**引用**:李车翔,赵嘉欣,侯亚群,等. 考虑必经工序的混合流水车间调度的迭代贪婪算法研究[J]. 聊城大学学报(自然科学版),2025,38(3):346-361.

**Citation**: LI Chexiang, ZHAO Jiaxin, HOU Yaqun, et al. Research on iterated greedy algorithms for hybrid flow shop scheduling considering mandatory operations[J]. Journal of Liaocheng University (Natural Science Edition), 2025, 38(3): 346-361.

**文章编号** 1672-6634(2025)03-0346-16 **DOI** 10.19728/j.issn1672-6634.2024070013

# 考虑必经工序的混合流水车间调度的迭代贪婪算法研究

李车翔,赵嘉欣,侯亚群,郑倩,李功圣,王玉亭

(聊城大学 计算机学院,山东 聊城 252059)

**摘要** 针对混合流水车间调度问题(Hybrid Flow Shop Scheduling Problem, HFSP)展开深入研究,建立了以最小化最大完工时间为优化目标的数学模型,提出了基于必经工序的迭代贪婪算法(Mandatory Operations-based IG Algorithm, MOAIG)。首先,给出了与必经工序相关的 4 个引理;其次,设计了调度序列的图空间表示方式,并针对 HFSP 多阶段的特点,将图空间中关键路径上的必经工序进行局部搜索,提高了局部搜索效率,拓展了搜索空间;然后,为了增加破坏操作的灵活性和多样性,提出了保守跳跃破坏策略;最后,通过对 576 个典型测试算例的数值仿真以及与 3 种代表算法的统计比较,验证了所提基于必经工序的加速迭代贪婪算法的有效性和优越性。

**关键词** 混合流水车间调度;最大完工时间;图空间;保守跳跃破坏;必经工序;迭代贪婪算法

**中图分类号** TP18;TH186 **文献标志码** A **开放科学(资源服务)标识码(OSID)**

# Research on iterated greedy algorithms for hybrid flow shop scheduling considering mandatory operations

LI Chexiang, ZHAO Jiaxin, HOU Yaqun, ZHENG Qian, LI Gongsheng, WANG Yuting

(School of Computer Science, Liaocheng University, Liaocheng 252059, China)

**Abstract** In-depth research has been conducted on the Hybrid Flow Shop Scheduling Problem (HFSP), establishing a mathematical model aimed at minimizing the makespan. A Mandatory Operations-based Iterated Greedy Algorithm (MOAIG) has been proposed. First, four lemmas related to mandatory operations are presented. Next, a graph space representation for the scheduling sequence is designed. For the multi-stage characteristics of HFSP, local searches are performed on the mandatory operations along the critical path within the graph space to enhance local search efficiency and expand the search space. Then, to increase the flexibility and diversity of disruption operations, a conservative jump disruption strategy is

**收稿日期**:2024-07-17

**基金项目**:国家自然科学基金项目(61973203, 62106073);山东省自然科学基金项目(ZR2023MF022,ZR2024MF112);聊城大学光岳青年创新团队项目(LCUGYTD2022-03)资助

**通信作者**:王玉亭,男,汉族,硕士,副教授,研究方向:智能优化调度,E-mail:wangyuting@lcu-cs.com。

introduced. Finally, through numerical simulations of 576 typical test cases and statistical comparisons with three representative algorithms, the effectiveness and superiority of the proposed accelerated iterative greedy algorithm based on mandatory operations are validated.

**Keywords** hybrid flow-shop scheduling; maximum completion time; graph space; adaptive jump destruction; mandatory operations; iterated greedy algorithm

## 0 引言

为了提高生产效率，混合流水车间调度问题（Hybrid Flow Shop Scheduling Problem，HFSP）备受关注，它在传统流水线生产模式基础上，在每个阶段引入多台可选的并行处理设备，以应对多变的生产需求。以汽车涂装车间为例，其生产过程需经过预处理、喷漆、烘干、打磨等多个阶段，每个阶段存在多个处理设备，优化的目标是合理安排汽车的涂装顺序及合理选择每个阶段的处理设备，以确保生产线的高效运转。研究表明，HFSP 属于 NP-hard 问题，它的研究不仅有助于优化生产流程，提高生产效率，还可以降低成本、提高资源利用率和交货准时率。因此，HFSP 问题的研究具有重要的学术意义和工程价值[1]。

针对 HFSP，研究者们已经提出了很多智能优化方法进行求解。Nowicki 等[2]基于图中关键路径和作业块的概念采用禁忌搜索算法（Tabu Search，TS）解决大规模 HFSP 取得了出色的数值性能。Liao 等[3]提出了一种带有瓶颈启发机制的粒子群优化（Particle Swarm Optimization，PSO）算法，并在 Carlier 和 Néron[4]提供的基准问题上进行了测试，表现出了优越性能。Pan 等人[5]提出了离散人工蜂群算法（Discrete Artificial Bee Colony，DABC），用以解决 HFSP 问题，这种算法在性能上超越了传统的粒子群优化（Particle Swarm Optimization，PSO）和人工免疫系统（Artificial Immune System，AIS）算法。Li 等人[6]开发了一种混合可变邻域搜索（Hybrid Variable Neighborhood Search，HVNS）算法，该算法能够在较短的计算时间内提供令人满意的调度结果。

上述算法主要在规模为 $N!$ 的解空间内进行求解（$N$ 代表工件数量），然而，HFSP 的真正的解空间是 $(N!)^S\prod_{s=1}^{S}\binom{N+m_s-1}{m_s-1}$（$S$ 代表阶段数量，$m_s$ 代表第 $s$ 阶段的机器数量）。为了扩大解空间范围，相关研究探索了超出 $N!$ 大小的解空间，即 $(N!)^S$，$(N!)^S\prod_{s=1}^{S}\binom{N+m_s-1}{m_s-1}$ 和 $(N!)^S\prod_{s=1}^{S}\binom{N+m_s-1}{m_s-1}$。不难发现，当在不同阶段之间采用不同的规则进行作业调度时，则会扩大搜索的解空间，且增加产生高质量调度方案的概率。Paternina-Arboleda 等[7]采用了不同的作业排序规则，并将求解空间扩展为 $(N!)^S$。Vairaktarakisd 等[8]采用动态规划算法在 $(N!)^S$ 大小的解空间中探索高质量调度方案。Lin 等[9]提出了混沌增强模拟退火（Chaos-enhanced Simulated Annealing，CSA）算法，采用列表调度、置换调度和非置换调度规则对各阶段的作业序列进行调度。Fan 等[10]采用了基于析取图表示的禁忌搜索方法进行局部扰动，通过将关键路径上的工序移动到同一阶段的不同机器上，将搜索空间扩大到了 $(N!)^S\prod_{s=1}^{S}\binom{N+m_s-1}{m_s-1}$。

根据混合流水车间调度问题特性不难发现：HFSP 可能存在多个关键路径，而这些关键路径上一些工序的移动并不会影响最大完成时间。其中关键路径是指所有工序加工所需时间最长的路径，而关键工序则是该路径上包含的所有工序。当存在多条关键路径时，必经工序是每条关键路径上关键工序的交集；当只存在一条关键路径时，关键工序即为必经工序。现有文献所提算法一方面限制了调度方案的搜索空间，另一方面插入工序的选择不合理，使得计算成本大大提高。针对混合流水车间调度问题，不难发现对关键路径上必经工序进行扰动，可以不受规则约束，扩大搜索空间，而且可以减少不必要的工序插入，从而降低计算成本，提高优化调度效率。鉴于此，本文提出一种基于必经工序的加速迭代贪婪算法（Mandatory Operations-based Accelerated IG Algorithm，MOAIG），具体贡献如下：首先，针对 HFSP 特性，定义了必经工序的概念，提出了基于必经工序的局部搜索的相关引理和证明；其次，引入了基于必经工序的局部搜索，扩大了解空间大小和提高了全局搜索能力。然后，设计了一种新的保守跳跃破坏策略，以引导参与破坏的工件数目取值趋向于目标值变小的方向变化。最后，通过大量仿真实验，验证了所提算法能够减少参与插入的工序的个数，避免了工序的无效移动，节约了计算成本，提高了优化调度效率。

# 1 混合流水车间调度问题

相关符号及解释见表1,相关决策变量及相应解释见表2。要求满足以下约束条件:(1) 所有工件在各个生产阶段按相同顺序进行加工;(2) 每个阶段 $M_s \geqslant 1$ 且至少有一个阶段 $M_s > 1$;(3) 每个工件在每个阶段只能选择该阶段的一台机器进行加工;(4) 每台机器在任意时刻只能加工一个工件。

HFSP 满足的约束及假设如下[11-12]:(1) 所有工件在各个阶段的加工时间已知;(2) 所有工件和机器在0时刻都是可用的状态;(3) 两个阶段之间的缓冲区无限大;(4) 不考虑机器工件转换时间和工件在两阶段之间的运输时间;(5) 所有工件一旦被加工,不可被抢占。本文研究的目标是在每个生产阶段,需要为每个工件分配机器,并确定每台机器上加工工件的顺序,以使得最大完工时间最小。

**表1 符号及相应解释**

**Tab. 1 Symbols and corresponding interpretations**

| Symbols | Explanations | Symbols | Explanations |
|---|---|---|---|
| $\mu$ | 工件集合 | $M_{js}$ | 工件 $j$ 在第 $s$ 阶段加工的机器,$j \in \{1,\cdots,N\}$,$s \in \{1,\cdots,S\}$ |
| $N$ | 工件数量 | $m_s$ | 第 $s$ 阶段的机器数量,$s \in \{1,\cdots,S\}$ |
| $j$ | 工件编号 | $k$ | 机器编号,$k \in \{1,\cdots,m_s\}$ |
| $d$ | 破坏工件的数量 | $S_{js}$ | 工件 $j$ 在第 $s$ 阶段的开始加工时间,$j \in \{1,\cdots,N\}$,$s \in \{1,\cdots,S\}$ |
| $N^e$ | 保守跳跃破坏阶段破坏的工件数量 | $P_{js}$ | 工件 $j$ 在第 $s$ 阶段的加工时间,$j \in \{1,\cdots,N\}$,$s \in \{1,\cdots,S\}$ |
| $S$ | 阶段数量 | $C_{js}$ | 工件 $j$ 在第 $s$ 阶段的正向完工时间,$j \in \{1,\cdots,N\}$,$s \in \{1,\cdots,S\}$ |
| $s$ | 阶段编号 | $E_{js}$ | 工件 $j$ 在第 $s$ 阶段的逆向完工时间,$j \in \{1,\cdots,N\}$,$s \in \{1,\cdots,S\}$ |
| $M_s$ | 第 $s$ 阶段的可用机器集合,$s \in \{1,\cdots,S\}$ | $C_{\max}$ | 工件序列的最大完工时间 |
| $L$ | 一个很大的常数 | $o_{js}$ | 工件 $j$ 在第 $s$ 阶段的工序,$j \in \{1,\cdots,N\}$,$s \in \{1,\cdots,S\}$ |

