Heuristic Algorithms on Flexible Flow Shop with Unrelated Machines, Setup Time, Limited Storage Capacity, and Specification Constraint

An-Che Liang , Wei-Hsiang Huang, Debbie Huang, You-Ming Yeh, Ling-Chieh Kung
Department of Information Management
National Taiwan University

Abstract

In this paper, we address the production-planning challenge in a multi-stage flexible flow shop with unrelated machines, sequence-dependent setup times, limited storage capacity, and product-specification constraints. We first formulate the problem as a mixed-integer linear program (MILP). Recognizing that solving large-scale instances to optimality is computationally prohibitive, we then propose two greedy heuristics: Simulated Annealing–based Greedy Allocation (SAGA) and Linear Relaxation–based Greedy Allocation (LRGA). Extensive computational experiments show that SAGA and LRGA achieve average optimality gaps of 8.44% and 23.34%, respectively, as long as the storage capacity of each stage are not too limited, while delivering runtimes orders of magnitude faster than leading MILP solvers—enabling fast, reliable production scheduling in practice. In a real-world case study spanning 141 days with 92 orders, SAGA and LRGA eliminate all delayed deliveries and boost the on-time delivery rate from 9.1% under current practice to 69.7% and 53.0%, respectively.

Keywords: flexible flow shop, unrelated machines, limited storage capacity, specification constraint, simulated annealing, linear relaxation.

1 Introduction

Manufacturers worldwide compete in markets where on-time delivery is as crucial as product quality. Conventional planning methods, which often rely on heuristic "rules of thumb," suffer two major drawbacks, which are tardiness risk and excess inventory. Tardiness risk arises when an order finishes after its promised date, causing it to miss its delivery window and may forfeit sales. On the other hand, excess inventory happens when an order completes too early, and thus surplus goods occupy storage, which ties up capital and driving up holding costs.

This study is motivated by a real-world case with Shang Tian Aluminum Co., a manufacturer specializing in aluminum windows and doors, which approached us to address their production scheduling challenges. By analyzing their historical order data and factory resources, we gained insights that enabled us to generalize our approach for broader applications in production planning, which may help small to medium enterprises in the manufacturing industry.

Most factories operate under a flow-shop model, in which jobs move through successive stages (e.g., assembly, inspection, packaging). Each order may have distinct processing times, setup times, storage-space requirements, and specification constraints that make manual scheduling increasingly complex.

To address these challenges, we first formulate the scheduling problem as a mixed-integer linear program (MILP) that minimizes the combined penalties for earliness and tardiness. Because the MILP is NP-hard, we then introduce two greedy-

2025 International Conference and Annual Meeting of the Operations Research Society of Taiwan, Undergraduate Thesis Competition (ORSTW 2025).

based heuristic algorithms designed to generate near-optimal solutions within practical time limits.

The remainder of this paper is organized as follows. Section 2 reviews related literature. Section 3 formulates our problem as a mixed-integer linear program. Section 4 introduces the proposed heuristic algorithms. Section 5 evaluates the performance of heuristic algorithms. Section 6 presents a real-world case study involving with Shang Tian Aluminum Co. Finally, Section 7 concludes the paper.

2 Literature review

The problem in our study is a multi-stage flexible flow shop with unrelated machines, specification constraint, independent setup time, and storage space limits problem. In the operation research field, multi-stage flow shop problem is a classic, long-standing, and extensively studied NP-hard problem. Many previous studies on multi-stage production planning problems have used linear programming to develop planning solutions.

Gabbay (1979) [2] studies a multi-stage, multi-item, capacitated production and inventory planning problem. The author proposes a hierarchical planning framework. It first solves an aggregate capacity planning problem using linear programming under certain cost assumptions, then decomposes the solution into detailed item-level production schedules. He develops both a one-pass algorithm and a hierarchical procedure that reduces detailed forecasting needs, divides decisions between tactical and operational levels, and remains computationally efficient without sacrificing optimality.

