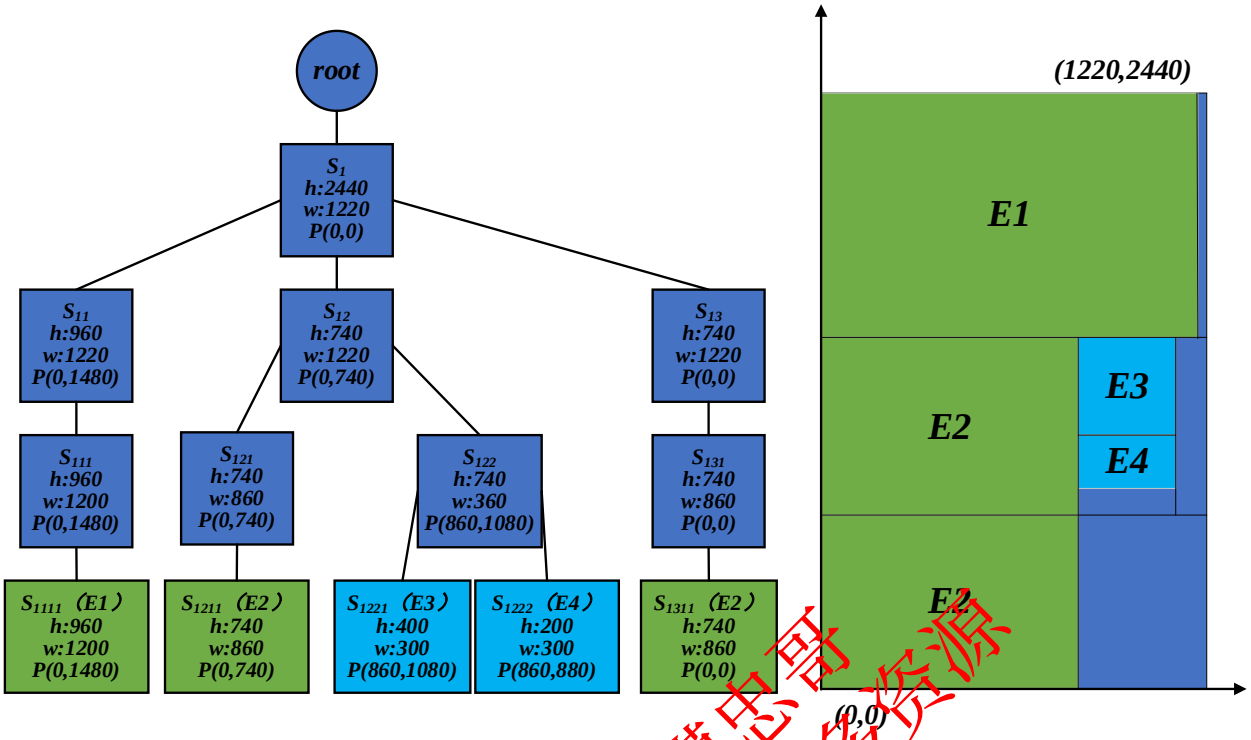

图 5 切割树与对应的排样方案

图中每个子节点中的 h 代表矩形高度，w 代表宽度，P 是矩形左上角点的坐标，坐标系的摆放如右图所示。图中根节点对应的级别为-1，它代表整个切割模式，根节点是所有零级节点即 $S=\{S_1,S_2,...,S_n\}$ 的父节点，可以看作是切割的第零阶段，第零级子节点代表未切割时的原板材，第一级子节点代表第一刀切出来的三个矩形，第二级子节点代表第二刀切出的矩形，后续过程以此类推。为了确保元素始终能够显示到最后一级叶节点，如果有切割过程在第三阶段之前就完成的话那么此过程始终会生成只有一个节点的子节点。如图中的 *E1* 矩形产生的过程。每个子节点中都存储了齐头切获得的矩形的长宽以及它在原板材中的绝对位置坐标。

反过来看以上过程其实可以看做在一个固定大小的板上排列小矩形的问题，本算法的排布原则是在宽度方向上所有矩形的宽度总是非递增的，同样的在高度方向上矩形堆叠时高度也总是非递增，这样就保证了矩形的排布空间与废料总是在原材料的边缘。

整个算法的步骤如下伪代码所示：

3-SHST 算法伪代码
输入：所有产品项信息，原片长度，原片宽度 输出：排样方案，板材利用率
1. 按非递增的宽度顺序排列产品项 Items 2. for 产品项 i：所有产品项 Items： 3. ⎸ for 切割树节点 n：切割树的后序遍历列表： 4. ⎸ ⎸ 计算当前节点 n 的剩余空间和维度信息 5. ⎸ ⎸ if 有足够空间容纳产品项 i： 6. ⎸ ⎸ ⎸ 在对应位置插入产品项 i 7. ⎸ ⎸ end 8. ⎸ end 9. ⎸ if 产品项 i 未插入到切割树中： 10. ⎸ ⎸ 生成新原片插入到切割树根节点，将产品项 i 放置到新原片上 11. ⎸ end 12. end 13. 返回切割树根节点，生成排样方案并计算板材利用率

3-SHST 算法流程图如图 6 所示：

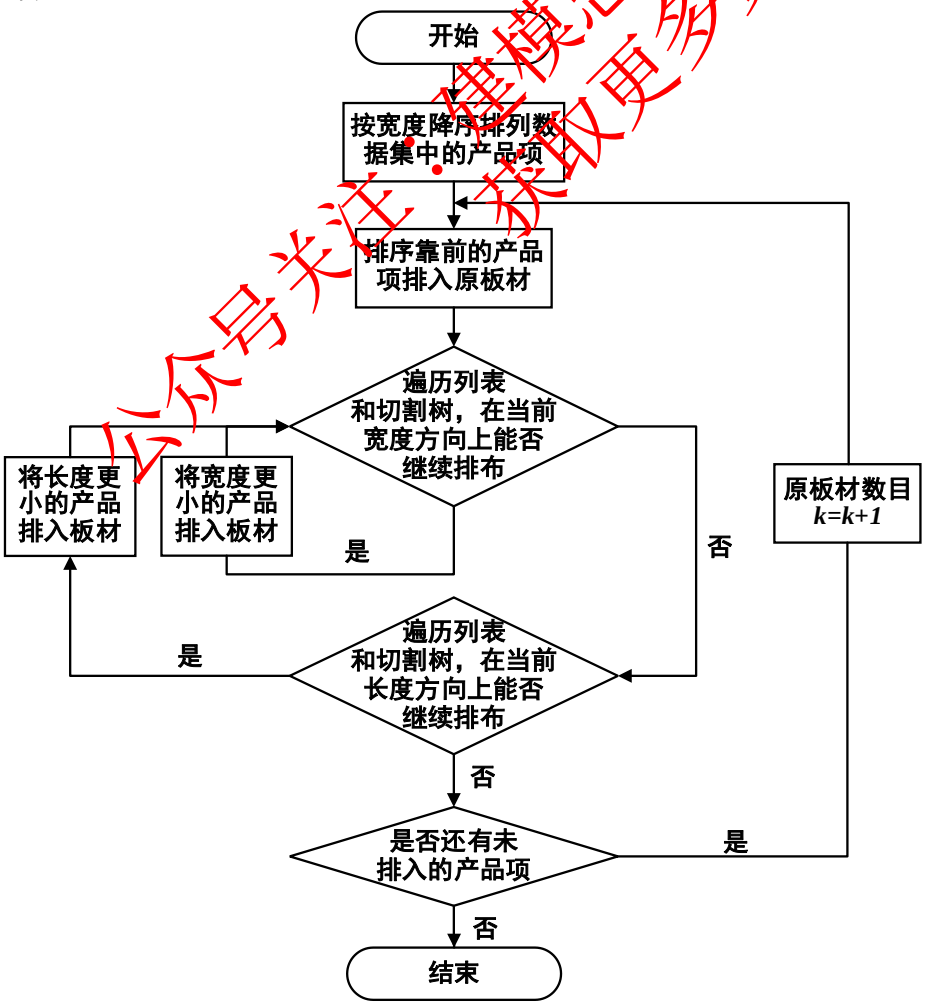

图 6 3-SHST 算法流程图

3-SHST 算法使用后序遍历方法遍历整个切割树，后序遍历是二叉树遍历的一种，也

叫做后根遍历、后序周游，在此遍历过程中总是先遍历左子树，然后遍历右子树，最后访问根节点。在具体实现中，本文将后序遍历推广到切割树这种的多叉树中，即先按序递归遍历子节点，最后访问父节点。有关后序遍历的遍历顺序如图 7 所示。

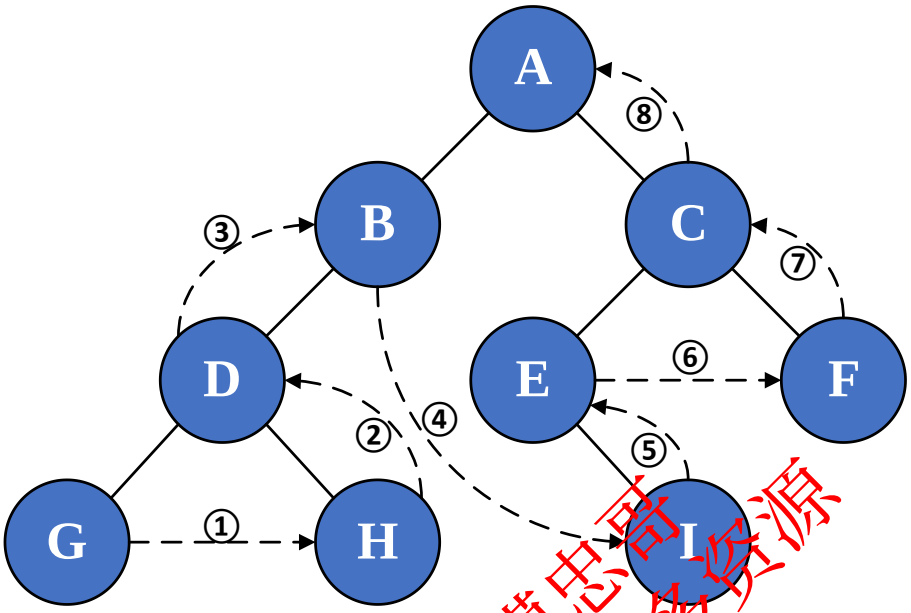

图 7 后序遍历顺序

以图 5 为例，在搜索时首先遍历 $S_{1111}(E_1)$ 节点，随后依次遍历 S_{111}、S_{11}、$S_{1211}(E_2)$、S_{121}、$S_{1221}(E_3)$、$S_{1222}(E_4)$、S_{122}、S_{12}、$S_{1311}(E_2)$、S_{131}、S_{13}、S_1 节点，这样就完成了对该切割树中所有节点的遍历，也就获取了其中已排样的产品项的信息。同时对各产品项的遍历的顺序（即 E_1、E_2、E_3、E_4、E_2）也符合前文中提出的排样逻辑：在宽度和高度方向上产品项的宽度和高度总是非递增的。