**表2 相关决策变量及解释**

**Tab. 2 Relevant decision variables and their interpretations**

| Decision Variables | Explanations |
|---|---|
| $X_{jsk}$ | 当 $X_{jsk}=1$ 时,工件 $j$ 在第 $s$ 阶段的机器 $k$ 上加工;当 $X_{jsk}=0$ 时,工件 $j$ 不在第 $s$ 阶段的机器 $k$ 上加工,$j \in \{1,\cdots,N\}$,$s \in \{1,\cdots,S\}$,$k \in \{1,\cdots,m_s\}$ |
| $Y_{j_1j_2s}$ | 当 $Y_{j_1j_2s}=1$ 时,在第 $s$ 阶段,工件 $j_1$ 先于工件 $j_2$ 加工;当 $Y_{j_1j_2s}=0$ 时,在第 $s$ 阶段,工件 $j_2$ 先于工件 $j_1$ 加工,$j_1,j_2 \in \{1,\cdots,N\}$,$j_1 \neq j_2$,$s \in \{1,\cdots,S\}$ |

优化目标:

$$\min C_{\max}\ ; \tag{1}$$

约束条件:

$$S_{js} \geqslant 0\ \forall j \in \{1,\cdots,N\},\forall s \in \{1,\cdots,S\}\ , \tag{2}$$

$$C_{\max} \geqslant C_{js}\ \forall j \in \{1,\cdots,N\},\forall s \in \{1,\cdots,S\}\ , \tag{3}$$

$$C_{js} = S_{js} + P_{js}\ \forall j \in \{1,\cdots,N\},\forall s \in \{1,\cdots,S\}\ , \tag{4}$$

$$S_{j(s+1)} - S_{js} \geqslant P_{js}\ \forall j \in \{1,\cdots,N\},\forall s \in \{1,\cdots,(S-1)\}\ , \tag{5}$$

$$\sum_{k=1}^{m_s} X_{jsk} = 1\ \forall j \in \{1,\cdots,N\},\forall s \in \{1,\cdots,S\},\forall k \in \{1,\cdots,m_s\}\ , \tag{6}$$

$$Y_{j_1j_2s} + Y_{j_2j_1s} \leqslant 1\ \forall j_1,j_2 \in \{1,\cdots,N\},j_1 \neq j_2,\forall s \in \{1,\cdots,S\}\ , \tag{7}$$

$$S_{j_2s} - C_{j_1s} + L \times (3 - Y_{j_1j_2s} - X_{j_1sk} - X_{j_2sk}) \geqslant 0\ \forall j_1,j_2 \in \{1,\cdots,N\},j_1 \neq j_2,$$
$$\forall s \in \{1,\cdots,S\},\forall k \in \{1,\cdots,m_s\}\ , \tag{8}$$

$$Y_{j_1j_2s},Y_{j_2j_1s},X_{j_1sk},X_{j_2sk},X_{jsk} \in \{0,1\}\ \forall j,j_1,j_2 \in \{1,\cdots,N\},$$
$$j_1 \neq j_2,\forall s \in \{1,\cdots,S\},\forall k \in \{1,\cdots,m_s\}\ 。 \tag{9}$$

式(1)是最小化最大完工时间；式(2)要求所有工件在任意阶段的开始加工时间大于 0；式(3)要求最大完工时间大于所有工件在最后一个阶段的完工时间；式(4)用于计算每个工件在任意阶段的完工时间；式(5)要求每个工件必须在上一阶段加工完成后才能开始当前阶段的加工；式(6)确保每个工件在每个阶段的加工只能由该阶段的一台机器完成；式(7)和式(8)是对于同一台机器上加工的两个工件，只有在前一个工件完成加工后，后一个工件才能开始加工；式(9)表示 0-1 变量的约束。

## 2　必经工序的相关定义和引理

### 2.1　解在图空间中的表示

为了更好地描述混合流水车间调度问题，本文采用有向图来表示问题的调度序列，以 7 个工件 3 个阶段为例。7 个工件在第一阶段的加工时间分别为 5，5，3，7，3，3，5，在第二阶段的加工时间分别为 6，5，3，6，6，5，7，在第三阶段的加工时间分别为 4，3，6，3，3，3，6，调度顺序为{1，2，5，4，3，6，7}。有向图记为 $G_Y$，图中每个节点对应一个工序 $o_{js}$，如图 1 中黑色边框、红色边框和蓝色边框表示的节点。每个节点上方表示工序名称，左下方表示工序的逆向完工时间，正下方是工序的加工时间，右下方表示工序正向完工时间。黑色边框的节点表示非关键工序，红色边框和蓝色边框的节点表示关键工序，其中红色边框的节点表示必经工序。每个节点的权值等于 $o_{js}$ 的加工时间 $P_{js}$。灰色圆圈表示虚拟工序，用于表示每个阶段 $s$ 的开始和结束，为了方便计算正向完工时间和逆向完工时间，它们的加工时间均为 0。在 $G_Y$ 中，$o_{js}$ 和 $o_{j(s+1)}$ 之间存在一条弧 $\langle o_{js}, o_{j(s+1)}\rangle$，这里 $j\in\{0,\cdots,N\}$，$s\in\{1,\cdots,(S-1)\}$，表示 $o_{j(s+1)}$ 必须在 $o_{js}$ 加工完成后才能加工。对于工序 $o_{js}$ 和 $o_{j^b_s}$，这里 $j, j^b\in\{1,\cdots,N\}$，$s\in\{1,\cdots,S\}$，如果 $M_{js}=M_{j^b_s}$，且加工完 $o_{js}$ 紧接着加工 $o_{j^b_s}$，则在这两个工序之间存在一条弧 $\langle o_{js}, o_{j^b_s}\rangle$。如果 $o_{js}$ 为 $M_{js}$ 上第一个加工的工序，则在 $o_{0s}$ 和 $o_{js}$ 存在一条弧 $\langle o_{0s}, o_{js}\rangle$；如果 $o_{js}$ 为 $M_{js}$ 上最后一个加工的工序，则在 $o_{js}$ 和 $o_{(N+1)s}$ 存在一条弧 $\langle o_{js}, o_{(N+1)s}\rangle$。在 $G_Y$ 中分别构建了两种类型的双向链表，一种是同一机器上的双向链表，称为机器链表。它们是通过 $o_{js}$ 和 $o_{j^b_s}$ 之间的弧 $\langle o_{js}, o_{j^b_s}\rangle$ 连接，$o_{j^b_s}$ 为同机器 $o_{js}$ 的后继工序，如图中黑色双向实线；另一种是同一工件在不同阶段的双向链表。它们是通过 $o_{js}$ 和 $o_{j(s+1)}$ 之间的弧 $\langle o_{js}, o_{j(s+1)}\rangle$ 连接，称为工件链表，如图 1 中黑色双向虚线。通过有向图的表示，可以清晰地展示每个工序之间的关系，方便后期对工序进行任意扰动。有向图表示解的调度序列如图 1 所示，甘特图表示解的调度序列如图 2 所示。

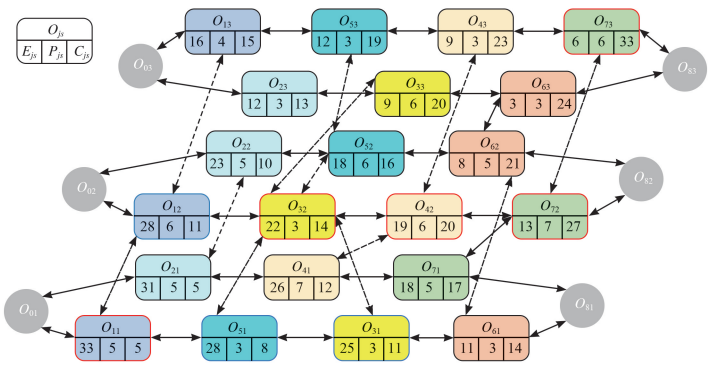



**图 1　解在图空间的表示(其中红色边框的盒子代表必经工序，蓝色边框的盒子代表关键工序)**

**Fig. 1　The representation in the graph space (where boxes with red and red borders represent mandatory and critical operations, respectively)**

### 2.2　必经工序的相关定义

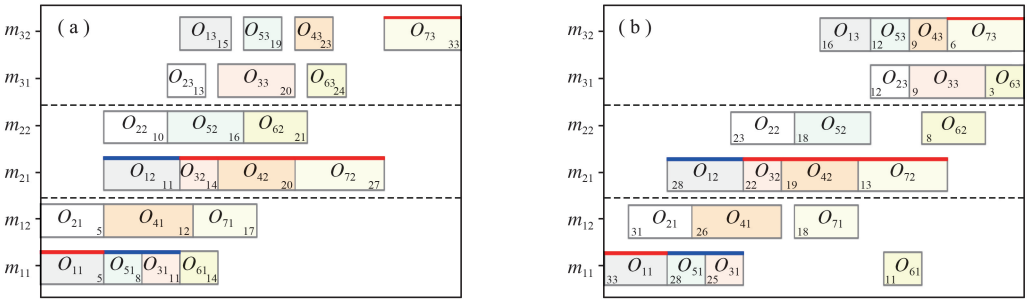
**定义 1(路径长度)**　在 $G_Y$ 中，路径 $P=\{o_{js},\cdots,o_{j^b_s b}\}$ 的长度定义为从起点 $o_{js}$ 到终点 $o_{j^b_s b}$ 所有工序加工时间的总和，记为 $L_{G_Y}(P)$。

在有向图 $G_Y$ 中，如果从 $o_{js}$ 到 $o_{j^b_s b}$ 存在 $l$ 条路径，其中最长的路径有 $T$ 条，记这 $T$ 条最长路径中的第 $t$ 条为 $P^t_{G_Y}(o_{js}, o_{j^b_s b})$，也就是说，$P^t_{G_Y}(o_{js}, o_{j^b_s b})$，$t=1,\cdots,T$，为 $G_Y$ 中 $o_{js}$ 到 $o_{j^b_s b}$ 最大长度的有序顶点的集合。这 $T$ 条最长路径的长度记为 $L_{G_Y}(o_{js}, o_{j^b_s b})$。