Tan et al. (2018) [5] addresses a two-stage flexible flow shop problem where the first stage involves batch processing machines. The focus is on minimizing makespan while considering batching constraints. The authors propose a hybrid scheduling approach combining decomposition techniques and variable neighborhood search (VNS) to efficiently solve the problem.

However, because we have setup time, it's natural for the model to generate binary variables, turning the model into an integer program (IP), which makes it infeasible to solve using purely mathematical programming. In such cases, where the problem becomes too complex to solve efficiently with standard mathematical approaches, heuristics like the listing algorithm are often employed. These heuristics provide a practical alternative, offering feasible solutions for flow shop problems by simplifying the decision-making process and reducing computational complexity.

Nawaz et al. (1983) [4] studied the flow-shop sequencing problem, where multiple jobs must be processed in the same machine order, aiming to minimize makespan. They propose a heuristic algorithm based on prioritizing jobs with higher total processing times. The algorithm builds the job sequence step by step, using a curtailed-enumeration strategy to efficiently find high-quality solutions.

Danneberg et al. (1999) [1] studies permutation flow shop scheduling problems where jobs belong to different groups and can be processed in batches, with limited batch size and setup times that depend on the group. They focus on minimizing either the makespan or the weighted sum of completion times. They propose and compare a variety of constructive heuristics and iterative local search algorithms, including standard metaheuristics like simulated annealing and tabu search, as well as multilevel search procedures using different neighborhood structures.

But because we have complications like specification constraints, unrelated machines, and storage limits in our problem, using just the listing algorithm isn't effective enough. Combining the above, we decided to develop a solution that integrates mathematical programming with the heuristic algorithm, aiming to achieve both low cost and low computation time. Therefore, our problem is worth of investigation.

3 Model

In this section, we first describe the general problem in Section 3.1. We then present a general transformation method to transform general problems into our abstract formulation in Section 3.2. Then in Section 3.3, we present our mathematical formulation of our model setting.

3.1 Problem description

In a typical flow shop problem, every order passes through the same set of stages. However, due to the presence of specification constraints in our problem, different machines within each stage may handle different tasks. As a result, the stages that each order needs to go through may vary. Below is the definition of our problem:

There are multiple stage groups on the flow line. Each stage group contains at least one stage. The stages within a stage group are independent of one another and can process tasks in parallel, but they share the same stage group storage space.

The planning process is divided into several periods, and each stage has a capacity limit in each period. In our problem, the capacity is measured in daily working time (minutes). After completing a stage, the produced order is transported to the next required stage for further processing. Each order has a due period. Early or late deliveries will incur corresponding earliness or tardiness costs. Each order also has its own specification constraint, which defines the required stages it must go through. Each stage group will only have one stage selected for any given order. The number of stages that each order needs to go through is equal to the number of stage groups. Producing an order at a stage incurs a specific setup time. If a stage processes an order during any period, it will consume the associated setup time for that order at that stage. We use Figure 1 to illustrate one example of a production process with three stage groups and six stages. In the example, stage group 1 has two stages 1-A and 1-B; stage group has three stages 2-A, 2-B, and 2-C; and stage group 3 has two stages 3-A and 3-C. The specification constraint of an order i' needs to be processed at stages 1-A, 2-B, and 3-A.