这里用图 5 中 S_{1221} 节点的生成过程来展示一步具体的排布操作。如图 8 所示，在左侧部分时程序已经按后续遍历方式遍历了五个节点，来判断是否有足够的空间放置 E_3，直到遍历到 S_{12} 节点时才发现板材还还有足够的空间来容纳 E_3，这时程序就会开辟空间生成新的 S_{122} 和 S_{1221} 节点，并将产品项 E_3 的信息存入 S_{1221} 节点中，完成了 E_3 在原板材中的排布。

在最坏情况下，3-SHST 算法在插入一个新产品项时需要遍历整棵树。因此 3-SHST 算法的**时间复杂度为 $O(m^2)$**，m 为产品项的总数。

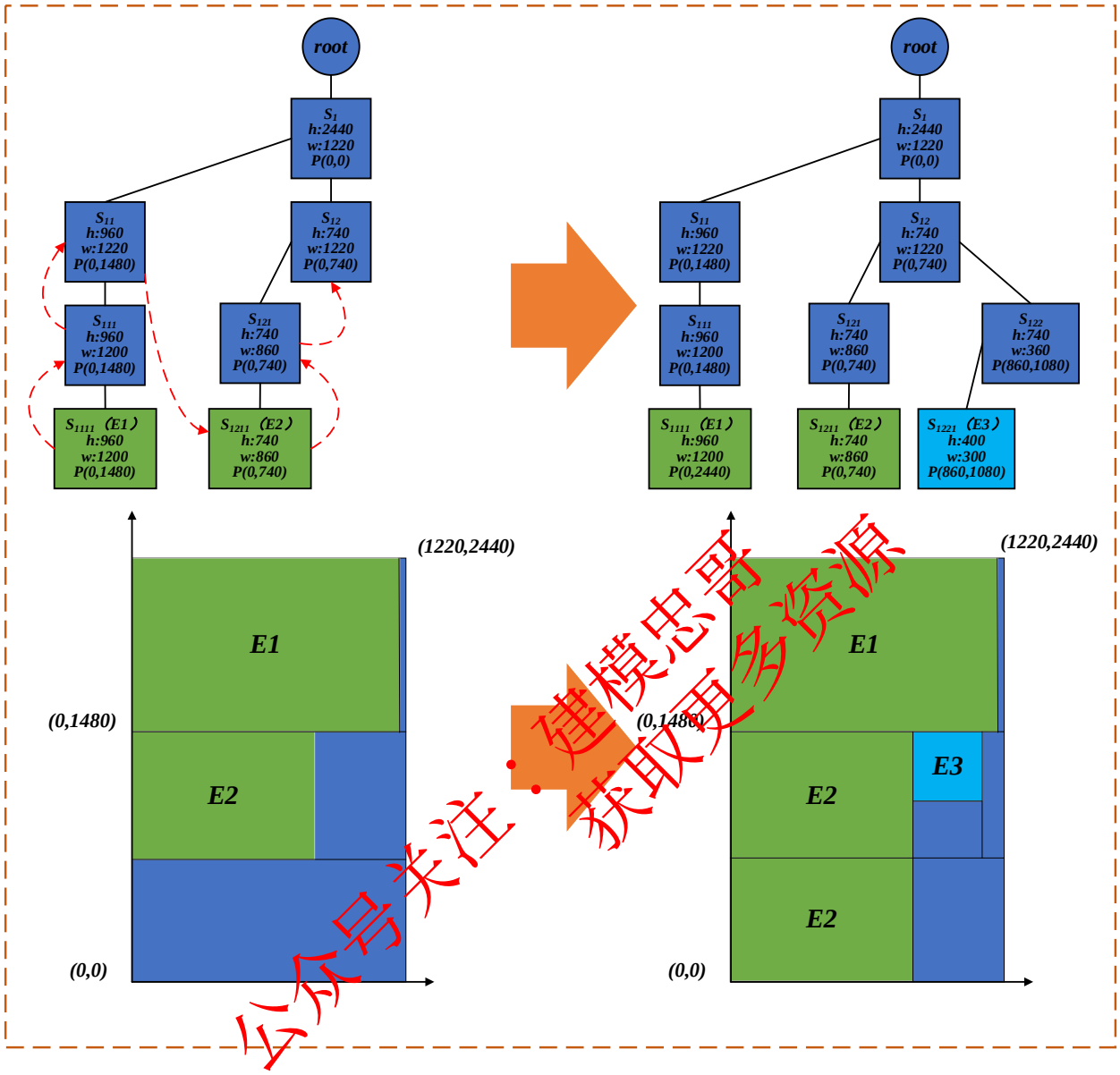

图 8 S_{1221} 节点的生成过程

3.2.3 计算结果

论文算法使用 python3.8 实现，运行在 12 核、睿频 4.5GHz 的 Inter 处理器 i5-12500H 上，最终运行求解出的排样结果以数据集 A1 为例，一共使用原板材 89 块，利用率达 93.87%，排样程序运行时间仅 0.1064s，整体排样效果图如图 9 所示（黑色区域代表未使用）：

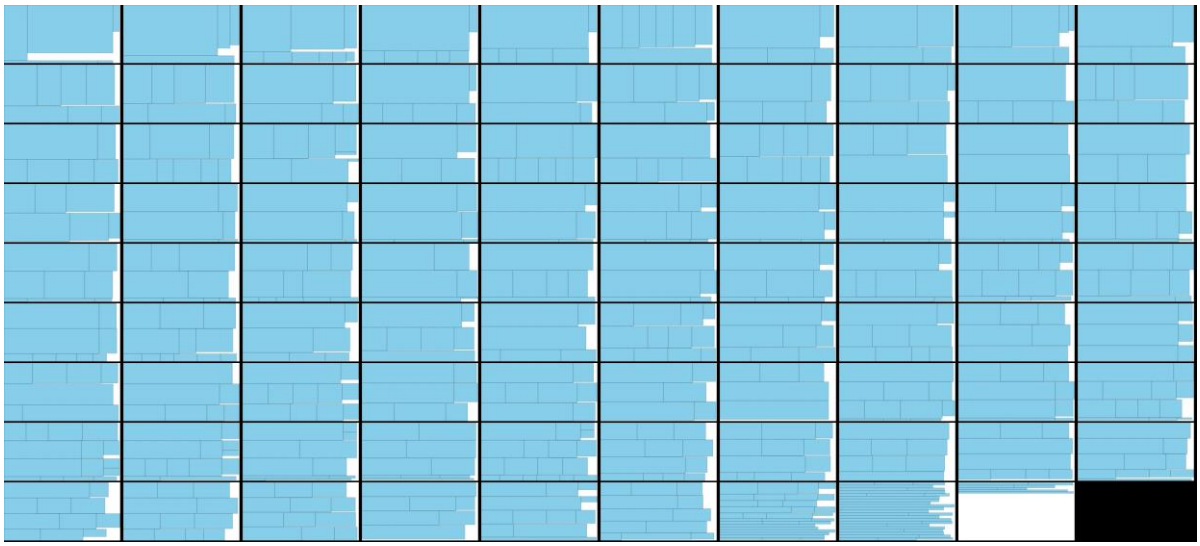

图 9 数据集 A1 排样效果

从图中可以看出排样是按照严格的三阶段齐头切来进行的，并且每块板的排样都符合我们在算法设计时所述逻辑。同时，由于 3-SHST 是通过后序遍历的方式遍历整棵切割树，在插入一个产品项时算法会考虑所有已有的板材，因此最后几块的板材可能会由于不被有效的访问导致较低的利用率，从图 9 中也可以看出数据集 A1 的最后一块板材有较低的利用率，本质上，算法通过牺牲部分板材利用率提高了总体的利用率。现以图中第 71 块原板材的排样分布效果图及对应的输出表格中的内容相互对照说明算法的正确性。