**定义 2($o_{js}$ 的正向完工时间)**　$G_Y$ 中 $o_{01}$ 到 $o_{js}$ 的最长路径长度定义为 $G_Y$ 中 $o_{js}$ 的正向完工时间，记为 $C_{js}(G_Y)$，即 $C_{js}(G_Y)=L_{G_Y}(o_{01}, o_{js})$。

显然，式(10)成立，

$$C_{\max}(G_Y)=L_{G_Y}(o_{01},o_{(N+1)S})。\tag{10}$$




注:(a) 正向调度甘特图;(b) 逆向调度甘特图。

**图 2 解对应的甘特图表示(其中红色边框的盒子代表必经工序,蓝色边框的盒子代表关键工序)**

**Fig. 2 The corresponding Gantt chart representation (where boxes with red and red borders represent mandatory and critical operations, respectively)**

**定义 3($o_{js}$ 的逆向完工时间)** $G_Y$ 中 $o_{js}$ 到 $o_{(N+1)S}$ 的最长路径长度定义为 $G_Y$ 中 $o_{js}$ 的逆向完工时间,记为 $E_{js}(G_Y)$,即 $E_{js}(G_Y)=L_{G_Y}(o_{js},o_{(N+1)S})$。

**定义 4(关键路径)** $G_Y$ 中从 $o_{01}$ 到 $o_{(N+1)S}$ 的每一条最长的路径 $P^t_{G_Y}(o_{01},o_{(N+1)S})$,$t$ 为路径编号,均定义为 $G_Y$ 的关键路径。显然,$G_Y$ 中任意一条关键路径的长度均等于 Makespan。

**定义 5(关键工序)** 包含在 $G_Y$ 的任意关键路径上除了 $o_{01}$ 和 $o_{(N+1)S}$ 之外的工序定义为 $G_Y$ 的关键工序。

从 $o_{01}$ 到 $o_{(N+1)S}$ 经过 $o_{js}$ 的最长路径定义为 $G_Y$ 中经过 $o_{js}$ 的关键路径,记为 $P^t_{G_Y}(o_{01},o_{js},o_{(N+1)S})$,路径的长度记为 $L_{G_Y}(o_{01},o_{js},o_{(N+1)S})$。假设 $o_{js}$ 在同机器前后的工序分别为 $o_{j^p_s}$,$o_{j^b_s}$,$C_{j^p_s}(G_Y)$ 表示在 $G_Y$ 中 $o_{j^p_s}$ 的正向完工时间,$E_{j^b_s}(G_Y)$ 表示在 $G_Y$ 中 $o_{j^b_s}$ 的逆向完工时间,则经过 $o_{js}$ 的关键路径长度如公式(11),

$$\begin{aligned}L_{G_Y}(o_{01},o_{js},o_{(N+1)S})&=\max(L_{G_Y}(o_{01},o_{j^p_s}),L_{G_Y}(o_{01},o_{j(s-1)}))+P_{js}+\max(L_{G_Y}(o_{j^b_s},o_{(N+1)S}),\\&\quad L_{G_Y}(o_{j(s+1)},o_{(N+1)S}))\\&=\max(C_{j^p_s}(G_Y),C_{j(s-1)}(G_Y))+P_{js}+\max(E_{j^b_s}(G_Y),E_{j(s+1)}(G_Y))。\end{aligned}\tag{11}$$

**定义 6(关键工序子图)** 由节点 $o_{01}$、$o_{(N+1)S}$ 和 $G_Y$ 中的关键工序节点,以及连接它们的弧组成的有向图定义为 $G_Y$ 的关键工序子图,记为 $G^S_Y$。

**定义 7(必经工序)** 在关键工序子图 $G^S_Y$ 中,如果去掉某个关键工序 $o_{js}$ 后,从 $o_{01}$ 到 $o_{(N+1)S}$ 不存在路径,则称 $o_{js}$ 为 $G_Y$ 的关键路径的必经工序(Mandatory Operations,MO)。

**定义 8(删除和最优插入)** 将 $o_{js}$ 从 $G_Y$ 中删除得到 $G^-_Y$,然后将 $o_{js}$ 重新插入到 $G^-_Y$ 中且所插入位置能得到所有插入位置最好的 Makespan,这个过程定义为删除和最优插入(Deletion and Optimal Insertion,DOI)。

**注 1** 上述定义将 $G_Y$ 换成 $G^-_Y$、$G^+_Y$ 也成立,$G^-_Y$ 表示 $G_Y$ 删除工序 $o_{js}$ 后的有向图,$G^+_Y$ 表示向 $G^-_Y$ 插入工序 $o_{js}$ 后的有向图。

### 2.3 必经工序的相关引理

**定理 1** 如果 $o_{js}$ 为关键工序,则 $C_{\max}(G_Y)=\max(C_{j^p_s}(G_Y),C_{j(s-1)}(G_Y))+P_{js}+\max(E_{j^b_s}(G_Y),E_{j(s+1)}(G_Y))$。

**证** 由公式(10)得

$$L_{G_Y}(o_{01},o_{js},o_{(N+1)S})=\max(C_{j^p_s}(G_Y),C_{j(s-1)}(G_Y))+P_{js}+\max(E_{j^b_s}(G_Y),E_{j(s+1)}(G_Y))$$

$o_{js}$ 为关键工序时有 $L_{G_Y}(o_{01},o_{js},o_{(N+1)S})=C_{\max}(G_Y)$,所以

$$C_{\max}(G_Y)=\max(C_{j^p_s}(G_Y),C_{j(s-1)}(G_Y))+P_{js}+\max(E_{j^b_s}(G_Y),E_{j(s+1)}(G_Y))。$$

**定理 2** 对必经工序进行移动可能优化最大完工时间,对非必经工序进行移动无法优化最大完工时间。

**证** 在 $G_Y$ 中删除工序 $o_{js}$ 得到 $G^-_Y$,可以分为以下两种情况。

(1) 若 $o_{js}$ 是 $G_Y$ 的必经工序,无论关键路径有一条甚至多条,删除工序 $o_{js}$ 之后,关键路径都会受到破坏,受破坏后的路径或者其他路径的长度必然小于 $L_{G_Y}(o_{01},o_{(N+1)S})$,即 $L_{G_Y^-}(o_{01},o_{(N+1)S})<L_{G_Y}(o_{01},o_{(N+1)S})$,从而 $C_{\max}(G_Y^-)<C_{\max}(G_Y)$,所以当再将工序 $o_{js}$ 插入到 $G_Y^-$,最大完工时间可能变优。

(2) 若 $o_{js}$ 不是 $G_Y$ 的必经工序,则在删除工序 $o_{js}$ 之后,至少存在一条关键路径,即 $L_{G_Y^-}(o_{01},o_{(N+1)S})=L_{G_Y}(o_{01},o_{(N+1)S})$,从而 $C_{\max}(G_Y^-)=C_{\max}(G_Y)$。

将 $o_{js}$ 插入到 $G_Y^-$ 得到 $G_Y^+$,若 $o_{js}$ 没有成为 $G_Y^+$ 的关键工序,关键路径不发生改变,则

$$C_{\max}(G_Y^+)=C_{\max}(G_Y)=C_{\max}(G_Y^-)。$$

若 $o_{js}$ 成为 $G_Y^+$ 的关键工序,关键路径发生改变且长度不会减少,则

$$C_{\max}(G_Y^+)\geqslant C_{\max}(G_Y^-)=C_{\max}(G_Y)。$$

所以当再将工序 $o_{js}$ 插入到 $G_Y^-$,最大完工时间无法优化。

**定理 3**　将 $G_Y$ 中的必经工序 $o_{js}$ 删除后,再插入到 $G_Y^-$ 中,若没有成为 $G_Y^+$ 中的关键工序,$C_{\max}(G_Y)$ 会得到优化。若成为 $G_Y^+$ 中的关键工序且 $L_{G_Y^+}(o_{01},o_{js},o_{(N+1)S})<C_{\max}(G_Y)$,$C_{\max}(G_Y)$ 也会得到优化。

**证**　在 $G_Y$ 中删除必经工序 $o_{js}$ 得到 $G_Y^-$,在 $G_Y^-$ 中插入工序 $o_{js}$ 得到 $G_Y^+$,插入可以分为以下两种情况。

(1) 若 $o_{js}$ 不是 $G_Y^+$ 中的关键工序,则 $G_Y^+$ 和 $G_Y^-$ 有相同的关键路径,所以有 $L_{G_Y^+}(o_{01},o_{(N+1)S})=L_{G_Y^-}(o_{01},o_{(N+1)S})$,

即 $C_{\max}(G_Y^+)=C_{\max}(G_Y^-)$。

根据**定理 2** 中的第一种情况有 $C_{\max}(G_Y^-)<C_{\max}(G_Y)$,从而 $C_{\max}(G_Y^+)<C_{\max}(G_Y)$。

(2) 若 $o_{js}$ 是 $G_Y^+$ 中的关键工序,则 $G_Y^+$ 的某条关键路径必然包含 $o_{js}$,所以有 $L_{G_Y^+}(o_{01},o_{(N+1)S})=L_{G_Y^+}(o_{01},o_{js},o_{(N+1)S})$,此时只需 $L_{G_Y^+}(o_{01},o_{js},o_{(N+1)S})<C_{\max}(G_Y)$ 成立,便可得到 $C_{\max}(G_Y^+)<C_{\max}(G_Y)$。

**定理 4**　对必经工序 $o_{js}$ 进行 DOI 操作后,

$$C_{\max}(G_Y^+)=\max(C_{\max}(G_Y^-),\max(C_{j^{p}_{s}}(G_Y^+),C_{j(s-1)}(G_Y^+))+P_{js}+\max(E_{j^{b}_{s}}(G_Y^+),E_{j(s+1)}(G_Y^+)))。$$

**证**　可以分为以下两种情况。

(1) 将 $o_{js}$ 插入到 $G_Y^-$ 中,若 $o_{js}$ 成为 $G_Y^+$ 的关键工序,$G_Y^+$ 中必有一条关键路径包含 $o_{js}$,即

$$L_{G_Y^+}(o_{01},o_{(N+1)S})=L_{G_Y^+}(o_{01},o_{js},o_{(N+1)S})。$$

因为 $L_{G_Y^+}(o_{01},o_{(N+1)S})\geqslant L_{G_Y^-}(o_{01},o_{(N+1)S})$,所以 $L_{G_Y^+}(o_{01},o_{(N+1)S})=\max((L_{G_Y^-}(o_{01},o_{(N+1)S}),L_{G_Y^+}(o_{01},o_{js},o_{(N+1)S}))$,