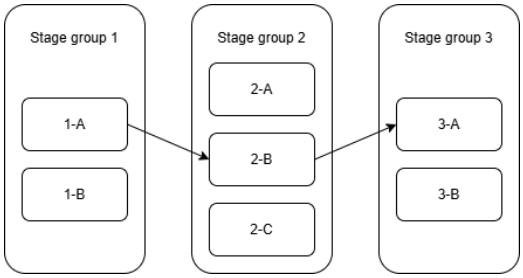

Figure 1: Stage groups example

3.2 Problem transformation

Based on the above problem definition, we can transform the flow shop problem with specification constraints into a standard flow shop problem through data preprocessing. In our transformation, we can flatten the stage groups to stages one by one, and assign 0 processing load at stages that an order i won't go through, that is, the stages other than the specification constraint of order i. In the meantime, we set both the setup cost and storage usage to 0. In the original problem, we would imagine the order i' being transferred to stage 2-B after being completed in stage 1-A, then processed, and subsequently transferred to stage 3-A. However, in Figure 2, we assign order i' 0 processing load at stages 2-A and 2-C, and set both the setup cost and storage usage at stages 2-A and 2-C to 0. After stage 1-A completes its work, the order is sequentially routed through stage 2-A, 2-B, and 2-C, undergoing virtual processing at 2-A, actual processing at 2-B, and virtual processing again at 2-C, before being sent to stage 3-A. Through this design, we successfully convert the flow shop problem with specification constraints into a standard flow shop problem.

Under this problem setting, our objective is to develop an effective production scheduling plan that enables the factory to determine, for each period t and stage s, which orders i should be processed, in order to minimize the total cost of earliness and tardiness. Let the indices and sets be defined as follows. Let $I = \{1, ..., I^*\}$ denote the set of orders, $T = \{1, ..., T^*\}$ denote the set of periods, $S = \{1, ..., S^*\}$ denote the set of stages, and $G = \{1, ..., G^*\}$ denote the set of stage groups. For each stage group $g \in G$, let $S_g \subseteq S$ denote the set of stages belonging to stage group g. Let $\delta \subseteq S$ represent the set of final stages within each stage group. In the following, we present the model. We use uppercase letters to denote sets and parameters, and lowercase letters for decision variables.

3.3 Mathematical formulation

The objective function aims to minimize the total delivery cost, which includes both earliness and tardiness penalties. Each order i has a scheduled due period D_i. If the order is delivered early or late, it incurs a penalty. Specifically, P_i is earliness penalty per period for order i, Q_i is tardiness penalty per period for order i, z_i is number of periods early that order i is delivered, w_i is number of periods late that order i is delivered, α is weight of earliness penalty in the objective function, and β is weight of tardiness penalty in the objective function. To minimize the total delivery cost, the objective function is defined as

$$\min \quad \alpha \sum_{i \in I} P_i z_i + \beta \sum_{i \in I} Q_i w_i,$$

this formulation allows the model to flexibly prioritize early delivery or on-time delivery based on the values of α and β.

During production and transportation at each stage, several constraints must be considered. We first focus on the constraints related to production

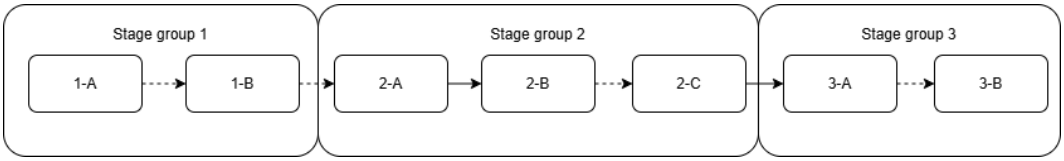

Figure 2: Transformed stages example

and transportation, which are

$$h^{\text{next}}_{tsi} = h^{\text{prev}}_{tsi} + \frac{x_{tsi}}{R_{is}} - \frac{x_{t(s+1)i}}{R_{i(s+1)}} \quad \forall t \in T, \forall s \in S \setminus \{S^*\}, \forall i \in I \tag{1}$$

$$h^{\text{next}}_{tS^*i} = h^{\text{prev}}_{tS^*i} + \frac{x_{tS^*i}}{R_{iS^*}} - c_{it} \quad \forall t \in T, \forall i \in I \tag{2}$$

$$h^{\text{prev}}_{(1)si} = 0 \qquad \forall s \in S, \forall i \in I \tag{3}$$

$$h^{\text{prev}}_{(t+1)si} = h^{\text{next}}_{tsi} \quad \forall t \in T \setminus \{T^*\}, \forall s \in S, \forall i \in I. \tag{4}$$