数据集 A1 中第 71 块原板材的排样效果图如图 10 所示：

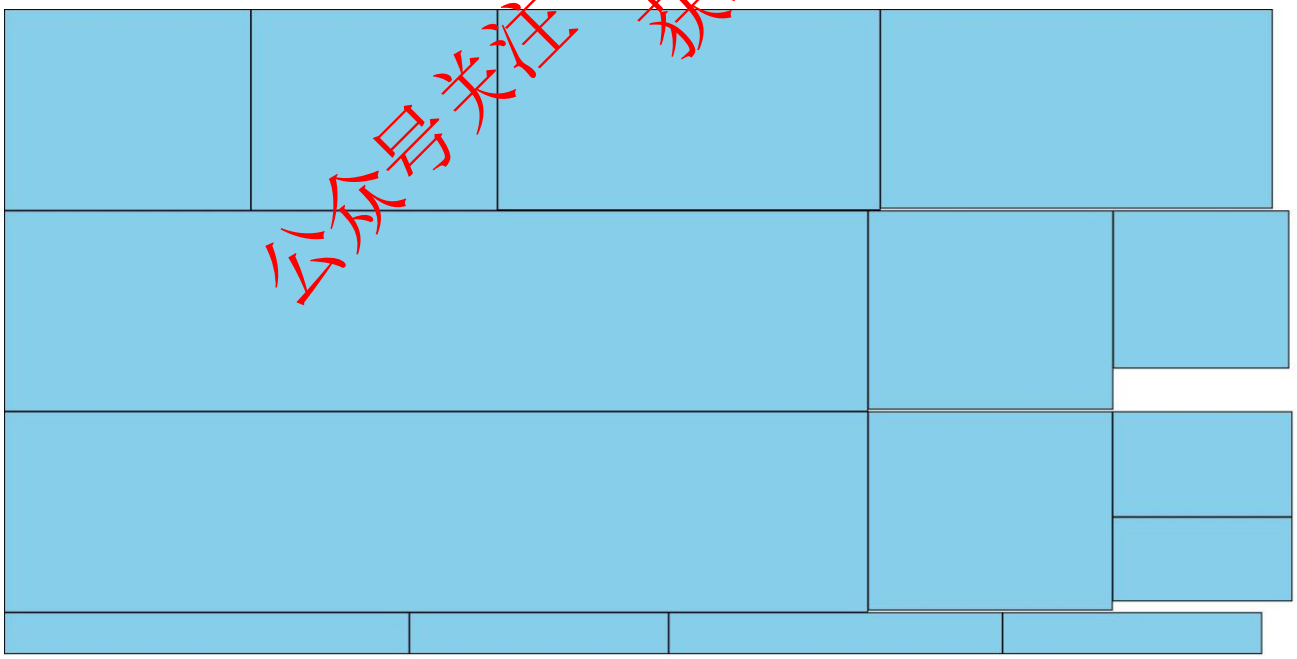

图 10 数据集 A1 中第 71 块原板材的排样效果图

它对应的输出信息表格的内容对应附件 cut_program_A1 的第 496-510 行，现列表如下：

表 3 数据集 A1 中 71 块原板材的排样信息

原片材质	原片序号	产品 id	产品 x 坐标	产品 y 坐标	产品 x 方向长度	产品 y 方向长度
YW10-0218S	70	665	0	841.2	464	378.8
YW10-0218S	70	666	464	841.2	464	378.8
YW10-0218S	70	731	928	843	720	377
YW10-0218S	70	555	1648	846	738	374
YW10-0218S	70	20	0	464.2	1625	377
YW10-0218S	70	14	1625	468.2	461	373
YW10-0218S	70	313	2086	545.2	331	296
YW10-0218S	70	132	0	87.2	1625	377
YW10-0218S	70	175	1625	91.2	460	373
YW10-0218S	70	23	2085	266.2	338	198
YW10-0218S	70	642	2085	108.2	338	158
YW10-0218S	70	210	0	9.2	762	78
YW10-0218S	70	226	762	9.2	488	78
YW10-0218S	70	323	1250	9.2	628	78
YW10-0218S	70	761	1878	9.2	488	78

表中的 x，y 坐标是产品的左底角 x，y 坐标，坐标原点(0,0)位于原片的左底角。对比图表可知，无论是原板材上的 item 数量、尺寸或者位置都是可以匹配的，故可以证明程序给出的排样效果没有问题。简单计算可知该块原片的利用率为：

$$\sum_{i=1}^{15} w_i \cdot l_i / (1220 * 2440) = 96.43\% \tag{3.12}$$

其他三个数据集的排样效果图如图 11-13 所示：

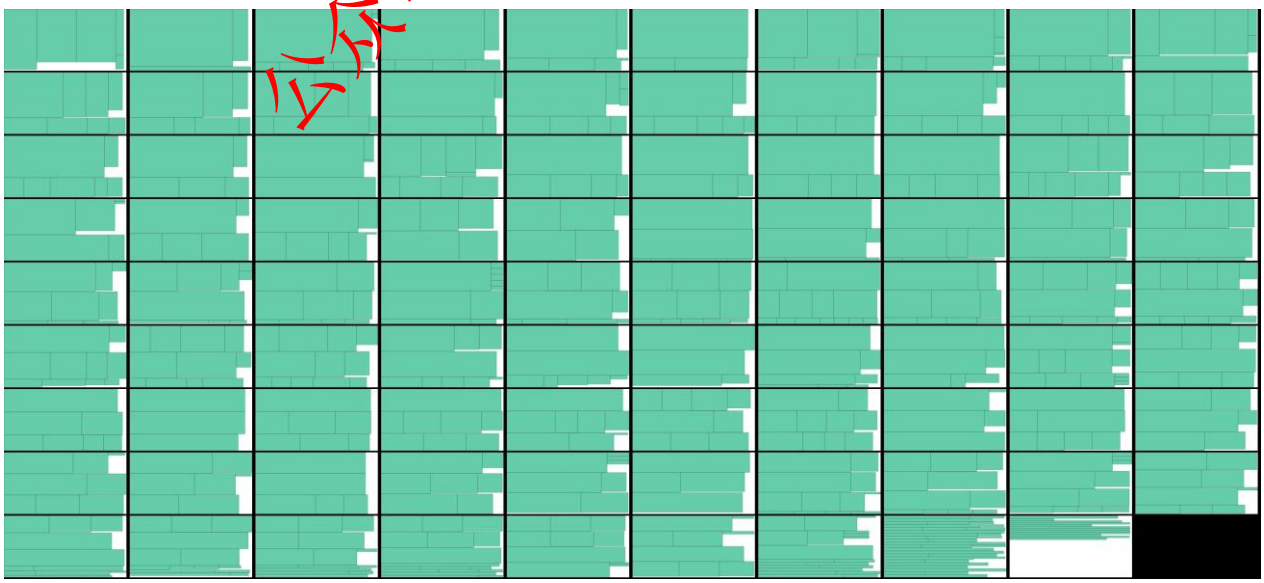

图 11 数据集 A2 排样效果

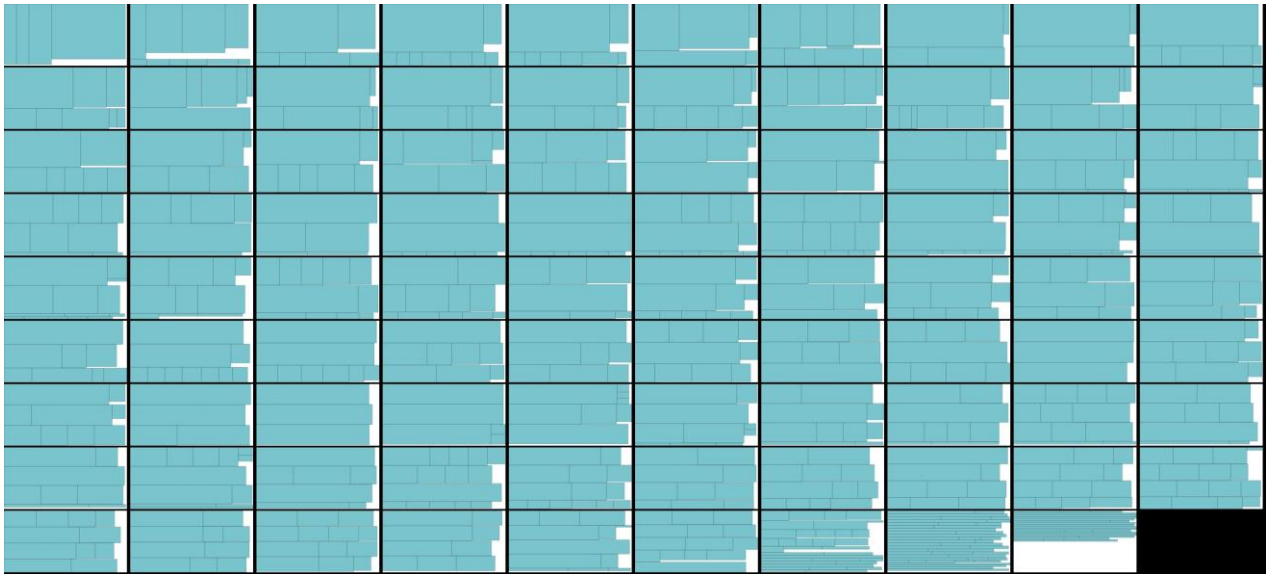

图 12 数据集 A3 排样效果

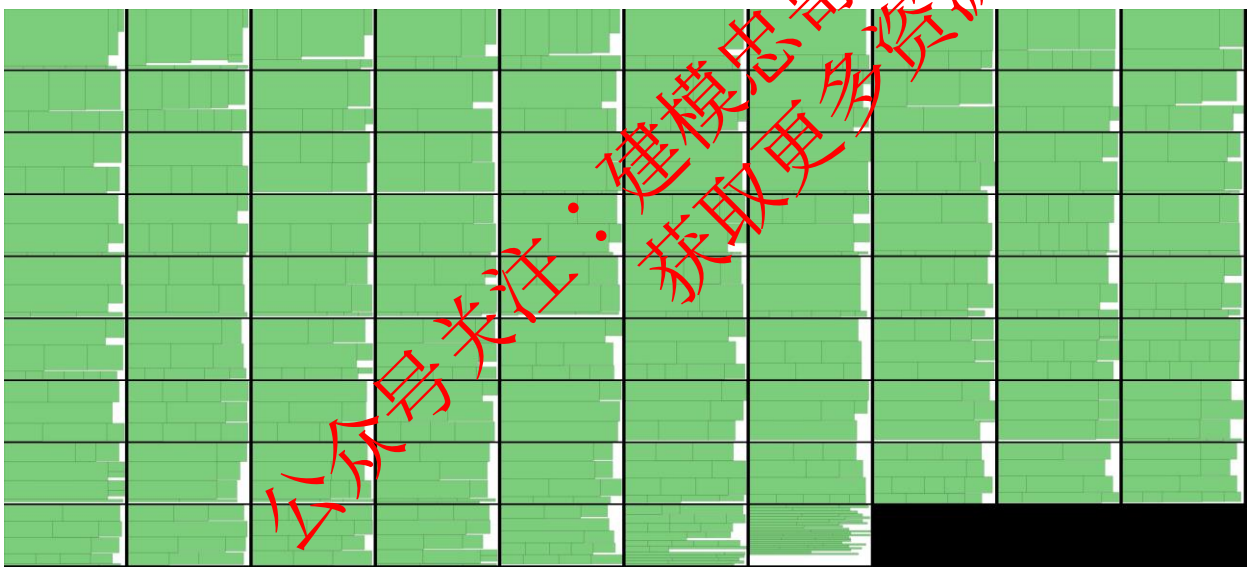

图 13 数据集 A4 排样效果

经过计算，最终数据集 A 中的 4 个子数据集所用原板材个数、利用率及排样时间如表 4 所示：

表 4 数据集 A 排样结果及程序运行时间

数据集	原板材个数	板材利用率	排样时间
A1	89	93.87%	0.1064s
A2	89	93.12%	0.0917s
A3	89	94.08%	0.1289s
A4	87	94.08%	0.1112s