根据式(10)、(11)有 $C_{\max}(G_Y^+)=\max(C_{\max}(G_Y^-),\max(C_{j^{p}_{s}}(G_Y^+),C_{j(s-1)}(G_Y^+))+P_{js}+\max(E_{j^{b}_{s}}(G_Y^+),E_{j(s+1)}(G_Y^+)))$。

(2) 将 $o_{js}$ 插入到 $G_Y^-$ 中,若 $o_{js}$ 没有成为 $G_Y^+$ 的关键工序,$G_Y^-$ 和 $G_Y^+$ 具有完全相同的关键路径,其中 $G_Y^+$ 没有任何一条关键路径经过 $o_{js}$,即 $L_{G_Y^+}(o_{01},o_{(N+1)S})=L_{G_Y^-}(o_{01},o_{(N+1)S})$,$L_{G_Y^+}(o_{01},o_{js},o_{(N+1)S})<L_{G_Y^+}(o_{01},o_{(N+1)S})$,所以,$L_{G_Y^+}(o_{01},o_{(N+1)S})=\max(L_{G_Y^-}(o_{01},o_{(N+1)S}),L_{G_Y^+}(o_{01},o_{js},o_{(N+1)S}))$。

根据式(10)、(11)有

$$C_{\max}(G_Y^+)=\max(C_{\max}(G_Y^-),\max(C_{j^{p}_{s}}(G_Y^+),C_{j(s-1)}(G_Y^+))+P_{js}+\max(E_{j^{b}_{s}}(G_Y^+),E_{j(s+1)}(G_Y^+)))。$$

## 3　基于必经工序的加速迭代贪婪算法设计

迭代贪婪算法结构简单,参数少,能够在较少的迭代次数内找到较优解[13]。然而,在传统的迭代贪婪算法中,破坏重构策略和局部搜索策略通常基于插入操作实现,这种方式在一定程度上限制了解空间的探索。尽管局部搜索能够帮助算法更好地优化当前解,但过度依赖局部搜索容易使得解在迭代过程中陷入局部最优解,导致解的多样性降低,难以跳出搜索范围。因此,为了提高算法的全局搜索能力和解的多样性,需要引入更多元化的搜索策略或者增加随机性,以更好地探索解空间并避免陷入局部最优解的困境。本文提出了

基于必经工序的加速迭代贪婪算法 MOAIG(如算法 1 所示),通过设计保守跳跃破坏策略和基于必经工序的局部搜索策略,对必经工序的删除和插入进行优化,旨在扩大解空间的规模,减少参与局部搜索的工序数量,从而更有效地探索潜在的最优解,以提高算法的全局搜索能力。

**算法 1 MOAIG()**

01: $(\pi, C_{\max}) = NEH(N)$ %Ref. 文献[14]
02: $(\pi^{best}, C_{\max}^{best}) = (\pi, C_{\max})$, $(\pi^{old}, C_{\max}^{old}) = (\pi, C_{\max})$, $flag = true$, $r = 0$, $E = \varnothing$
03: **while time** $\leqslant \lambda \times J^3 \times S$(ms) **do**
04: $(\pi, C_{\max}^{DR}, r) = \mathrm{D\&R}(\pi, flag, E, r, C_{\max})$ %算法 2
05: $(G_Y, C_{\max}) = \mathrm{DecodeToGraph}(\pi)$ %算法 3
06: $MO = \mathrm{ObatiningMO}(G_Y, C_{\max})$ %算法 4
07: $M\_C_{\max} = \mathrm{DOI}(G_Y, MO, C_{\max})$ %算法 5
08: **if** $M\_C_{\max} < C_{\max}^{best}$ **then**
09: $(\pi^{best}, C_{\max}^{best}) = (\pi, M\_C_{\max})$
10: **end if**
11: **if** $M\_C_{\max} \leqslant C_{\max}^{old}$ **then**
12: $(\pi^{old}, C_{\max}^{old}) = (\pi, M\_C_{\max})$, $flag = true$
13: **else**
14: $flag = false$
15: **if** $\mathrm{rand}(0,1) \leqslant \exp\left\{-\dfrac{(M\_C_{\max} - C_{\max}^{old}) \times N \times S \times 10}{0.4 \times (\sum_{s=1}^{S} \sum_{j=1}^{N} P_{js})}\right\}$ **then**
16: $(\pi^{old}, C_{\max}^{old}) = (\pi, M\_C_{\max})$
17: **else**
18: $(\pi, M\_C_{\max}) = (\pi^{old}, C_{\max}^{old})$
19: **end if**
20: **end if**
21: end while

**输出:** $C_{\max}^{best}$

迭代贪婪算法基于简单易行的原则,易于实施且表现优异,已广泛应用于解决各种问题。然而,以传统的插入操作为基础的破坏重构策略和一般的局部搜索策略,在迭代过程中,其操作过程仅在调度序列中微调少数工件的相对位置难以突破搜索范围,这导致解的多样性受到了限制。

为此本文提出了采用保守跳跃破坏策略,其优点分为以下几个方面:(1) 增加破坏操作的灵活性和多样性。(2) 避免算法陷入局部最优解的困境,显著提升算法的全局搜索能力。除此之外,本文还引入了基于必经工序的局部搜索策略,其优点分为以下几个方面:(1) 成功克服了传统 $N!$ 解空间带来的挑战,将解空间 $N!$ 尽量向 $(N!)^S \prod_{s=1}^{S} \binom{N+m_s-1}{m_s-1}$ 拓展。(2) 在不失去寻找最优解的机会的情况下减少删除插入的次数,可以有效的减少计算的时间。(3) 采用有向图结构,在对工序的删除和插入过程中效率更高。(4) 有助于优化局部搜索算法的性能,使其更有效地解决复杂的混合流水车间调度问题。这些创新举措将有助于优化算法的整体效率和准确性,为解决实际问题提供了更为可靠和有效的方法。

### 3.1 破坏重构策略

破坏(Destruction)重构(Reconstitution)在迭代贪婪算法中扮演着至关重要的角色。通过引入破坏操作,算法能够有针对性地扰乱当前解的结构,有效地打破局部最优解的限制,从而更好地探索解空间的广度和深度。破坏操作赋予算法更大的灵活性和多样性,有助于避免陷入局部最优解的困境,提升算法的全局搜索能力。在原始 IG 算法,通常会设置一个参数 $d$,该参数指定随机抽取的工件个数[15],并将这些工件放入设定的工件集合之中。本文采用保守跳跃破坏策略和重构策略,这种破坏策略赋予算法更大的灵活性和多

样性，有助于避免陷入局部最优解的困境，提升算法的全局搜索能力，具体实施如算法 2 所示。

**算法 2** D&R ($\pi, flag, E, r, C_{max}$)

**输入：** $\pi$（工件序列），$flag$（判断数组是否移动），$E$（破坏的工件集合），$r$（数组下标），$C_{max}$

$C_{max}^{DR} = C_{max}$

$JDArray[3] = \{3,4,5\}$

if $flag$ then

　$\pi$ = 从 $\pi$ 中随机破坏 $JDArray[r\%3]$ 个工件，并将破坏的工件保存在 $E$ 中

else

　$r = r+1$

　$\pi$ = 从 $\pi$ 中随机破坏 $JDArray[r\%3]$ 个工件，并将破坏的工件保存在 $E$ 中

end if

for $j = 1$ to $J^e$ do

　$\pi$ = 将工件 $E_j$ 插入到 $\pi$ 中的最好位置

　将 $\pi_j$ 插入 $\pi$ 中所有可能位置，记 Makespan 最小的位置为 $p^{best}$

　$C_{max}^{DR}$ = 最小的 Makespan，并将 $\pi_j$ 插入到 $p^{best}$ 位置

end for

**输出：** $\pi$，$C_{max}^{DR}$，$r$

文献[16]中发现与流水车间相关的问题 $d$ 的取值通常落在整数范围 3 到 5 之间。因此，本文选定 3，4，5 为跳跃破坏数组的取值。其中，算法 2 的第 3～8 行表示在每一轮迭代开始时，对 $flag$ 进行判定。其中第 3～5 行表示当 $flag$ 为 $true$ 时，$r$ 不变，继续选择上一次选定的破坏值；第 6～8 行表示当 $flag$ 为 $false$ 时，$r$ 增加 1，选择跳跃破坏数组的下一个破坏值作为破坏工件数量，当 $r=2$ 时，由于数组大小为 3，继续后移，会出现数组越界的情况，则从数组起始位置选择破坏值，将 $r$ 置为 0。采用保守主义的思想，倾向于选择过去使 Makespan 变优的结果。随着迭代次数的增加，使得 $d$ 值的选择更加谨慎和稳健，从而实现 Makespan 的最小化。

### 3.2　基于必经工序的局部搜索

3.2.1　*基于必经工序的局部搜索过程*。由于 HFSP 的可行解空间是 $(N!)^S \prod_{s=1}^{S} \binom{N+m_s-1}{m_s-1}$，仅仅通过扰动调度序列只能达到 $N!$ 的解空间，与可行解空间还有相当大的差距。本文将通过局部搜索策略来改善这种状况，即将一个必经工序 $o_{js}$ 在当前机器上前移或后移到某位置，或将其移动到第 $s$ 阶段的其他机器上的某位置，称这种策略为基于必经工序的局部搜索（Mandatory Operations-Based Local Search，MOLS），它能够将解空间由 $N!$ 尽量向 $(N!)^S \prod_{s=1}^{S} \binom{N+m_s-1}{m_s-1}$ 拓展，增大搜索范围，提升算法局部搜索能力，避免陷入局部最优解。基于必经工序的局部搜索过程具体如下。

（1）生成必经工序。首先，将通过 NEH 启发式算法、保守跳跃破坏和重构算法（见算法 2）得到的序列 $\pi$ 解码成有向图 $G_Y$（见算法 3）；然后，根据定理 1 构建关键工序子图 $G_Y^S$；最后，根据寻找强连通分量的 Tarjan 算法和必经工序判定法则实现了获取关键路径上的必经工序的非递归算法（见算法 4）。算法 4 中，对图深搜时，每一个工序只访问一次，被访问过的工序与弧构成搜索树，其中，时间戳 $dfn[o_{js}]$ 记录了工序 $o_{js}$ 第一次被访问的顺序；追溯值 $low[o_{js}]$ 记录了从工序 $o_{js}$ 出发，所能访问到的最早时间戳。