In these constraints, R_{is} is the required capacity for stage s to produce order i, x_{tsi} is the capacity invested by stage s in period t for producing order i, h^{prev}_{tsi} is the completion rate of order i at stage s at the beginning of period t, and h^{next}_{tsi} is the completion rate of order i at stage s at the end of period t. Constraint (1) ensures the balance between the input, output, and the completion rate of each order at every stage. The value of h^{next}_{tsi} equals h^{prev}_{tsi} plus the progress made during this period minus the progress transferred to the next stage. Constraint (2) applies to final stage and ensures the balance between the input, output, and the delivery progress of the order. Constraint (3) initializes the completion rate of all orders to zero at the beginning of the first period. Constraint (4) ensures that the starting completion rate of a given order at a stage in the current period equals the ending completion rate of the previous period. These constraints together govern how production progresses over time and how orders flow through the stages toward completion and delivery.

Next, we focus on the constraints related to production and setup cost, which are

$$\sum_{t \in T} x_{tsi} = R_{is} \qquad \forall s \in S, \forall i \in I \tag{5}$$

$$x_{tsi} \leq M y_{tsi} \qquad \forall t \in T, \forall s \in S, \forall i \in I \tag{6}$$

$$\sum_{i \in I} x_{tsi} + \sum_{i \in I} A_{is} y_{tsi} \leq L \forall t \in T, \forall s \in S. \tag{7}$$

In these constraints, A_{is} is the setup cost required for stage s to produce order i, L is the capacity limit of each stage per period, M is a sufficiently large constant used to enforce constraints only when necessary, $y_{tsi} \in 0, 1$ is a binary decision variable indicating whether stage s produces order i in period t. Constraint (5) ensures that the total production across all periods for each order at each required stage must meet the demand, i.e., the required production load must be satisfied. Constraint (6) ensures that production capacity x_{tsi} can only be allocated if $y_{tsi} = 1$. This is enforced using the big-M method. Constraint (7) ensures that the total capacity used by a stage in a period, including both the production load and the setup cost, does not exceed the stage's capacity limit L. Together, these constraints regulate how stages allocate capacity, trigger setup costs, and stay within production limits over time.

Next, we address the constraints related to stage groups, which are

$$\frac{x_{t(s+1)i}}{R_{i(s+1)}} \leq h^{\text{prev}}_{tsi} \quad \forall t \in T, \forall s \in \delta \setminus \{S^*\}, \forall i \in I \tag{8}$$

$$\sum_{s \in S_g} \sum_{i \in I} h^{\text{prev}}_{tsi} K_{is} \leq H_g \quad \forall t \in T, \forall g \in G. \tag{9}$$

In these constraints, K_{is} is the storage space required by order i at stage s, H_g is the storage capacity limit of stage group g, S_g is the set of stages in stage group g, δ is the set of final stages within each stage group. Constraint (8) means, for each final stage of a stage group, this constraint ensures that the input progress at the next stage (i.e., stage $s + 1$) in period t cannot exceed the completion progress h^{prev}_{tsi} at the current stage. This regulates that an order cannot proceed to the next stage unless it has been completed at the current one. Constraint (9) ensures that the total space occupied by all in-process orders at any given stage group does not exceed its storage capacity H_g. The storage occupied by an order is assumed to be proportional to its current progress, multiplied by the order's specific storage requirement K_{is}. Summing this across all relevant stages and orders gives the total space usage at the stage group. These constraints ensure that order flow respects processing sequence and that the shared storage within each stage group is not exceeded.

Then, we consider the constraints related to order delivery, which are

$$\sum_{t \in T} c_{it} = 1 \qquad \forall i \in I \qquad (10)$$

$$z_i \geq D_i - \sum_{t \in T} t c_{it} \qquad \forall i \in I \qquad (11)$$

$$w_i \geq \sum_{t \in T} t c_{it} - D_i \qquad \forall i \in I. \qquad (12)$$

In these constraints, D_i is the due period for order i, c_{it} is a binary decision variable indicating whether order i is delivered in period t, z_i is the number of periods by which order i is delivered early, w_i is the number of periods by which order i is delivered late. Constraint (10) ensures that each order is delivered in exactly one period. Constraint (11) calculates the earliness of delivery. If an order is delivered before its due period D_i, then z_i will reflect the number of periods it is early. Constraint (12) calculates the tardiness of delivery. If an order is delivered after its due period D_i, then w_i will reflect the number of periods it is late. These constraints ensure that each order is delivered once, and accurately track how early or late each delivery is, which in turn affects the penalty costs in the objective function.