板材的利用率使用饼状图的方式展示如图 14 所示：

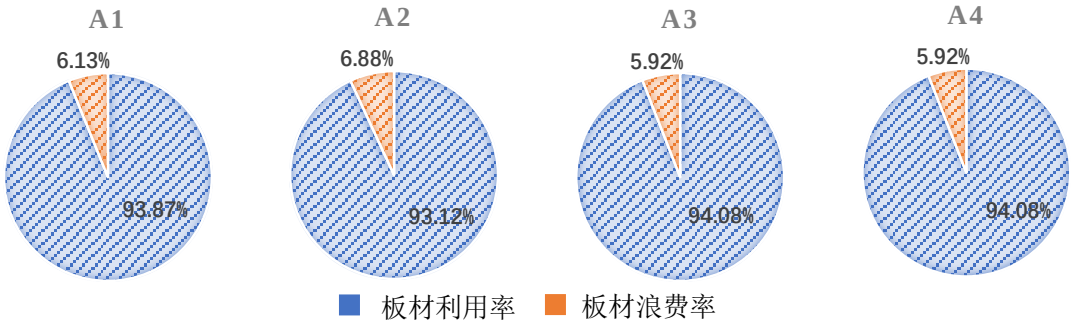

图 14 数据集 A 板材利用率

综上所述我们提出的三阶段启发式搜索树（3-SHST）算法能够有效的解决小批量、多品种的二维库存切割问题，且算法的求解效率处理不同数据集时都维持在较高水平。

四、问题二：方形件组批优化问题求解

4.1 数据分析与思考

问题二是一个组批优化问题，在问题一的基础上新增了同一订单只能出现在同一批次内、批次内 item 数量、批次内 item 面积和以及同材料排样四个约束，相当于在批次内排样优化之前增加了一个组批优化的问题。针对数据集 B，首先需要建立对数据的直观认识，统计表格 B1-B5 每个子集中的产品项数量、材料种类以及订单数量如表 5 所示：

表 5 数据集 B 各子集相关信息统计

数据集	产品项数量	材料种类	订单数量
dataB1	26811	130	546
dataB2	17952	146	403
dataB3	18028	166	410
dataB4	18526	142	381
dataB5	27901	132	604

由表可以看出数据集 B 的每个子集数据量相对数据集 A 多了很多，并且每个子集中包含数百个订单，产品项的材料也不尽相同，每个子集都有 100 多种材料，问题的复杂度上升了很多。但由于题干中给出的同一订单只能出现在同一批次的限制，我们必须将订单作为一个最小规划单位，因为它是不可分割的。到这里自然可以想到与问题一作一个类比，如果暂时不考虑材料的限制，组批优化中的订单其实类似于排样优化中的 item、批次类似于待切割的原板材、新增的两个约束限制条件即批次内 item 数量、批次内 item 面积和可以类比成问题一中原板材的两个指标即长和宽，他们的关系如图 15 所示。

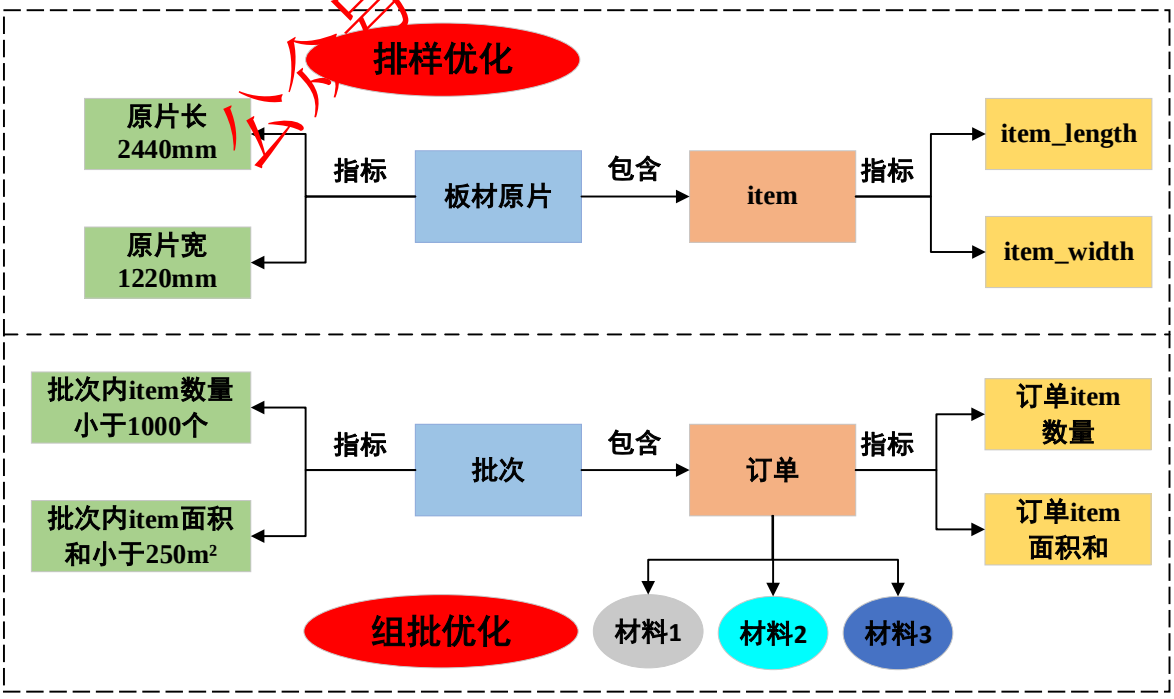

图 15 排样优化和组批优化的类比

从图中就可以很清晰的看出来导致问题一中的算法不能直接应用于问题二的原因就是因为同一订单中 item 的材料不同，而只有同材料的 item 才可以在一块原板材上排样，所以第二问的求解过程中需要综合考虑这些因素来平衡批次和利用率之间的关系。问题二求解思路如图 16 所示。

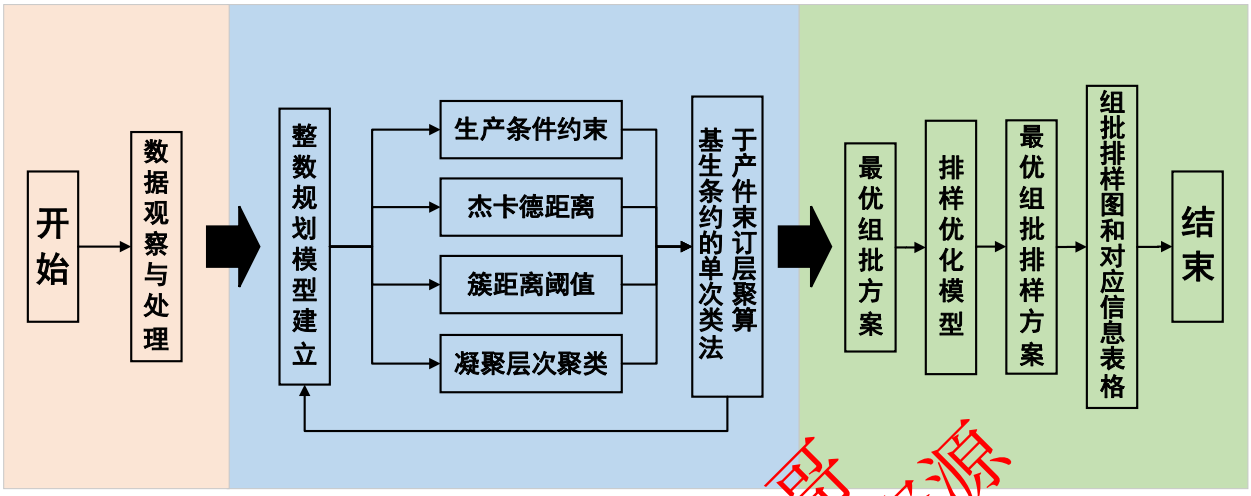

图 16 问题二求解思路

4.2 组批优化问题模型建立与求解

4. 2. 1 组批问题整数规划模型建立

针对问题二，要求对数据集 B 中的所有订单划分批次，接着对各个批次进行独立排样，使得原片板材的使用数量最少。在问题一约束的基础上，又提出了下列新的要求：

(1) 每份订单当且仅能出现在一个批次中；

(2) 每个批次中的相同材质的产品项（item）才能使用同一块板材原片进行排样；

(3)每个批次产品项（item）总数不能超过上限值 1000；

(4)每个批次产品项（item）的面积总和不能超过上限值 250 平方米。

为建立新的规划模型，定义以下变量

(1)定义 $G_{p_{ijk}}^{l}$ 为单批次中使用每种顺序编号为 p，尺寸类型为 l 的 item 的个数，i 表示此 item 在顺序编号为 i 的原片中，j 表示此 item 在顺序编号为 j 的订单中，k 表示此 item 在顺序编号为 k 的批次中。其中 $i=1,\ldots,a;\ j=1,\ldots,b;\ k=1,\ldots,c;\ p=1,\ldots,d$，$a$ 为该批次中使用的原片个数，j 为该批次中的订单个数，c 为所有的批次个数，d 为该批次中使用的 item 的个数。

(2)定义 A_i^l 表示顺序编号为 i，尺寸编号为 l 的 item 的面积。

由于问题二的约束基于问题一并与之相容，基于所述约束条件，本文考虑在问题一模型的基础上对问题二进行建模。模型如下：

（一）决策变量：

(1) $q_{k_a}^{l} \in \{0,1\}$：当且仅当顺序编号为 k 的 stack 可以被顺序编号为 a，尺寸类型为 l 的

stripe 包含时 $q_{k_a}^l=1$，否则 $q_{k_a}^l=0$，其中 $k=1,\ldots,t; l=1,\ldots,d; a=1,\ldots,m$。

(2) $y_{i_a}^l\in\{0,1\}$：当且仅当顺序编号为 i 的 item 可以被顺序编号为 a，尺寸类型为 l 的 stack 包含时 $y_{i_a}^l=1$，否则 $y_{i_a}^l=0$，其中 $i=1,\ldots,t; l=1,\ldots,d; a=1,\ldots,m$。