必经工序判定法则：

**法则一：**如果 $o_{js}$ 对应的节点在 $G_Y$ 中不是根节点，当搜索树上 $o_{js}$ 与 $o_{j^b s^b}$ 存在 $\langle o_{js}, o_{j^b s^b} \rangle$，且满足 $low[o_{j^b s^b}] \geqslant dfn[o_{js}]$，那么 $o_{js}$ 就是必经工序。

**法则二：**如果 $o_{js}$ 对应的节点在 $G_Y$ 中是根节点，当搜索树上存在与 $o_{js}$ 相连的两条弧，$\langle o_{js}, o_{j_1^b s_1^b} \rangle$ 和 $\langle o_{js}, o_{j_2^b s_2^b} \rangle$，且满足法则一中工序之间的条件，那么 $o_{js}$ 就是必经工序。

**算法 3** *DecodeToGraph*($\pi$)

**输入**:$\pi$

01:对每台机器构建一个带有头节点和尾节点双向链表

02:$C_{js}=0, j=1,\cdots,N, s=1,\cdots,S$ , $MI_{sk}=0, s=1,\cdots S, k=1,\cdots M_s$

03:**for** $j=1$ **to** $N$ **do**

04: **for** $k=1$ **to** $M_1$ **do**

05: 用工序节点代表 $o_{j1}$,找到最早空闲时间的机器 $mt$ ,并将其可用时间记为 $MI_{1k}$

06: 工序节点的开始时间=空闲机器 $mt$ 的可用时间 $MI_{1k}$ ,将工序节点插入在第 1 阶段的机器 $mt$ 的机器链表中

07: 计算完工时间 $C_{j1}=MI_{1k}+P_{j1}$ 和机器 $mt$ 的空闲时间= $C_{j1}$

08: **end for**

09:**end for**

10:**for** $s=2$ **to** $S$ **do**

11: $\pi$ =工件根据上一阶段的完工时间进行排序的结果

12: **for** $j=1$ **to** $N$ **do**

13: **for** $k=1$ **to** $M_s$ **do**

14: 用工序节点代表 $o_{js}$ ,找到最早空闲时间的机器 $mt$ ,将其可用时间记为 $MI_{sk}$

15: 工序节点的开始时间=空闲机器 $mt$ 的可用时间 $MI_{sk}$ ,将工序节点插入在第 $s$ 阶段的机器 $mt$ 的链表中

16: 计算完工时间 $C_{js}=\max(MI_{sk}, C_{j(s-1)})+P_{js}$ 和机器 $MI_{sk}$ 的空闲时间= $C_{js}$

17: **end for**

18: **end for**

19:**end for**

20:对于每个作业构建一个双链表,用来链接每个阶段的相同工件,计算逆向完工时间 $E_{js}$ 和 $C_{\max}$

**输出**:$G_Y, C_{max}$

**算法 4** ObatiningMO ($G_Y, C_{max}$)

**输入**:$G_Y$ , $C_{\max}$

01: $MO=\varnothing, timeStamp=1$ %*timeStamp* 记录当前时间

02:根据**定理 1** 构建关键工序子图 $G_Y^S$ , $dfn[o_{js}]=0, low[o_{js}]=0$, $\forall\langle o_{js}, o_{j^b_{s^b}b}\rangle\in G_Y^S$

03:创建栈 $NS$ 用于存储当前节点的弧以及其父节点, $is\_visited[\langle o_{js}, o_{j^b_{s^b}b}\rangle]=0$, $\forall\langle o_{js}, o_{j^b_{s^b}b}\rangle\in G_Y^S$

04: $NS.push(\langle null, o_{01}\rangle)$ , $dfn[o_{01}]=timeStamp$ , $low[o_{01}]=timeStamp$ , $timeStamp=timeStamp+1$

05:**while** ( $NS\neq\varnothing$ ) **do**

06: $\langle o_{j^{parent}_{s^{parent}}}, o_{js}\rangle=NS.top()$ , $\omega=\{\langle o_{js}, o_{j^b_{s^b}b}\rangle \mid \langle o_{js}, o_{j^b_{s^b}b}\rangle\in G_Y^S \wedge o_{j^b_{s^b}b}\neq o_{j^{pt}_{s^{pt}}pt} \wedge is\_visited[\langle o_{js}, o_{j^b_{s^b}b}\rangle]=0\}$

07: **if** $\omega\neq\varnothing$ **then**

08: 从 $\omega$ 中获取一个弧 $\langle o_{js}, o_{j^b_{s^b}b}\rangle$ , $is\_visited[\langle o_{js}, o_{j^b_{s^b}b}\rangle]=1$

09: **if** ($dfn[o_{j^b_{s^b}b}]=0$) then

10: $dfn[o_{j^b_{s^b}b}]=timeStamp, low[o_{j^b_{s^b}b}]=timeStamp, timeStamp=timeStamp+1, NS.push(\langle o_{js}, o_{j^b_{s^b}b}\rangle)$

11: **else**

12: $low[o_{js}]=\min(low[o_{js}], dfn[o_{j^b_{s^b}b}])$

13: **end if**

14: **else**

15: $NS.pop()$

16: **if** ( $NS\neq\varnothing \wedge o_{j^{pt}_{s^{pt}}pt}\neq o_{01} \wedge dfn[o_{j^{pt}_{s^{pt}}pt}]\leqslant low[o_{js}]$ ) **then**

17: 将 $o_{j^{pt}_{s^{pt}}pt}$ 加入 $MO$ 中

算法 4(续) ObatiningMO ($G_Y$,$C_{\max}$)

18:　　end if

19:　　$low[o_{j^{pt}s^{pt}}] = \min(low[o_{j^{pt}s^{pt}}], low[o_{js}])$

20:　end if

21:end while

**输出**:*MO*

**算法 5 DOI ($G_Y$,*MO*,$C_{\max}$)**

**输入**:$G_Y$ , *MO* , $C_{\max}$

01:$M\_C_{\max} = C_{\max}$ , *bestPos*

02:**while** *Improvement* do

03:　*Improvement* = *false*

04:　**for** $i = 1$ **to** $|MO|$ **do**

05:　　$j = MO_j$ , $s = MO_j$ ,从 $G_Y$ 中移除 $o_{js}$ ,并将移除位置记为 $pos_{or}$ ,计算 $G_Y^-$ 完工时间 $C_{\max}(G_Y^-)$

06:　　**for** $k = 1$ **to** $M_s$ **do**

07:　　尝试插入到第 $s$ 阶段上的各个位置 $pos_{M_sr}$ ,计算 $longestvalue = \max(C_{j^{p_s}}(G_Y^-), C_{j(s-1)}(G_Y)) + P_{js} + \max(E_{j^{b_s}}(G_Y^-), E_{j(s+1)}(G_Y))$

08:　　$C_{\max}(G_Y^+) = \max(C_{\max}(G_Y^-), longestvalue)$

　　%由**定理 4** 得

09:　　**if** $(C_{\max}(G_Y^+) < M\_C_{\max})$ **then**

10:　　$M\_C_{\max} = C_{\max}(G_Y^+)$ , $bestPos = pos_{M_sr}$

11:　　**end if**

12:　　if $M\_C_{\max} < C_{\max}$ then

13:　　将 $o_{js}$ 插入到第 $s$ 阶段上的位置 *bestPos* , *Improvement* = *true*

14:　　*break*

15:　　**else**

16:　　将 $o_{js}$ 插入到原位置 $pos_{or}$

17:　　**end if**

18:　**end for**

19:　**end for**

20:**end while**

输出:$M\_C_{\max}$

(2) 对必经工序进行 DOI。首先,记录最小的 Makespan 为 $M\_C_{\max}$ ,最初的 Makepsan 为 $C_{\max}$ ,并将其赋值给 $M\_C_{\max}$ 。其次,对必经工序集合 *MO* 中的每一个必经工序 $o_{js}$ 进行遍历,在 $G_Y$ 中移除该工序得到 $G_Y^-$ ,并将原位置记为 $pos_{or}$ 。然后,将工序 $o_{js}$ 尝试插入到第 $s$ 阶段的各个位置 $pos_{M_sr}$ 得到 $G_Y^+$ ,并计算 $C_{\max}(G_Y^+)$ ,并将最小的 Makespan 的位置记为 *bestPos* 。通过判断 $C_{\max}(G_Y^+)$ 和 $M\_C_{\max}$ 的大小关系,当 $M\_C_{\max} > C_{\max}(G_Y^+)$ 时,将 *bestPos* 更新为当前位置 $pos_{M_sr}$ , $M\_C_{\max}$ 更新为 $C_{\max}(G_Y^+)$ 。最后,通过判断 $M\_C_{\max}$ 和 $C_{\max}$ 的大小关系,当 $M\_C_{\max} < C_{\max}$ 时,将 $o_{js}$ 插入到 *bestPos* 位置。当 $M\_C_{\max} \geqslant C_{\max}$ 时,将 $o_{js}$ 插入到原位置 $pos_{or}$ 。

3.2.2　基于必经工序的局部搜索的快速性。关键路径在构建邻域结构时扮演着至关重要的角色,因为邻域结构的设计是否合理直接决定了局部搜索算法的优化性能。在调度问题中,常常通过对关键路径上的工序进行扰动来实现邻域的移动。然而,并非关键路径上的所有工序的移动都能确保最大完工时间的优化。由

定理 2 可知,当存在多条关键路径时,对关键路径上的一些关键工序进行移动并未使目标得到优化,这就会耗费大量的时间。因此,本文排除了那些无法优化最大完工时间的工序(非必经工序),而是专注于移动那些可能优化最大完工时间的工序(必经工序),以此减少 DOI 的次数,进而降低时间复杂度。另外,由于每次重新计算 Makespan 会耗费大量时间,由定理 4 可知,本节选择了对 $C_{\max}(G_Y^+)$ 进行快速计算,进一步降低时间复杂度。

(1) 对必经工序的 DOI。由定义 7 可知,必经工序是关键工序的子集,且必经工序的数量 $|MO|$ 必然小于等于关键工序的数量 $|CO|$,所以当我们计算整个局部搜索过程的时候,时间复杂度由 O($|CO|\times$(DOI 的时间复杂度)) 变成了 O($|MO|\times$(DOI 的时间复杂度)),大大减小了时间复杂度,减少了算法迭代一轮时由于算法的最大运行时间相同,减少了算法迭代一轮时间,就增加了算法迭代次数,提高算法的准确性和稳定性,使算法更好地适应复杂的问题。