Finally, we introduce the variable domain constraints, which are

$$x_{tsi}, h^{\text{prev}}_{tsi}, h^{\text{next}}_{tsi}, z_i, w_i \geq 0 \quad \forall t \in T, \forall s \in S, \forall i \in I \qquad (13)$$

$$y_{tsi}, c_{it} \in \{0, 1\} \ \forall t \in T, \forall s \in S, \forall i \in I. \qquad (14)$$

Constraint (13) ensures that the decision variables x_{tsi}, h^{prev}_{tsi}, h^{next}_{tsi}, z_i, w_i are non-negative continuous variables, while constraint (14) ensures that the decision variables y_{tsi}, c_{it} are binary variables.

We summarize the complete sets, parameters, and decision variables in the following tables in the Appendix: Table 5, Table 6, and Table 7.

4 Algorithm

In this section, we first present the core greedy procedure for allocating processing time to orders in Section 4.1. We then introduce two complete heuristic algorithms: SAGA (Section 4.2) and LRGA (Section 4.3), which augment the greedy allocation procedure with different sorting strategies.

4.1 Greedy Allocation procedure (GA)

The core of our heuristic is the *Greedy Allocation* (GA) procedure, which sequentially assigns processing time and storage space to each order in a given sequence. GA (Algorithm 1) processes orders one at a time, finding a feasible schedule for order i before proceeding to the next, so the choice of sequence is critical to overall performance.

When scheduling order i, GA aims to complete its final stage precisely at its due date d_i, thereby minimizing both earliness and tardiness penalties. Beginning at stage S in period d_i, GA performs a backward pass through each preceding stage s, computing feasible start and end times while enforcing setup durations, specification constraints, and storage limits. If no feasible schedule exists for completion at d_i, GA iteratively shifts the target completion earlier or later guided by the relative earliness and tardiness penalty weights until a valid solution is found.

To compute how much time and space can be allocated to order i at stage s in period t, we define

$$\texttt{calculate\_h}(t, s, i) \ \rightarrow \ \left(h^{\text{prev}}_{tsi}, \ h^{\text{next}}_{tsi}\right),$$

where h^{prev}_{tsi} and h^{next}_{tsi} represent the order completion rate before and after t, respectively. We compute these values using the allocation results of earlier orders, as detailed in Algorithm 2. Given these values, function `check_residual` (Algorithm 4) using time and space limits, subtracting the production time already allocated and the space occupied, returns the residual processing time and storage capacity available for allocation.

Since we can know how much time and space can be allocated to order i at stage s in period t, we have to schedule the process stage by stage according to given period t. In the `schedule_backward` function (Algorithm 3), we tried to start from the last stage and work backwards, allocating time to order production at each stage. It is also essential to account for *future* space already reserved by the current order. For example, if at $t = 3$ GA occupies all 10 m^2 of residual storage, then at $t = 2$ it must recognize that those 10 m^2 are no longer available. We enforce this via the variables `residual_space_higher_bound` and `available_space` to prevent over-allocation. Additionally, unsaturated orders from previous stages may retain storage longer than anticipated; GA checks that these do not exceed future storage capacity.

Combining these components yields the GA procedure, which, for any given order sequence, produces a feasible production plan. Applying different sorting strategies to this sequence gives rise to our two complete heuristics: SAGA and LRGA.