(3) $x_{i_a j}^l\in\mathbb{Z}^+$：表示在顺序编号为 a，尺寸类型为 l 的 stack 中，包含尺寸类型为 j 的 item 的个数。此 stack 因尺寸类型为 i 的 item 的加入而产生。其中 $i=1,\ldots,t;\ j\geq i; l=1,\ldots,d; a=1,\ldots,m$。

(4) $z_{k_a i}^l\in\mathbb{Z}^+$：表示在顺序编号为 a，尺寸类型为 l 的 stripe 中，包含尺寸类型 i 的 stack 的个数，此 stripe 因尺寸类型为 k 的 stack 的加入而产生，其中 $k=1,\ldots,t; i\geq k; l=1,\ldots,d; a=1,\ldots,m$。

(5) $w_{p_{ijk}}^l\in\{0,1\}$：当且仅当尺寸编号为 l，顺序编号为 p 的 item 可以被安排进顺序编号为 i 的原片及顺序编号为 j 的订单且顺序编号为 k 的批次中时，$w_{p_{ijk}}^l=1$，否则 $w_{p_{ijk}}^l=0$，其中 $i=1,\ldots,a; j=1,\ldots,b; k=1,\ldots,c; p=1,\ldots,d$

（二）目标函数：

为了保证所使用的板材数量尽量少，我们考虑将目标函数定义为最小化所有被使用的 stripe 的和。$q_{k_a}^l$ 保证了当且仅当顺序编号为 k 的 stack 可以被顺序编号为 a，尺寸类型为 l 的 stripe 包含时该高度才能累加。

$$Z=\min\sum_{l=1}^{d}\sum_{k=1}^{t}\sum_{a=1}^{m}q_{k_a}^l L_l \tag{4.1}$$

（三）约束条件：

1.采用等式约束所有 item 均按照需求合理安排进原片，确保了每个尺寸类型为 j 的 item 都能够被安排 e_j 次，如式(4.2)所示：

$$\sum_{l=1}^{d}\sum_{a=1}^{m}\left(\sum_{i=1}^{j}x_{i_a j}^l+y_{j_a}^l\right)=e_j\qquad j=1,\ldots,t \tag{4.2}$$

2.同理，式(4.3)确保每个被安排的 stack 都在已使用的 stripe 内：

$$\sum_{a=1}^{m}\left(\sum_{k=1}^{i}z_{k_a i}^l+q_{i_a}^l\right)=\sum_{a=1}^{m}\left(y_{i_a}^l\right)\qquad i=1,\ldots,t; l=1,\ldots,d \tag{4.3}$$

3.每个 stack 在插入 stripe 时需考虑其宽度是否超出当前 stripe 的剩余宽度，式(4.4)确保每个被插入的 stack 的宽度不超过当前尺寸类型为 l 的 stripe 的宽度。

$$\sum_{a=1}^{m} w_i z_{k_a i}^l \le \left(W_l - w_k\right) q_{k_a}^l \qquad k=1,\ldots,t;\ a=1,\ldots,m;\ l=1,\ldots,d \tag{4.4}$$

4.每个 item 在插入 stack 时需考虑其高度是否超出当前 stack 的剩余高度，式(4.5)确保每个被插入的 item 的高度不超过当前尺寸类型为l的 stack 的高度。

$$\sum_{j=i}^{m} h_j x_{i_a j}^l \le \left(L_l - h_i\right) y_{i_a}^l \qquad i=1,\ldots,t;\ a=1,\ldots,m;\ l=1,\ldots,d \tag{4.5}$$

5.式(4.6)确保顺序编号为k的 stack 不可包含于多种尺寸类型的 stripe，具有唯一性。

$$\sum_{l=1}^{d} q_{i_a}^l \le 1 \qquad k=1,\ldots,t;\ a=1,\ldots,m \tag{4.6}$$

6.式(4.7)确保顺序编号为i的 item 不可包含于多种尺寸类型的 stack，具有唯一性。

$$\sum_{l=1}^{d} y_{i_a}^l \le 1 \qquad i=1,\ldots,t;\ a=1,\ldots,m \tag{4.7}$$

7.式(4.8)确保单个批次中 item 的总数不超过 1000 个

$$\sum_{j=1}^{b}\sum_{i=1}^{a} w_{p_{ijk}}^l G_{p_{ijk}}^l \le 1000 \qquad k=1,\ldots,c;\ p=1,\ldots,d \tag{4.8}$$

8.式(4.9)确保单个批次中所有 item 的面积总和不超过 $250\,m^2$

$$\sum_{j=1}^{b}\sum_{i=1}^{a} w_{p_{ijk}}^l G_{p_{ijk}}^l A_i \le 250 \qquad k=1,\ldots,c;\ p=1,\ldots,d \tag{4.9}$$

综上所述，根据问题一所建立的整数规划模型总结如下：

$$Z=\min\sum_{l=1}^{d}\sum_{k=1}^{t}\sum_{a=1}^{m}q_{k_a}^{l}L_l$$

$$s.t.\begin{cases}\sum_{l=1}^{d}\sum_{a=1}^{m}\left(\sum_{i=1}^{j}x_{i_aj}^{l}+y_{j_a}^{l}\right)=e_j & j=1,\ldots,t\\ \sum_{a=1}^{m}\left(\sum_{k=1}^{i}z_{k_ai}^{l}+q_{i_a}^{l}\right)=\sum_{a=1}^{m}\left(y_{i_a}^{l}\right) & i=1,\ldots,t;\ l=1,\ldots,d\\ \sum_{a=1}^{m}w_i z_{k_ai}^{l}\le\left(W_l-w_k\right)q_{k_a}^{l} & k=1,\ldots,t;\ a=1,\ldots,m;\ l=1,\ldots,d\\ \sum_{j=i}^{m}h_j x_{i_aj}^{l}\le\left(L_l-h_i\right)y_{i_a}^{l} & i=1,\ldots,t;\ a=1,\ldots,m;\ l=1,\ldots,d\\ \sum_{l=1}^{d}q_{i_a}^{l}\le 1 & k=1,\ldots,t;\ a=1,\ldots,m\\ \sum_{l=1}^{d}y_{i_a}^{l}\le 1 & i=1,\ldots,t;\ a=1,\ldots,m\\ \sum_{j=1}^{b}\sum_{i=1}^{a}w_{p_{ijk}}^{l}G_{p_{ijk}}^{l}\le 1000 & k=1,\ldots,c;\ p=1,\ldots,d\\ \sum_{j=1}^{b}\sum_{i=1}^{a}w_{p_{ijk}}^{l}G_{p_{ijk}}^{l}A_i\le 250 & k=1,\ldots,c;\ p=1,\ldots,d\end{cases}\tag{4.10}$$

4.2.2 基于生产条件约束的订单层次聚类算法设计

根据前文分析以及题目中给出的约束条件，综合考虑批次内生产条件约束、排样优化约束等因素，提出了基于生产条件约束的订单层次聚类算法来求解上面提出的整数规划模型，该算法可以保证将同材质、满足批次内生产条件约束的订单尽量放在同一批次，从而达到提高生产效率、减少原片使用的目标。

（1） 订单相似性分析

由 4.1 部分的分析可知不同材料的 item 对组批问题有很大限制，所以应该尽量将有相似材料的订单放到一个批次中，所以需要提取一个指标来衡量不同订单之间材料的相似度。这里本文采用杰卡德相似性系数（Jaccard similarity coefficient,JSC）来评估两个订单之间 item 材质的相近程度。相应的，使用杰卡德距离（Jaccard Distance，JD）来评估两个订单在材质上的区分度。关于它们具体的定义如下文所述。

①杰卡德相似系数：两个订单 A、B 中 item 材料种类的交集在两者材料种类的并集中所占的比例定义为两个订单集合的杰卡德相似系数，用符号 $J\left(A,B\right)$ 表示。

$$J\left(A,B\right)=\frac{|A\cap B|}{|A\cup B|}\tag{4.11}$$

②杰卡德距离：两个订单中不同 item 材料种类的交集与数据集中所有材料种类的比值定义为杰卡德距离。用符号 $J_\delta(A,B)$ 表示。

$$J_\delta\left(A,B\right)=1-J(A,B)=\frac{|A\cup B|-|A\cap B|}{|A\cup B|}\tag{4.12}$$

③订单簇之间计算公式：在层次聚类算法中计算聚类间距离的算法有很多种，本文采用类平均距离法来计算各订单簇之间的距离，即计算两个簇之间所有订单的相互距离，将所有距离的均值作为两个订单簇之间的距离，计算公式如下：

$$D_{HK}=\sqrt{\frac{1}{n_H n_K}\sum_{\substack{i\in H\\ j\in K}} d_{ij}^2} \tag{4.13}$$

其中 d_{ij}^2 表示属于 H 类的任一样本 X_i 和属于 K 类的任一样本 X_j 之间的欧式距离平方。

若 K 类由 I 类和 J 类合并而来，则其递推公式为：

$$D_{HK}=\sqrt{\frac{n_I}{n_I+n_J}D_{HI}^2+\frac{n_J}{n_I+n_J}D_{HJ}^2} \tag{4.14}$$

（2） 订单凝聚层次聚类算法

层次聚类（Hierarchical Clustering）通过对数据集在不同层次进行划分，从而形成树形的聚类结构。又可以按创建订单树时是采用“自底向上”还是“自顶向下”将层次聚类法分为凝聚（agglomerative）层次聚类和分裂（divsive）层次聚类[]。本文采用凝聚层次聚类算法，开始时将每个单独的订单作为一个簇，将相距最近的簇进行合并然后反复对给过程进行迭代，直到没有可以合并的簇为止，通过这种方式创建一棵具有层次的嵌套聚类树。

基于生产条件约束的凝聚层次聚类算法流程如下：

Step1:初始状态下的 K 订单自成一类，即有 K 个订单簇：$C_1(0)$，$C_2(0)$，…，$C_n(0)$，通过计算各订单簇之间的距离得到一个 $K*K$ 维距离矩阵，其中“0”表示初始状态。