(2) $C_{\max}(G_Y^+)$ 的快速计算。由定理 4 可知

$$C_{\max}(G_Y^+)=\max(C_{\max}(G_Y^-),\max(C_{j^{p_s}}(G_Y^+),C_{j(s-1)}(G_Y^+))+P_{js}+\max(E_{j^{b_s}}(G_Y^+),E_{j(s+1)}(G_Y^+)))\ ,$$

根据 $G_Y$、$G_Y^-$ 和 $G_Y^+$ 之间的关系,可得

$$C_{j^{p_s}}(G_Y^+)=C_{j^{p_s}}(G_Y^-)\ ,$$

$$C_{j(s-1)}(G_Y^+)=C_{j(s-1)}(G_Y^-)=C_{j(s-1)}(G_Y)\ ,$$

$$E_{j^{b_s}}(G_Y^+)=E_{j^{b_s}}(G_Y^-)\ ,$$

$$E_{j(s+1)}(G_Y^+)=E_{j(s+1)}(G_Y^-)=E_{j(s+1)}(G_Y)\ ,$$

所以 $C_{\max}(G_Y^+)=\max(C_{\max}(G_Y^-),\max(C_{j^{p_s}}(G_Y^-),C_{j(s-1)}(G_Y))+P_{js}+\max(E_{j^{b_s}}(G_Y^-),E_{j(s+1)}(G_Y)))$。

因为 $C_{j^{p_s}}(G_Y^-)$、$C_{j(s-1)}(G_Y)$、$E_{j^{b_s}}(G_Y^-)$ 和 $E_{j(s+1)}(G_Y)$ 在前面均以计算。此时计算 $C_{\max}(G_Y^+)$ 的时间复杂度仅为 O(1),大大减少了的计算时间,增加算法迭代轮数,进而加快算法的收敛速度,使得算法更快地达到最优解。

3.2.3 基于必经工序的局部搜索策略实例验证。基于 3.3.1 节和 3.3.2 节的分析可知,基于必经工序的局部搜索策略能够更加有针对性地选择对最大完工时间产生积极影响的工序进行移动,从而提升调度方案的质量和效率,优化局部搜索算法的性能,使其更有效地解决 HFSP。本节将以甘特图的形式,通过具体实例来说明该策略的有效性和正确性。

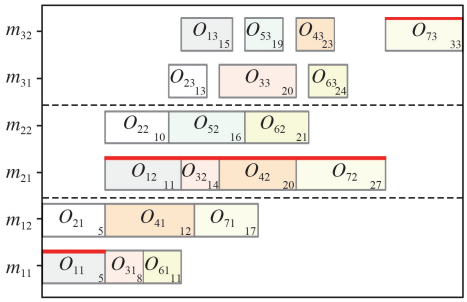



**图 3 删除非必经工序 $O_{51}$ 得到 $G_Y^-$ 的甘特图**

**Fig. 3 The Gantt chart of $G_Y^-$ obtained by removing $O_{51}$**

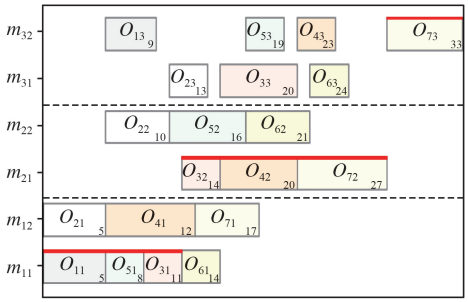



**图 4 删除非必经工序 $O_{12}$ 得到 $G_Y^-$ 的甘特图**

**Fig. 4 The Gantt chart of $G_Y^-$ obtained by removing the non-mandatory operations $O_{12}$**

如图 2(a)所示,存在两条关键路径,分别是 $P_1=\{o_{11},o_{51},o_{31},o_{32},o_{42},o_{72},o_{73}\}$ 和 $P_2=\{o_{11},o_{12},o_{32},o_{42},o_{72},o_{73}\}$,Makespan 为 33。由图 3 所示,当删除关键路径上的第 1 阶段的非必经工序 $o_{51}$ 时,关键路径 $P_1$ 遭到破坏,但关键路径 $P_2$ 仍然存在,Makespan 仍然保持在 33 不变。同样在图 4 中,当删除第 2 阶段的非必经工序 $o_{12}$ 时,关键路径 $P_2$ 遭到破坏,但关键路径 $P_1$ 仍然存在。当我们再次向 $G_Y^-$ 中插入工序时,Makespan 不会减小,最小为 33。这说明对于关键路径上的非必经工序的调整可能并不会产生预期的效果,使得 Makespan 保持不变甚至可能增加。以上示例为定理 2 提供了合理的论据。

由图 5 所示,当删除关键路径上的第 2 阶段的必经工序 $o_{32}$ 时,关键路径 $P_1$ 和 $P_2$ 都遭到破坏,形成一条新的关键路径 $P_3=\{o_{21},o_{41},o_{42},o_{72},o_{73}\}$,且 $L_{G_Y^-}(P_3)<L_{G_Y}(P_1)$,$L_{G_Y^-}(P_3)<L_{G_Y}(P_2)$,Makespan

减小为 31。由图 6 所示，将 $o_{32}$ 插入到 $G_Y^-$ 中得到 $G_Y^+$ 中，$o_{32}$ 没有成为关键工序，关键路径不发生改变，仍为 $P_3$，Makespan 也不发生改变，仍为 31。与图 2(a)相比，Makespan 得到了优化，减小了 2。这说明对于关键路径上的必经工序的调整会产生预期的效果，使得 Makespan 可能减小。以上示例为定理 2、3 提供了合理的论据。

这些示例突显了在生产计划中对关键路径和每个阶段工序顺序进行优化的重要性，同时为定理 2、3 提供了强有力的论据。对必经工序进行 DOI，Makespan 可能优化，对非必经工序进行 DOI，Makespan 无法优化。当存在多条关键路径时，删除关键路径上的工序可能使 Makepsan 减小，而删除必经工序 Makespan 一定减小，所以基于必经工序的局部搜索可以减少删除和插入次数，减少时间复杂度，同时能在较短的时间内寻找最优解。

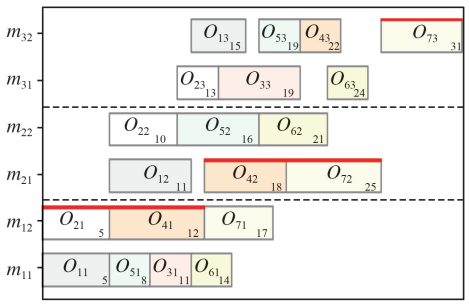



**图 5　删除必经工序 $O_{32}$ 得到 $G_Y^-$ 的甘特图**

**Fig. 5　The Gantt chart of $G_Y^-$ obtained by removing $O_{32}$**

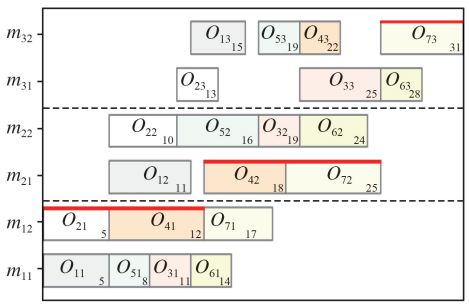



**图 6　将工序 $o_{32}$ 插入到 $G_Y^-$ 中得到 $G_Y^+$ 的甘特图**

**Fig. 6　The Gantt chart obtained by inserting $O_{32}$ into $G_Y^-$ to get $G_Y^+$**

## 4　仿真实验

所有的算法均采用 Visual Studio 2019 C++编写，并在同一台 PC 上针对所有测试用例独立运行 5 次，其中 PC 配置为英特尔酷睿 i7 处理器@2.90 GHz 和 8 GB 内存，操作系统为 Windows 11 64 位。为了公平起见，所有算法采用相同的终止条件，即算法运行时间 TimeLimit= $0.003\times J^3\times S$ ms，其中 $J$ 表示工件的数量，$S$ 表示阶段的数量。

### 4.1　实验数据集和性能指标

为了测试所提 MOAIG 的性能，本文采用了 Carlier 和 Neron 提出的 96 个基准用例，以及 Fernando-Viagas 和 Framinan 2020[18] 提出的 240 小规模和 240 个大规模基准用例来进行验证。其中，96 个基准用例由 $J\times S$ 组合而来，$J\in\{10,15\}$，$S\in\{5,10\}$，得到 4(2×2)个组合，每个组合又有 24 个规模相同但阶段中机器数量不同而组成，总计 96 个基准用例；240 个小规模用例由 $J\times S$ 组合而来，其中 $J\in\{10,15,20,25,30,35\}$，$S\in\{5,10,15,20\}$ 得到 24(6×4)个组合，每个组合又包含 10 个规模相同但加工时间不同的测试用例，总计 240(24×10)个小规模用例。同理 240 个大规模用例由 $J\times S$ 组合而来，其中 $J\in\{40,80,120,160,200,240\}$，$S\in\{5,10,15,20\}$，得到 24(6×4)个组合，每个组合又包含 10 个规模相同但加工时间不同的测试用例，总计 240(24×10)个大规模用例。本文采用相对增长百分比(Relative Percentage Increase，RPI)作为性能指标，其计算公式为[19]

$$N_{\mathrm{RPI}}=(C_{\mathrm{avg}}-C_{\mathrm{best}})/C_{\mathrm{best}}\times 100\ 。\tag{12}$$

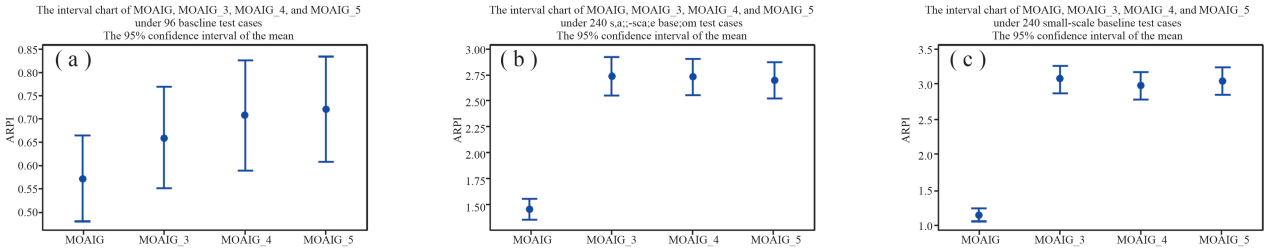
针对每一个测试用例，$C_{\mathrm{avg}}$ 统计了每个算法独立执行 5 次后的平均最大完工时间，$C_{\mathrm{best}}$ 是所有算法得到的最小 Makespan 值。由于 Carlier 和 Fernando-Viagas 测试用例规模庞大，受篇幅的限制，本文按照相同规模进行分组，即采用对相同规模算例求平均值的方法来得到最终的实验结果 ARPI。