Algorithm 1 `GA`

Input
  OS orders' sequence
  T^* number of periods
for order $\in$ OS **do**
  $i :=$ order's id
  $t :=$ order's due period
  while $t < T^*$ **do**
    scheduling_result = schedule_backward(i, t)
    if scheduling_result is not None **then**
      apply_scheduling_result(i, t, scheduling_result)
      $break$
    end if
    $t := t +$ step_direction $\times$ step_size
    step_direction := $-1 \times$ step_direction
    step_size := step_size $+ 1$
  end while
end for

Algorithm 2 `calculate_h`

Input
  t plan period
  s stage_id
  i order_id
$s_{\text{next}} :=$ the stage_id that this order will go through in the next stage group
h_prev := 0 #begin completion ratio
for $t' \in t$ **do**
  update h_prev based on previously applied x_{tsi} and $x_{ts_{\text{next}}i}$
end for
h_next := h_prev #end completion ratio
update h_next
$return$ h_prev, h_next

4.2 Simulated Annealing based Greedy Allocation (SAGA)

In SAGA, we use a simulated annealing strategy on top of our greedy allocation heuristic to enhance solution quality. We begin with an initial sequence of orders sorted by non-decreasing due dates, then apply the greedy allocation procedure to produce a feasible schedule whose objective value we call the "energy" e. At each iteration k (up to $k_{\max}$), we perform the following steps: First, we generate a neighbor sequence by swapping two orders in the current sequence. After we generate the new sequence, we apply GA to the sequence and yield energy e'. If $e' < e$, we accept the new sequence. Otherwise, we would accept with probability

$$P_k \;=\; \exp\Big(-\tfrac{e'-e}{T(k)}\Big),$$

where

$$T(k) = \frac{k_{\max}}{k+1}.$$

Whenever a swap is accepted, we set the current sequence to the new sequence and update $e \leftarrow e'$.

We set $k_{\max} = 100$; after at most $k_{\max}$ iterations, SAGA returns the best production plan it has found.

4.3 Linear Relaxation based Greedy Allocation (LRGA)

The MILP formulation delivers exact solutions but becomes computationally infeasible for medium to large scale instances, as its runtime grows exponentially with problem size. Although the LP relaxation may yield fractional (and hence infeasible) schedules, these fractional variables provide valuable priority information for the greedy allocation (GA) procedure.

In LRGA, we first solve the LP relaxation to optimality, obtaining fractional decision variables c'_{it}. We then compute each order's priority index

$$t_i \;=\; \max\big\{\, t \in T \mid c'_{it} > 0 \big\}.$$

Algorithm 3 `schedule_backward`

Input
    t plan period
    i order_id
allocation_result := {}
$list_s$:= order's processing stages in each stage group
used_space = 0
for stage_id $\in list_s$ **do**
    used_space = 0
    remaining_process_time = R_{is}
    space_ratio = $\frac{R_{is}}{K_{is}}$
    while $t >= 0$ **do**
        residual_time, residual_space = check_residual(stage_id, t)
        residual_space_higher_bound = $\min$(residual_space_higher_bound, residual_space)
        available_space = residual_space_higher_bound − used_space
        if available_space < 0 **then**
            No available space, break the loop
        end if
        max_processing_time = $\min$(residual_time − setup_time, available_space × space_ratio)
        allocated_processing_time = $\min$(max_processing_time, remaining_process_time)
        if allocated_processing_time > 0 **then**
            record the allocated processing time in result
            remaining_process_time := remaining_process_time − allocated_processing_time
            used_space = used_space + allocated_processing_time × space_ratio
        end if
        $t := t - 1$
        if remaining_processing_time ≤ 0 **then**
            the work for this stage is done, break the loop
        end if
    end while
    time_worked_in := list of periods that process order i in the future
    if stage_id is not the last stage **then**
        for $t' \in$ time_worked_in **do**
            next_stage_id := stage id of the next stage
            _, max_allowed_space = check_residual(next_stage_id, t')
            calculate remaining_blocking_space
            if remaining_blocking_space > max_allowed_space **then**
                The unsaturated order takes up too much space after it has been completed
                The result is not feasible, $return\ None$
            end if
        end for
    end if
end for
$return$ result