Step2:计算各订单簇之间的类平均距离矩阵 $D(n)$，找出 $D(n)$ 中最小距离的两个订单簇 $C_x(n)$ 与 $C_y(n)$，其中 n 表示聚类合并的迭代次数。

Step3:将 $C_x(n)$ 与 $C_y(n)$ 这两个订单簇尝试合并成新的订单簇 $C_0(n)$，判断合成后的订单簇 $C_0(n)$ 是否满足内部 item 数量不超过 1000 以及 item 面积和不超过 250m² 的要求，如果不满足则返回步骤二，找出 $D(n)$ 中次小的距离对应的订单簇进行合并检验；如果满足约束条件则进行合并，并由此建立新的订单簇：$C_1(n+1)$，$C_2(n+1)$，…

Step4:计算合并之后的新订单簇之间的距离，得 $D(n+1)$。

Step5:返回步骤二重读计算并合并。

结束条件：1）设置订单簇间距离阈值 T，当 $D(n)$ 的最小分量超过给定值 T 时算法停止；2）所有订单簇合并后都不满足内部 item 数量不超过 1000 以及 item 面积和不超过 250mm² 的要求时直接停止合并，程序结束。以上步骤用流程图表示如下图所示，图中生产约束条件指代：批次内部 item 数量不超过 1000 且 item 面积和不超过 250m²。

以上步骤绘制成算法流程图如图 17 所示：

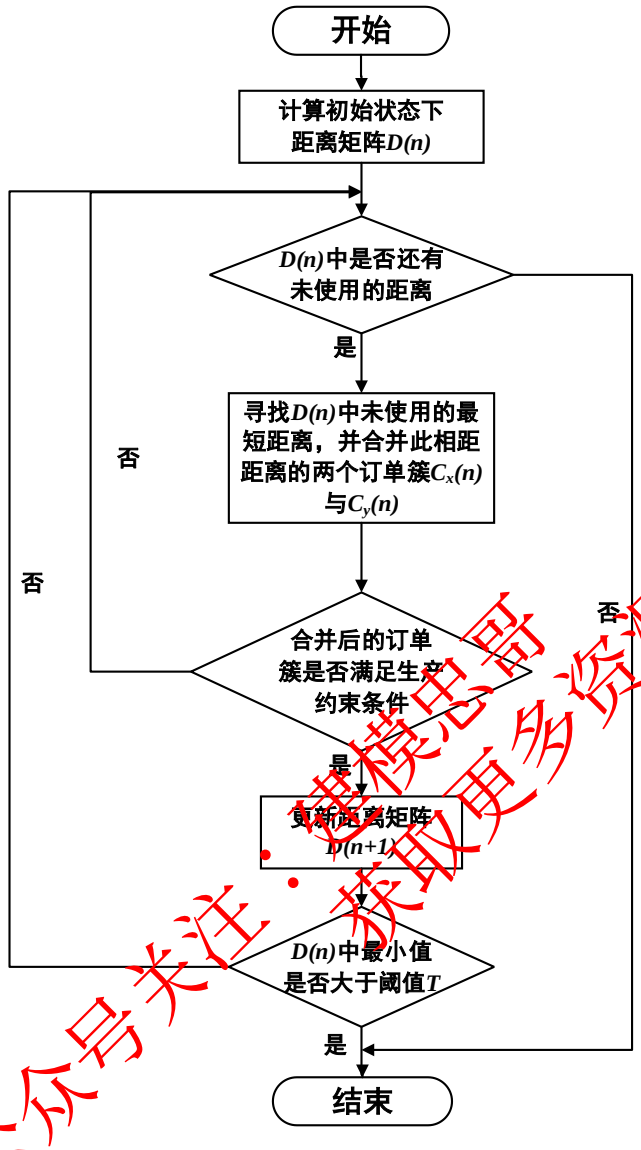

图 17 组批优化问题算法流程图

4.2.3 计算结果

论文算法使用 python3.8 实现，运行在 12 核、睿频 4.5GHz 的 Inter 处理器 i5-12500H 上，最终运行求解出的组批排样结果以数据集 B1 为例，一共组成批次 55 批，使用板材原片 3726 块，利用率达 79.93%，聚类程序运行时间 32.2528s，排样程序运行时间 1.1882s，整体组批排样效果图如图 18 所示：

图 18 数据集 B1 组批排样效果图

为证明程序的正确性，以第一批次（对应图中 B1_0）为例，其批次内排样效果图如图 19 所示，对应输出信息表格 sum_order_B1 的 1-799 行（表格太长这里不单独列出），使用原板材 95 块，图表的信息完全相符。从图中可以看出有部分原片没有得到充分利用，这里可能是与其同材料产品项比较稀少，或同材料产品项所在订单与本批次组批不满足生产约束条件所致故没有一起组批所致。

图 19 B1_0 批次内板材排样效果图

其他四个数据集的组批排样效果图分别如图 20-23 所示：

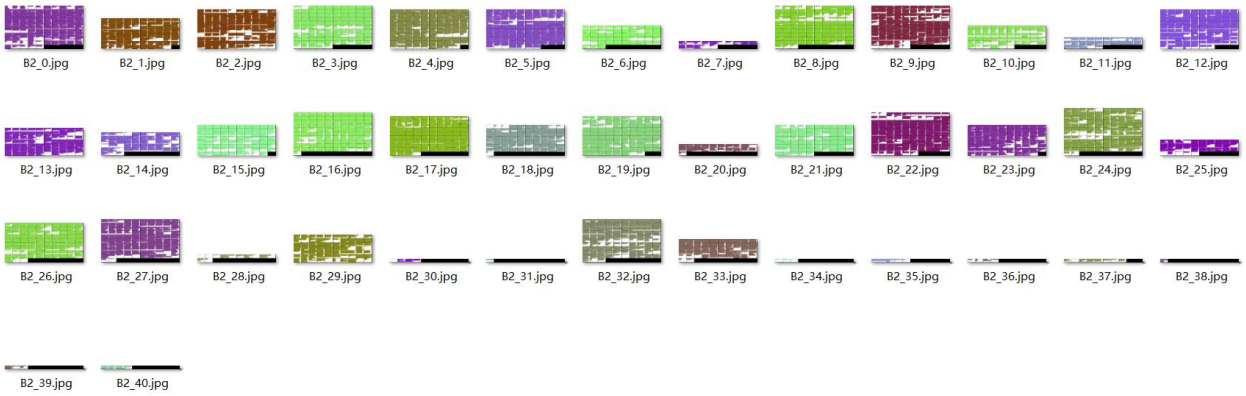

图 20 数据集 B2 组批排样效果图

图 21 数据集 B3 组批排样效果图

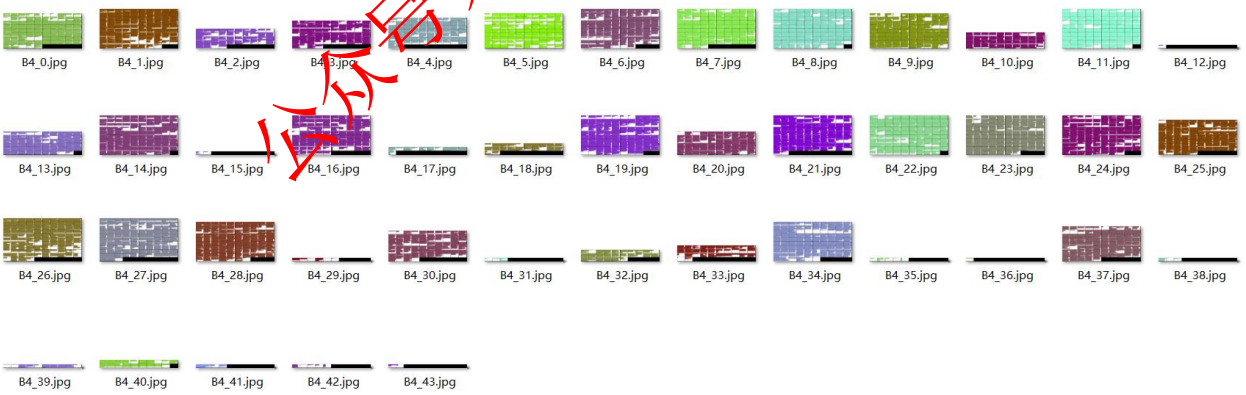

图 22 数据集 B4 组批排样效果图

图 23 数据集 B5 组批排样效果图

最终对数据集 B 的组批排样优化结果及程序运行时间如表 6 所示，可以看出排样优化程序在处理激增的数据时仍保持着较高的求解效率，主要的求解时间花费在聚类算法上。同时板材的利用率基本维持在 77%-80%区间内，保持了一个较为良好的水平。

表 6 组批优化结果和程序运行时间

数据集	批次个数	板材个数	板材利用率	聚类时间	排样时间	总时间
B1	55	3726	79.93%	32.2528s	1.1882	33.4410
B2	41	2498	77.14%	12.4876s	0.7751	13.2627
B3	51	2497	77.44%	13.0631s	0.7286	13.7917
B4	44	2565	79.15%	10.5335s	0.7741	11.3077
B5	70	4004	77.17%	45.8286s	1.2365	47.0652

组批排样优化模型最终对各数据集板材利用率情况如图 24 所示：

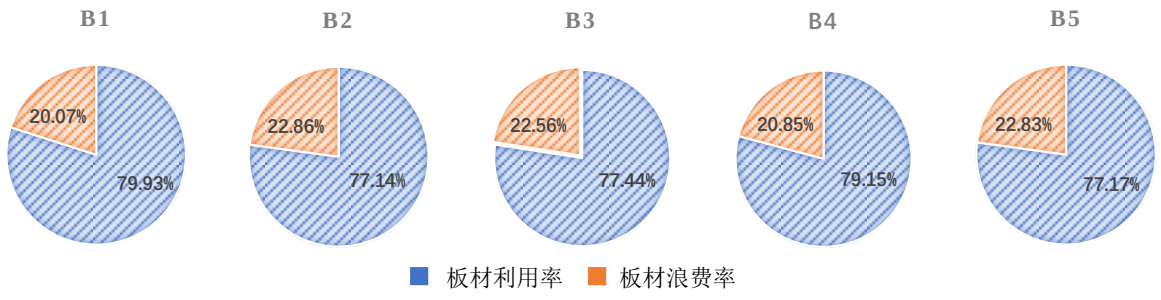

图 24 组批排样优化模型最终板材利用率

五、模型评价

5.1 模型的优点及创新

(1) 本文针对方形块的订单组批与排样优化问题进行研究，分别建立了对应的整数规划模型。多组实验数据的实验结果表明，本文建立的模型以及对应的求解算法能准确快速的解决方形快排样和组批优化问题， 有效地提高生产中的原材料利用率和生产效率。