### 4.2　保守破坏策略性能的验证

在传统的迭代贪婪算法中，参数 $d$ 决定了破坏的工件数目，它的大小直接影响了算法的收敛性以及多样性。为了降低参数对算法性能的影响，本文提出了保守跳跃破坏策略，实现了对算法的参数无依赖性，使具通用性和鲁棒性，能够更好地适应不同问题的特性，提高算法的性能。为了验证所提保守跳跃破坏策略的

有效性,本节采用 $d$ 固定值设置与所提保守跳跃破坏策略进行比较,其中,MOAIG_3,MOAIG_4 和 MOAIG_5 分别代表破坏工件数 $d$ 为 3,4 和 5 的算法。本文对每个测试用例运行了 5 次,求得了每个规模的平均 Makespan 值(记为 AVG)和平均 RPI 值(记为 ARPI)。

由图 7 的实验结果可知,对于 96 个,240 个小规模和 240 个大规模基准用例,本文采用所提保守跳跃破坏策略得到的 ARPI 值和 AVG 均小于采用 MOAIG_3,MOAIG_4 和 MOAIG_5 算法得到的值。在 95%的置信区间下,MOAIG 的 ARPI 值均小于其他三种算法,且在 240 个小规模和 240 个大规模基准用例下,与其它三种算法的 ARPI 置信区间无交集。上述结果表明,所提保守跳跃破坏策略根据当前优化的结果,自适应调整参数 $d$ 的取值,提高了搜索效率,可以更好地平衡算法的收敛性和多样性。




注:(a) 96 个基准用例;(b) 240 个小规模基准用例;(c) 240 个大规模基准用例。

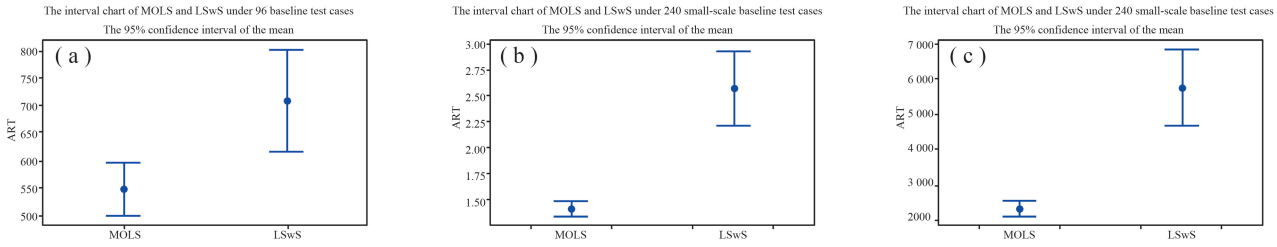
**图 7 不同基准用例下 MOAIG,MOAIG_3,MOAIG_4,MOAIG_5 的区间图**

**Fig. 7 The interval graphs of MOAIG, MOAIG_3, MOAIG_4, and MOAIG_5 under different baseline test cases**

### 4.3 基于必经工序的局部搜索策略性能的验证

本文最重要的创新是提出 MOLS 策略。由于基于关键路径的快速局部搜索方法(Local Search with Speed-up, LSwS)[20]已被证明具有快速性的特点,且在快速迭代贪婪(Iterated Greedy with Speed-up, IGwS)算法中表现出色,所以本文选取了 LSwS 进行比较来验证所提策略的快速性。为了避免其他策略对局部搜索策略产生的影响,首先,这两种局部搜索方法采用相同的初始序列进行初始化;其次,算法采取相同的算法框架,将解解码成图,然后执行局部搜索。本文对每个测试用例迭代一轮,总共运行了 5 次,求得了每个规模的平均运行时间(记为 ART,单位为μs)。

由图 8 的实验结果可知,对于 96 个,240 个小规模和 240 个大规模基准用例,本文采用所提基于必经工序的局部搜索策略得到的 ART 均小于采用 LSwS 得到的值。在 95%的置信区间下,MOLS 的 ART 值均小于 LSwS 的 ART 值,且在三种用例下,与 LSwS 的 ART 置信区间无交集。上述实验结果表明,所提 MOLS 运行时间更短,求解效率更高,表现 MOLS 的快速性。




注:(a) 96 个基准用例;(b) 240 个小规模基准用例;(c) 240 个大规模基准用例。

**图 8 不同基准用例下 MOLS 和 LSwS 的区间图**

**Fig. 8 Interval graphs of MOLS and LSwS under different baseline test cases**

### 4.4 基于必经工序的局部搜索策略整体性能的比较

为进一步测试 MOAIG 的性能,首先,本文选取了三种针对 HFSP 提出的元启发式算法比较每个规模的 AVG 和 ARPI,分别为 CSA 算法(Chaos-enhanced Simulated Annealing)、IGwS 算法和文献[21]中的 TC-NSCA 算法(Two-stage Cross-neighborhood Search Algorithm),来测试算法的实现最小化最大完工时间目标的优越性。然后,将本文所提算法和上述算法在不同工件数和阶段数的用例下进行比较 ARPI,来测试算法的稳定性和鲁棒性。本文对每个测试用例运行了 5 次,求得了每个规模的 AVG 和 ARPI。

由表 3、4、5 的实验结果可知：对于 96 个，240 个小规模和 240 个大规模基准用例，本文所提算法 MOAIG 得到的 ARPI 值和 AVG 值均小于 IGwS、CSA 和 TCNSCA 得到的值。上述实验结果表明，所提算法在实现最小化最大完工时间的目标上更加优秀，验证了 MOAIG 的有效性和正确性。

**表 3　96 个基准用例下 MOAIG 与 IGwS、CSA 和 TCANSA 的 AVG、RPI 的比较结果**

**Tab. 3　Comparison of AVG and RPI results for MOAIG, IGwS, CSA and TCANSA under 96 baseline test cases**

| Instances | MOAIG | | IGwS | | CSA | | TCANSA | |
|---|---|---|---|---|---|---|---|---|
| | AVG | RPI | AVG | RPI | AVG | RPI | AVG | RPI |
| 10×5 | 93.64 | 0.70 | 93.72 | 0.79 | 94.12 | 1.40 | 138.72 | 2.74 |
| 10×10 | 136.07 | 0.66 | 136.32 | 0.86 | 136.27 | 0.86 | 96.58 | 4.45 |
| 15×5 | 134.21 | 0.52 | 134.29 | 0.62 | 134.69 | 1.08 | 189.89 | 2.85 |
| 15×10 | 185.89 | 0.27 | 186.04 | 0.40 | 186.49 | 0.72 | 136.61 | 2.89 |
| Mean | 137.45 | 0.54 | 137.59 | 0.67 | 137.89 | 1.02 | 140.45 | 3.23 |

**表 4　240 个小规模新基准用例下 MOAIG 与 IGwS、CSA 和 TCANSA 的 AVG、RPI 的比较结果**

**Tab. 4　Comparison of RPI results for MOAIG, IGwS, CSA and TCANSA under 240 new small-scale baseline test cases**

| Instances | MOAIG | | IGwS | | CSA | | TCANSA | |
|---|---|---|---|---|---|---|---|---|
| | AVG | RPI | AVG | RPI | AVG | RPI | AVG | RPI |
| 10×5 | 421.54 | 2.49 | 428.84 | 4.30 | **419.12** | **1.94** | 451.70 | 9.86 |
| 10×10 | **801.38** | **1.17** | 815.12 | 2.86 | 807.38 | 1.97 | 870.24 | 9.85 |
| 10×15 | 941.60 | 1.71 | 955.82 | 3.31 | **939.56** | **1.56** | 986.30 | 6.44 |
| 10×20 | **1 346.68** | **1.44** | 1 360.30 | 2.36 | 1 351.04 | 1.78 | 1417.56 | 6.66 |
| 15×5 | 485.38 | 2.78 | 488.20 | 3.30 | **481.38** | **1.91** | 531.72 | 12.63 |
| 15×10 | 898.88 | 1.75 | 907.76 | 2.81 | **897.56** | **1.56** | 978.06 | 10.43 |
| 15×15 | 1 140.66 | 1.75 | 1 155.66 | 2.98 | **1 139.34** | **1.75** | 1 211.38 | 7.99 |
| 15×20 | **1 477.30** | **2.03** | 1 500.72 | 3.48 | 1 480.44 | 2.38 | 1 578.04 | 8.97 |
| 20×5 | 605.40 | 1.99 | 607.94 | 2.38 | **603.12** | **1.59** | 667.78 | 12.60 |
| 20×10 | 923.30 | 1.62 | 925.92 | 1.88 | **923.14** | **1.61** | 998.80 | 9.90 |
| 20×15 | 1 277.56 | 2.09 | 1 293.16 | 3.23 | **1 275.86** | **2.01** | 1 372.36 | 9.52 |
| 20×20 | **1 464.94** | **1.20** | 1 486.48 | 2.67 | 1 476.04 | 1.96 | 1 545.74 | 6.68 |
| 25×5 | **721.12** | **1.54** | 732.78 | 3.15 | 721.70 | 1.60 | 795.22 | 11.96 |
| 25×10 | 987.60 | 2.33 | 997.24 | 3.34 | **983.02** | **1.88** | 1 069.62 | 10.93 |
| 25×15 | **1 259.60** | **1.85** | 1 276.56 | 3.23 | 1 262.24 | 2.04 | 1 350.26 | 9.19 |
| 25×20 | 1 471.92 | 1.90 | 1 493.84 | 3.44 | **1 470.86** | **1.83** | 1 557.88 | 7.91 |
| 30×5 | **750.78** | **0.97** | 760.78 | 2.32 | 756.16 | 1.75 | 843.72 | 13.51 |
| 30×10 | **1 058.10** | **1.23** | 1 078.36 | 3.18 | 1 066.76 | 2.04 | 1 161.90 | 11.15 |
| 30×15 | **1 463.84** | **1.41** | 1 493.02 | 3.44 | 1 481.64 | 2.61 | 1 610.36 | 11.43 |
| 30×20 | **1 835.00** | **1.45** | 1 859.08 | 2.88 | 1 855.42 | 2.56 | 1 997.66 | 10.02 |
| 35×5 | **912.50** | **1.07** | 927.88 | 2.73 | 916.58 | 1.58 | 1 014.14 | 12.57 |
| 35×10 | **1 264.34** | **1.21** | 1 280.70 | 2.33 | 1 268.22 | 1.61 | 1 402.28 | 12.48 |
| 35×15 | **1 580.46** | **1.39** | 1 608.64 | 3.07 | 1 599.04 | 2.66 | 1 716.86 | 10.18 |
| 35×20 | **1 828.42** | **1.29** | 1 854.02 | 2.76 | 1 851.42 | 2.61 | 1 956.48 | 8.50 |
| Mean | **1 121.60** | **1.65** | 1 137.03 | 2.98 | 1 126.13 | 1.95 | 1 211.92 | 10.06 |