Algorithm 4 `check_residual`

Input
    t plan period
    s stage_id
    I^* number of orders
residual_time := time per period #available time at stage s in period t
residual_space := space of the stage group #available time at stage s in period t
for $i \in$ range(I^*) **do**
    if stage s process order i at period t **then**
        residual_time := residual_time $-x[t, s, i] - A[i, s]$
    end if
end for
for $i \in$ range(I^*) **do**
    for $s' \in$ stage_group **do**
        _, h = calculate_h(t, s', i)
        residual_space := residual_space $-h \times K[i, s]$
    end for
end for
$return$ residual_time, residual_space

To ensure that the relaxed LP solution remains a close proxy for the original MILP, it is critical to choose the smallest possible M that still validates the constraint. Since M appears only in Constraint (6), we simply set $M = L$.

Sorting orders in descending order of t_i produces a sequence of orders, which serves as the input to the GA procedure. Then, the GA procedure is applied to this prioritized sequence to produce a feasible production plan.

5 Performance evaluation

To evaluate how SAGA and LRGA performs in generalize scenario, we compare it with two other algorithms, an exact mixed-integer solver and a naïve heuristic algorithm, the performance evaluation is conducted by incorporating three factors. Below we will describe the experiment setting in Section 5.1, explain the exact mixed-integer solver and the naïve heuristic algorithm in Section 5.2, and demonstrate the results of the performance evaluation in Section 5.3. A discussion about computation time of different algorithms is presented in Section 5.4.

5.1 Experiment Setting

Our experiments are grounded in the actual production environment of our industry partner. Because they plan on a one-day horizon, we set $L = 480$ to represent 480 minutes of available work time per period. We also choose $n_T = 60, n_S = 3, n_I = 25, n_G = 2, G = ((1, 2), (3))$. All the remaining parameters, such processing times, setup times, and space requirements—are elicited through consultation with our partner and then randomized (see Table 1). Here, the interval notation $[m, n]$ indicates that, during instance generation, integer values are sampled uniformly at random from the inclusive range $[m, n]$.

Parameter	Value
α	1.0
β	1.0
D_i	$[0, 10]$
R_{is}	$[120, 180]$
A_{is}	$[10, 30]$
K_{is}	$[2, 5]$
P_i	$[10, 20]$
Q_i	$[10, 20]$
H_g	$[20, 25]$

Table 1: Detailed parameters of the baseline instance.

To compare the impact of different factors: processing times, setup times, and space occupancy, we propose a basic benchmark scenario with all factors set to their normal levels and generate the extended $3 \times 2 = 6$ scenarios by setting one factor to its high or low extreme level and other factors to their normal level, i.e., 2-a indicates a scenario that orders' process time is lower than those in the baseline scenario, and 4-b shows a scenario that orders' space occupancy is higher than those in the baseline. The detailed setting of each scenario is listed in Table 2. For each scenario, we produce 40 randomized instances.

All experiments were run on a remote server with a 3.0 GHz Intel(R) Core i9-10980XE processor and 128 GB of RAM. The heuristic algorithms were implemented in Python 3.10, and the MILP model was solved using Gurobi Optimizer 12 via the OR-Tools Python package. To ensure a fair comparison, every algorithm was executed in single-core.

5.2 An exact mixed-integer solver and a naïve heuristic algorithm

To evaluate SAGA and LRGA algorithms, we benchmark them against the exact MILP solution—or, when optimality cannot be proven within the time limit, against the program's best dual bound. For each instance, we solve the MILP using Gurobi Optimizer with a 3,600 second time limit on a single CPU core. If Gurobi fails to prove optimality within this limit, we record its final dual bound as the theoretical lower bound for the minimization objective value.

To validate the benefit of our sorting strategies, we introduce a naïve baseline called Due Date–based Greedy Allocation (DDGA). DDGA uses the same greedy allocation (GA) procedure as SAGA and LRGA but sort the input sequence by ascending due dates, so that orders with the earliest deadlines are always processed first.