(2) 针对问题一，相对于通过直接求解整数规划模型得到模型的精确解，本文设计的 3-SHST 算法巧妙地解决了整数规划模型中的决策变量数目爆炸问题，基于启发式的 3-SHST 算法能在很短的时间内得到方形快排样的有效解，算法的高效的求解速度弥补了算法在解质量上相对于精确解的略微不足。

(3) 针对问题二，限制条件的增加进一步导致直接求解问题的困难，因此本文将问题二的求解分开成订单组批和排版优化两个子问题，对于两个子问题来分开求解。算法的求解速度以及高效的解质量验证了分开求解的合理性以及本文设计的 3-SHST 算法和订单层次聚类算法的有效性。

(4) 本文设计的模型全面，便于理解。所设计的启发式算法降低了直接求解线性规划模型的难度，提高了本文所提方法的实用性，提高了在实际生产中使用的可能性。

5.2 模型的缺点及改进

(1) 针对问题一，本文所设计的 3-SHST 算法只能找到局部的最优解，并不能找到全局最优解。3-SHST 本质上是基于构造的算法，在搜索解空间时缺乏随机性，后续可将基于 3-SHST 算法构造的解作为初始解，再通过邻域搜索等方式进一步探索解空间来提高解的质量。

(2) 针对问题一，本文在设计 3-SHST 算法时因时间有限未考虑产品项的旋转，后续可考虑加入产品项的旋转予以改进。

(3) 针对问题二，本文在定义层次聚类算法的方法只采用了基于订单材料种类的杰卡德距离，该距离可能并不能有效的表征两个订单之间的区分度，后续可尝试对比多种距离选取最优。

(4) 针对问题二，本文将批次划分和排样分开优化，对模型进行了简化处理。本文采用的层次聚类算法也只是间接地去尽可能提高原料利用率，两个求解算法的内在的不一致性和孤立会导致的最终原料利用率的不足。后续可尝试协同优化等方式，提高原料利用率。

参考文献

[1] Elsa Silva, Filipe Alvelos, J.M. ValéRio De Carvalho. An integer programming model for two- and three-stage two-dimensional cutting stock problems[J]. European Journal of Operational Research, 2010, 205(3): 699-708.

[2] Puchinger J, Raidl G R. Models and algorithms for three-stage two-dimensional bin packing[J]. European Journal of Operational Research, 2007, 183(3): 1304-1327.

[3] Cui Y D, Huang B X. Reducing the number of cuts in generating three-staged cutting patterns[J]. European Journal of Operational Research, 2012, 218: 358-365.

[4] Elsa Silva, José Fernando Oliveira, Tiago Silveira, et al. The Floating-Cuts model: a general and flexible mixed-integerprogramming model for non-guillotine and guillotine rectangularcutting problems[J]. Omega, 2022, 114(4): 102-133.

[5] Lodi A, Monaci M. Integer linear programming models for 2-stagedtwo-dimensional knapsack problems[J]. Mathematical Programming, 2003, 94(2-3): 257-278.

[6] 邓俊. 机组组合混合整数线性规划模型的研究与改进[D]. 南宁: 广西大学. 2015.

[7] Frederico Dusberger, ĞUnther R. Raidl. Solving the 3-Staged 2-Dimensional Cutting Stock Problem by Dynamic Programming and Variable Neighborhood Search[J]. Electronic Notes in Discrete Mathematics, 2015, 47: 133-140.

[8] 李佳. 基于机器学习的多孔介质渗透率预测研究[D]. 杭州: 浙江大学. 2019.

附录

程序和数据结构

问题一：

Item 类：包括数据属性产品项长度 length、产品项宽度 width、产品项唯一表示产品 id 和产品项需求量，在本问题中产品项的需求量都为 1。

```
class Item:
    def __init__(self, length, width ,id, demand = 1):
        self.length = length # 产品长度
        self.width = width # 产品宽度
        self.id = id # 产品 ID
        self.demand = demand # 产品需求量
    def __str__(self):
        return "length: {}, width: {}, id: {}".format(self.length, self.width, self.id)
```

TreeNode 类：正文中提到的切割树节点数据结构，包括数据属性节点长度 length、节点的宽度 width、节点所处树的深度 depth（根节点对应的级别为-1），节点长度或宽度和 sum（该属性为判断是否有空间插入一个新节点）、节点的产品项信息（只有叶子节点有该属性，该属性为在完全生成切割树之后通过树构造具体的排样方案）和节点的子节点列表 childs；方法 insert_node 实现正文中描述的三阶段启发式搜索树算法，通过递归实现树的后序遍历，并对处于树不同位置的节点进行不同处理方式，例如如果当前处理的节点位于的级别是 2 时，即 stack 级别，程序通过判断加入产品项的宽度之和是否小于 stack 的宽度以及产品项的长度是否等于 stack 长度来判断是否有多余的空间。方法 add_child 加入新的子节点。

```
class TreeNode():
    def __init__(self, length, width, depth = 0, sum = 0, item = None) -> None:
        self.length = length # 节点长度信息
        self.width = width # 节点宽度信息
        self.depth = depth # 节点所处树的位置，root 为-1
        self.sum = sum # 节点长度或宽度和 依据所处位置
        self.item = item # 只有叶子节点有该属性，表示处于叶子节点处的实际 item
        self.childs = list()

     def insert_node(self, node):
         for child in self.childs:
             if child.insert_node(node):
                 return True
         if self.depth == 3:
             return False
```

```
16.
17.         # 当节点深度为 2 的时候，子节点的切法是横着切，只在当前节点下方加一个新节点
18.         elif self.depth == 2:
19.             cur_pos = self.childs[len(self.childs) - 1].sum + node.width
20.             if  cur_pos <= self.width and node.length == self.length:  # 查看是否有空间放置(暂时不考虑旋转)
21.                 node.sum = cur_pos
22.                 node.depth = 3
23.                 self.add_child(node)
24.                 return True
25.         # 当节点深度为 1 的时候， 子节点的切法是竖着切，在当前节点下方要加两个新节点(一个 stack 和一个 item)
26.         elif self.depth == 1:
27.             cur_pos = self.childs[len(self.childs) - 1].sum + node.length
28.             if node.width <= self.width and cur_pos <= self.length:
29.                 stack = TreeNode(node.length, self.width, depth = 2 ,sum = cur_pos)
30.                 self.add_child(stack)
31.                 node.sum = node.width
32.                 node.depth = 3
33.                 stack.add_child(node)
34.                 return True
35.         # 当处理节点深度为 0，即该块是某个原件，子节点的切法是横着切，在当前节点下方要加入三个新节点(一个 strip，一个 stack 和一个 item)
36.         elif self.depth == 0:
37.             cur_pos = self.childs[len(self.childs) - 1].sum + node.width
38.             if cur_pos <= self.width:
39.                 strip = TreeNode(self.length, node.width, depth = 1, sum = cur_pos)
40.                 self.add_child(strip)
41.                 stack = TreeNode(node.length, strip.width, depth = 2, sum = node.length)
42.                 strip.add_child(stack)
43.                 node.sum = node.width
44.                 node.depth = 3
45.                 stack.add_child(node)
46.                 return True
47.         # 当处理节点深度为-1 时，即当前无原件可以放置当前块，则要拿一块新原件放置该块，在根节点下方要加入四个新节点(一个 sheet， 一个 strip，一个 stack 和一个 item)
48.         else:
49.             sheet = TreeNode(self.length, self.width, depth = 0)
50.             self.add_child(sheet)
51.             strip = TreeNode(sheet.length, node.width, depth = 1, sum = node.width)
52.             sheet.add_child(strip)
53.             stack = TreeNode(node.length, strip.width, depth = 2, sum = node.length)
54.             strip.add_child(stack)
```

```
55.             node.sum = node.width
56.             node.depth = 3
57.             stack.add_child(node)
58.             return True
59.
60.         return False
61.
62.     def add_child(self, node):
63.         self.childs.append(node)
64.         # sorted(self.childs)
65.
66.     def __str__(self):
67.         return "length: {}, width: {}, depth: {}".format(self.length, self.width, self
.depth
```

可视化排样方案程序：

```
1. from PIL import Image, ImageDraw
2. from cut_tree_node import Item, TreeNode
3.
4. def visualize_sheet(node, save_file, fill = (135, 206, 235), outline = "black", width =
 2):
5.     # 可视化图，返回方形件被占面积以及方形件中所有被切割元件信息
6.     # fill，outline，width 为矩形的填充，边框等信息
7.     # 深度优先搜索
8.     def dfs_node(node):
9.         info = {}
10.         node_index = 0
11.         if node.depth == 2:
12.             node_index = node.sum - node.length
13.         elif node.depth == 1 or node.depth == 3:
14.             node_index = node.sum - node.width
15.         info[node_index] = list()
16.         for child in node.childs:
17.             info[node_index].append(dfs_node(child))
18.
19.         if node.depth == 3:
20.             return {"y": node_index, "item": node.item}
21.         return info
22.
23.     img = Image.new('RGB', (node.length, node.width), (255, 255, 255))
24.     draw = ImageDraw.Draw(img)
25.     info = dfs_node(node)[0] # 通过深度优先搜索遍历树来得到被切割块所处 x，y 坐标
26.
27.     # 返回值
28.     sum_area = 0 # 当前方形件被占面积
```

```
    item_infos = [] # 当前方形件中被切割块的信息，包括 x,y 坐标，块长度宽度，

    # 循环三阶段得到切割块所处的 strip, stack, 可修改的更清晰
    for strips in info:
        for y1, stacks in strips.items():
            for stack in stacks:
                for x, items in stack.items():
                    for item_info in items:
                        item_y = item_info['y'] + y1
                        item_x = x
                        item = item_info["item"]
                        draw.rectangle((item_x, item_y, item.length + item_x, item.width + item_y), fill, outline, width)

                        sum_area += item.length * item.width
                        item_infos.append({"sheet_index": save_file.split("/")[-1].split(".")[0], "item_id": item.id, "item_x_coordinate": item_x, "item_y_coordinate": item_y, "item_x_length": item.length, "item_y_length": item.width})
    img.save(save_file, "JPEG")
```