表 5 240 个大规模新基准用例下 MOAIG 与 IGwS、CSA 和 TCANSA 的 AVG、RPI 的比较结果

Tab. 5 Comparison of RPI results for MOAIG, IGwS, CSA and TCANSA under 240 new large-scale baseline test cases

| Instances | MOAIG | | IGwS | | CSA | | TCANSA | |
|---|---|---|---|---|---|---|---|---|
| | AVG | RPI | AVG | RPI | AVG | RPI | AVG | RPI |
| 40×5 | **800.82** | **0.91** | 813.94 | 2.50 | 809.02 | 1.97 | 863.60 | 8.67 |
| 40×10 | **941.90** | **1.82** | 955.70 | 3.35 | 947.74 | 2.35 | 990.20 | 6.81 |
| 40×15 | **1 341.20** | **1.17** | 1 360.62 | 2.49 | 1 349.76 | 1.76 | 1 407.82 | 6.09 |
| 40×20 | **418.06** | **1.79** | 426.40 | 3.84 | 418.64 | 1.94 | 450.00 | 9.57 |
| 80×5 | 895.24 | 1.41 | 910.88 | 3.19 | **894.42** | **1.27** | 976.86 | 10.35 |
| 80×10 | **1 131.24** | **1.14** | 1 154.74 | 3.02 | 1 139.06 | 1.93 | 1 208.52 | 7.93 |
| 80×15 | 1 481.68 | 2.11 | 1 499.24 | 3.23 | **1 480.78** | **2.20** | 1 572.42 | 8.39 |
| 80×20 | 486.02 | 2.50 | 487.80 | 2.79 | **482.72** | **1.82** | 530.52 | 11.97 |
| 120×5 | 923.70 | 2.04 | 926.16 | 2.25 | **921.80** | **1.78** | 1 006.84 | 11.19 |
| 120×10 | 1 278.30 | 1.89 | 1 291.38 | 2.74 | **1 277.26** | **1.80** | 1 368.14 | 8.75 |
| 120×15 | **1 468.64** | 1.39 | 1 489.20 | 2.80 | 1 478.64 | 2.07 | 1 548.24 | 6.91 |
| 120×20 | **601.74** | 1.65 | 607.14 | 2.55 | 604.32 | 2.12 | 670.34 | 13.52 |
| 160×5 | 981.52 | 1.59 | 1 001.54 | 3.64 | **980.22** | **1.46** | 1 068.72 | 10.72 |
| 160×10 | **1 254.56** | **1.59** | 1 279.04 | 3.60 | 1 262.72 | 2.25 | 1 357.46 | 9.99 |
| 160×15 | **1 457.66** | **1.22** | 1 491.36 | 3.57 | 1 475.58 | 2.46 | 1 558.08 | 8.28 |
| 160×20 | **709.70** | **0.59** | 731.16 | 3.64 | 723.40 | 2.53 | 791.60 | 12.21 |
| 200×5 | **1 054.74** | **1.40** | 1 077.64 | 3.62 | 1 066.12 | 2.44 | 1 164.10 | 11.96 |
| 200×10 | **1 448.02** | **1.04** | 1 491.26 | 4.18 | 1 480.04 | 3.32 | 1 606.24 | 11.99 |
| 200×15 | **1 810.36** | **1.00** | 1 861.000 | 3.91 | 1 852.26 | 3.31 | 1 996.38 | 10.99 |
| 200×20 | **748.24** | **0.88** | 760.70 | 2.60 | 757.88 | 2.22 | 844.56 | 13.88 |
| 240×5 | **1 259.50** | **1.19** | 1 282.92 | 2.95 | 1 267.88 | 2.00 | 1 414.10 | 13.83 |
| 240×10 | **1 556.72** | **0.78** | 1 604.54 | 3.80 | 1 595.302 | 3.19 | 1 722.56 | 11.55 |
| 240×15 | **1 809.20** | **0.93** | 1 854.54 | 3.50 | 1 851.14 | 3.26 | 1 966.00 | 9.81 |
| 240×20 | **904.48** | **0.65** | 928.70 | 3.35 | 919.66 | 2.42 | 1 016.00 | 13.406 |
| Mean | **1 115.14** | **1.36** | 1 136.98 | 3.21 | 1 126.52 | 2.24 | 1 212.47 | 10.37 |

## 5 结论

混合流水车间调度在制造业中具有极其重要的意义。合理的混合流水车间调度可以帮助企业降低生产成本,缩短生产周期,提高交付效率,从而增强市场竞争力。此外,通过优化混合流水车间的调度,可以实现生产过程的平稳运行,减少生产中的浪费和停顿,提升生产线的稳定性和可靠性。混合流水车间调度的重要性在于其直接关系到企业的经济效益、生产效率和客户满意度,是提升制造业竞争力和实现持续发展的关键之一。本文对混合流水车间调度问题展开了研究,首先,对混合流水车间调度问题进行了问题描述和符号定义,建立了以最小化最大完工时间为优化目标的数学模型;然后针对混合流水车间调度问题的特性提出了一种基于必经工序的加速迭代贪婪算法;最后通过仿真实验与一些经典算法进行比较分析,验证所提算法求解效率高,解的质量优良,具有更好的性能。

未来,考虑到制造环境的日益复杂,分布式车间调度、客户满意度和柔性交货计划等因素至关重要,将本文所提方法扩展到上述应用场景。特别地,本文未考虑必经工序的有效位置选择问题,也就是说必经工序不再尝试同一阶段的所有位置插入,而是挖掘位置的特性,选择能使目标值变优的位置插入,这一策略将进一

步提升求解的速度。总体而言，所提快速评价方法具有显著提高调度效率、降低生产成本的潜力，在这一领域的持续研究和发展无疑将带来创新的解决方案。

## 参 考 文 献


[1] QIN H X, HAN Y Y, CHEN Q D, et al. Energy-efficient iterative greedy algorithm for the distributed hybrid flow shop scheduling with blocking constraints[J]. IEEE Transactions on Emerging Topics in Computational Intelligence, 2023, 7(5): 1442-1457.

[2] NOWICKI E, SMUTNICKI C. The flow shop with parallel machines: a tabu search approach[J]. European Journal of Operational Research, 1998, 106(2-3): 226-253.

[3] LIAO C J, TJANDRADJAJA E, CHUNG T P. An approach using particle swarm optimization and bottleneck heuristic to solve hybrid flow shop scheduling problem[J]. Applied Soft Computing, 2012, 12(6): 1755-1764.

[4] CARLIER J, NERON E. An exact method for solving the multi-processor flow-shop[J]. RAIRO-Operations Research, 2000, 34(1): 1-25.

[5] PAN Q K, WANG L, LI J Q, et al. A novel discrete artificial bee colony algorithm for the hybrid flowshop scheduling problem with makespan minimisation[J]. Omega, 2014, 45: 42-56.

[6] LI J Q, PAN Q K, WANG F T. A hybrid variable neighborhood search for solving the hybrid flow shop scheduling problem[J]. Applied Soft Computing, 2014, 24: 63-77.

[7] PATERNINA-ARBOLEDA C D, MONTOYA-TORRES-TORRES J R, ACERO-DOMINGUEZ M J, et al. Scheduling jobs on a k-stage flexible flow-shop[J]. Annals of Operations Research, 2008, 164(1): 29-40.

[8] VAIRAKTARAKIS G, ELHAFSI M. The use of flowlines to simplify routing complexity in two-stage flowshops[J]. IIE Transactions, 2000, 32(8): 687-699.

[9] LIN S W, CHENG C Y, POURHEJAZY P, et al. New benchmark algorithm for hybrid flowshop scheduling with identical machines [J]. Expert Systems with Applications, 2021, 183: 115422.

[10] FAN J X, Li Y L, XIE J, et al. A hybrid evolutionary algorithm using two solution representations for hybrid flow-shop scheduling problem[J]. IEEE Transactions on Cybernetics, 2021, 53(3): 1752-1764.

[11] 任彩乐，张超勇，孟磊磊，等. 基于改进候鸟优化算法的混合流水车间调度问题[J]. 计算机集成制造系统，2019，25(3)：643.

[12] 李俊青，李荣昊，陶昕瑞，等. 帝国竞争算法求解资源约束混合流水车间调度问题[J]. 聊城大学学报(自然科学版)，2022，35(2)：14-26.

[13] LI C S, HAN Y Y, ZHANG B, et al. A novel collaborative iterative greedy algorithm for hybrid flowshop scheduling problem with batch processing machines and variable sublots[J]. International Journal of Production Research, 2024, 62(11): 4076-4096.

[14] NAWAZ M, ENSCORE JR E E, HAM I. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem[J]. Omega, 1983, 11(1): 91-95.

[15] 王凌，潘子肖. 基于深度强化学习与迭代贪婪的流水车间调度优化[J]. 控制与决策，2021，36(11)：2609-2617.

[16] RUIZ R, STÜTZLE T. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem[J]. European Journal of Operational Research, 2007, 177(3): 2033-2049.

[17] 秦浩翔，韩玉艳，陈庆达，等. 求解阻塞混合流水车间调度的双层变异迭代贪婪算法[J]. 控制与决策，2022，37(9)：2323-2332.

[18] FERNANDEZ-VIAGAS V, FRAMINAN J M. Design of a testbed for hybrid flow shop scheduling with identical machines[J]. Computers & Industrial Engineering, 2020, 141: 106288.

[19] 潘玉霞，潘全科，桑红燕，等. 解决批量流水线调度问题的离散微粒群算法[J]. 聊城大学学报(自然科学版)，2009，22(3)：90-93.

[20] FERNANDEZ-VIAGAS V. A speed-up procedure for the hybrid flow shop scheduling problem[J]. Expert Systems with Applications, 2022, 187: 115903.

[21] KUANG Y, WU X L, CHEN Z Q, et al. A two-stage cross-neighborhood search algorithm bridging different solution representation spaces for solving the hybrid flow shop scheduling problem[J]. Swarm and Evolutionary Computation, 2024, 84: 101455.


(责任编辑：刘庆松)