主程序：

```
import os
import csv
import time

import pandas as pd

from visualize_sheet import visualize_sheet
from cut_tree_node import Item, TreeNode


if __name__ == "__main__":
    src_dir = "E:/B/datasetA/"
    dest_dir = "E:/cutting_stock/"
    file = ["dataA{}.csv".format(i) for i in range(1, 5)]
    dataframe = []

    # 读取数据
    for index, f in enumerate(file):
        dataframe.append(pd.read_csv(os.path.join(src_dir, f)))

    items = list()
    dataset_index = 3

    for idx, row in dataframe[dataset_index].iterrows():
```

```
25.         items.append(Item(row["item_length"], row["item_width"], row["item_id"]))
26.
27.     # 材料排样
28.     start = time.time()  # 计算时间
29.     items = sorted(items, key=lambda d: d.width, reverse=True) # 将被切割元件按照宽度非升序排列
30.
31.     root = TreeNode(2440, 1220, depth=-1)
32.     for item in items:
33.         tree_node = TreeNode(item.length, item.width, item=item)
34.         root.insert_node(tree_node)
35.     end = time.time() # 排样结束时间
36.     print("排样时间:", end - start)
37.     print("板材个数: ", len(root.childs))
38.     sum_area = 0
39.     items_info = []
40.
41.     # 统计和可视化排样结果
42.     for index, child in enumerate(root.childs):
43.         area, item = visualize_sheet(child, "{}/solutions{}/{}.jpeg".format(dest_dir, dataset_index + 1, index))
44.         sum_area += area
45.         items_info.extend(item)
46.
47.
48.     items_info = sorted(items_info, key = lambda items: int(items["sheet_index"]))
49.     print("板材利用率: ", sum_area/(len(root.childs) * 2440 * 1220))
50.
51.     # 将结果写入 CSV 文件中
52.     with open("{}/solutions{}/cut_program.csv".format(dest_dir, dataset_index + 1), "w", newline="") as csvfile:
53.         writer = csv.writer(csvfile)
54.         # 写入标题行
55.         writer.writerow(list(items_info[0].keys()))
56.         # 写入数据
57.         for item in items_info:
58.             writer.writerow(list(item.values()))
```

问题二：

TreeNode 类和可视化排样方案代码与问题一基本类似，在此不再赘述。

Item 类：相比问题一加上数据属性产品项材料，产品项所属订单。

```
1. class Item:
2.     def __init__(self, id: int, length: float, width: float , material: str, order: str , demand = 1):
```

```
        self.id = id
        self.length = length
        self.width = width
        self.material = material
        self.order = order
        self.demand = demand

    def __str__(self):
        return "id: {}, length: {}, width: {}, material: {}, order: {}".format(self.id, self.length, self.width, self.material, self.order)
```

Order 类：订单数据结构，包括数据属性产品项列表 items，订单材料集合 material，订单中产品项面积总和 area 和订单中产品项数目总和 num；方法 jacard_distance 计算两个订单的杰卡德距离。

```
class Order():
    def __init__(self, id: str):
        self.id = id
        self.items = list() # 订单中的 item 列表
        self.material = set() # 订单中的材料集合
        self.area = 0 # 订单中产品项面积总和
        self.num = 0 # 订单中产品项数目总和

    # 根据 item 的列表初始化订单
    def initialize(self, items : list):
        self.items.extend(items)
        self.num += len(items)
        for item in items:
            self.area += item.length * item.width
            self.material.add(item.material)

    # 两个订单的杰卡德距离（不考虑产品项的数目）
    def jacard_distance(self, other):
        # 只考虑订单材质的个数
        inter_material = self.material & other.material
        collec_material = self.material | other.material
        return (len(collec_material) - len(inter_material)) / len(collec_material)


    def __str__(self):
        return "id: {}, material: {}, area: {}, num: {}".format(self.id, str(self.material), self.area, self.num)
```

聚类订单算法：算法中聚类的距离阈值默认为 0.8

```
from dis import dis
import math
```

```
3.
4. def cluster_order(orders, num_restriction, area_restriction, distance_threshold = 0.8):

5.     # orders 是所有 order
6.     cluster = [] # order 的聚类信息
7.     inial_distance = {}
8.     for i in range(len(orders)):
9.         cluster.append([orders[i]])
10.         inial_distance[orders[i].id] = {}
11.         for j in range(len(orders)):
12.             inial_distance[orders[i].id][orders[j].id] = orders[i].jacard_distance(ord
ers[j])
13.
14.     flag = True
15.     while flag:
16.         distance = cal_distance(cluster, inial_distance=inial_distance)
17.         distance = sorted(distance, key = lambda d: d[1])
18.         for pair in distance:
19.             if pair[1] > distance_threshold:
20.                 flag = False
21.                 break
22.             i, j = (int(num) for num in pair[0].split(","))
23.             # 判断第 i 和第 j 个是否可以聚类
24.             cur_num = 0
25.             cur_area = 0
26.             for order in (cluster[i] + cluster[j]):
27.                 cur_num += order.num
28.                 cur_area += order.area
29.
30.             if cur_num <= num_restriction and cur_area <= area_restriction:
31.                 # 聚类
32.                 cluster[i].extend(cluster[j])
33.                 cluster.remove(cluster[j])
34.                 break
35.             else:
36.                 # 无法聚类，则考虑下一个
37.                 continue
38.     return cluster
39.
40.
41.
42. def cal_distance(cluster, inial_distance):
43.     # 计算当前集群之间的距离，通过类平均距离法计算
44.     # 返回距离的列表，元素是元组，第一个以逗号分隔的序号(i, j)，第二个是距离
```

```
    ret_distance = []
    for i in range(len(cluster)):
        orderi_id = [order.id for order in cluster[i]]
        for j in range(i + 1, len(cluster)):
            orderj_id = [order.id for order in cluster[j]]

            distance = 0
            for id_i in orderi_id:
                for id_j in orderj_id:
                    distance += math.pow(inial_distance[id_i][id_j], 2)
            ret_distance.append(("{},{}".format(i, j), math.sqrt(distance/(len(orderi_
id) * len(orderj_id)))))

    return ret_distance
```

主程序：

```
import pandas as pd

import os
import csv
import time
import random

from order import Order, Item
from tree_node import TreeNode
from cluster_order import cluster_order
from visualize_sheet import visualize_sheet


if __name__ == "__main__":

    src_dir = "E:/datasetB/"
    dest_dir = "E:/order_group/"
    file = ["dataB{}.csv".format(i) for i in range(1, 6)]
    dataframe = []

    # 读取数据
    for index, f in enumerate(file):
        dataframe.append(pd.read_csv(os.path.join(src_dir, f)))
    items = []


    dataset_index = 4
    for idx, row in dataframe[dataset_index].iterrows():
```

```
30.         items.append(Item(row["item_id"], row["item_length"], row["item_width"], row["
item_material"], row["item_order"]))
31.
32.     items = sorted(items, key = lambda item: item.order) # 将原件按订单排序
33.
34.     # 将原件按订单分类
35.     item_order = []
36.     i, j = 0, 0
37.
38.     while i < len(items) and j < len(items):
39.         if items[i].order == items[j].order:
40.             j += 1
41.         else:
42.             item_order.append(items[i:j])
43.             i = j
44.     item_order.append(items[i:j])
45.
46.     orders = []
47.     for item_collection in item_order:
48.         order = Order(item_collection[0].order)
49.         order.initialize(item_collection)
50.         orders.append(order)
51.
52.     start = time.time()  # 计算时间
53.     cluster = cluster_order(orders, 1000, 250000000)  # 聚类订单
54.     cluster_time = time.time() # 计算集群运算时间
55.     print('集群运算时间: ',cluster_time - start)
56.
57.     sum_area = 0
58.     items_info = []
59.     sheet_index = 0
60.
61.     # 材料排样
62.     start = time.time()  # 计算时间
63.     for batch_index, orders in enumerate(cluster):
64.         blue = int(random.random() * 255)
65.         green = int(random.random() * 255)
66.
67.         materials = set()
68.         for order in orders:
69.             materials = materials | order.material
70.         for material in materials:
71.             root = TreeNode(2440, 1220, depth=-1)
72.
```

```
73.             # 取得对应 material 的产品项
74.             corresponding_items = []
75.             for order in orders:
76.                 for item in order.items:
77.                     if item.material == material:
78.                         corresponding_items.append(item)
79.
80.             # 排样对应材料的产品项
81.             corresponding_items = sorted(corresponding_items, key=lambda d: d.width, r
everse=True) # 将被切割元件按照宽度非升序排列
82.             for item in corresponding_items:
83.                 tree_node = TreeNode(item.length, item.width, item=item)
84.                 root.insert_node(tree_node)
85.
86.             # 可视化结果(计算运行时间时须注释)
87.             for child in root.childs:
88.                 area, item = visualize_sheet(child, "{}/solutions{}/{}.jpeg".format(de
st_dir, dataset_index + 1, sheet_index), batch_index
89.                                              , fill = (135, green, blue))
90.                 sheet_index += 1
91.                 sum_area += area
92.                 items_info.extend(item)
93.
94.     end = time.time() # 计算排样耗费时间
95.     print('排样运算时间：',end - start)
96.
97.     # 记录数据
98.     print("批次个数: ", len(cluster))
99.     print("板材个数: ", sheet_index)
100.        print("板材利用率: ", sum_area / (sheet_index * 2440 * 1220))
101.
102.        items_info = sorted(items_info, key = lambda item: (item["batch_index"], int(
item["sheet_index"])))
103.        with open("{}/solutions{}/sum_order.csv".format(dest_dir, dataset_index + 1),
 "w", newline="") as csvfile:
104.            writer = csv.writer(csvfile)
105.            # 写入标题行
106.            writer.writerow(list(items_info[0].keys()))
107.            # 写入数据
108.            for item in items_info:
109.                writer.writerow(list(item.values()))
```