5.3 Performance of SAGA and LRGA

Table 3 presents the full results of our evaluation experiments. The values z^{DDGA}, z^{SAGA}, and z^{LRGA} denote the objective values produced by the DDGA, SAGA, and LRGA heuristics, respectively. A 3,600 second time limit was imposed on every algorithm run on all the instances.

Across all scenarios, SAGA achieves an average optimality gap of 15.78%, while LRGA's gap is 38.05%. Although these heuristics do not always reach optimality, they require far less computation time than the MILP solver (see subsection 5.4 more for details). In contrast, the DDGA baseline exhibits a 45.44% average gap, confirming the effectiveness of our sorting strategies.

We observe that in Scenario 4-b (Increased Space Occupancy), our heuristics exhibit a substantially larger optimality gap than in the other scenarios. For a representative instance in Scenario 4-b, where the gap reaches 53.75%. In the MILP solution, four orders—#5, #6, #2, and #0—are completed exactly at period 8. By contrast, SAGA delivers only three orders—#15, #10, and #5—on time. This difference stems from the greedy allocation procedure: SAGA allocates all the production capacity at the given period to the first few orders it encounters, causing subsequent orders to finish early or delay. Moreover, under tight storage space constraints, the GA must enforce additional feasibility checks, rendering it overly conservative and limiting its ability to process multiple orders concurrently. Nonetheless, the GA-based heuristics perform well in all other scenarios.

It is also interesting to compare the performance the trade-offs between LRGA and SAGA. Overall, LRGA achieves a smaller optimality gap than SAGA but incurs substantially higher runtime—its time complexity is on the order of $k_{\max} = 100$ times that of SAGA. In contrast, solving the linear relaxation of the MILP problem within SAGA adds only negligible overhead to its total compute time.

5.4 Computation time

To analyze the time complexity of SAGA and LRGA, let n_I be the number of orders, n_S the number of stages per order, and n_T the number of time periods. The core GA procedure allocates each stage of each order across all periods. We have three sub functions: `calculate_h` runs in $O(n_T)$; `check_residual` runs in $O(n_I n_S n_T)$; and `schedule_backward` runs in $O(n_I n_S^2 n_T^2)$.

Since `schedule_backward` is invoked for every order–period pair ($O(n_I\, n_T)$ times), the overall time complexity of the `GA` procedure is $O(n_I\, n_T) \times O\big(n_I\, n_S^2\, n_T^2\big) = O\big(n_I^2\, n_S^2\, n_T^3\big)$. Then the approximate time complexities for the heuristic algorithms are:

- **DDGA:** $O\big(n_I^2\, n_S^2\, n_T^3\big)$.
- **SAGA:** $O\big(k_{\max} n_I^2\, n_S^2\, n_T^3\big)$, since it executes GA procedure $k_{\max}$ times.
- **LRGA:** $O\big(n_I^2\, n_S^2\, n_T^3\big) + O(m^3)$, where m is the number of functional constraints—solving a general LP with m constraints by the simplex method scales as $O(m^3)$[3].

To empirically validate our time-complexity analysis and benchmark the heuristics against an exact MILP solver, we design a follow-up experiment in which only the order count n_I varies. Adopting the baseline parameters from Section 5.1, we test $n_I \in \{5, 10, 15, \ldots, 40\}$. For each value of n_I, we generate 20 random instances and run all algorithms, and set the computation time limit to 3,600 seconds on all the runs.

Scenario	Scenario name	R_{is}	A_{is}	K_{is}
1	Baseline	-	-	-
2-a	Reduced Process Time	$[60, 90]$	-	-
2-b	Increased Process Time	$[180, 270]$	-	-
3-a	No Setup Time	-	0	-
3-b	Increased Process Time	-	$[40, 120]$	-
4-a	No Space Occupancy	-	-	0
4-b	Increased Space Occupancy	-	-	$[4, 10]$

Table 2: The setting of different scenarios.

Scenario	Average Optimality Gap			Average Computing Time (seconds)			
	$\frac{z^{\text{DDGA}}}{z^{\text{MILP}}} - 1$	$\frac{z^{\text{SAGA}}}{z^{\text{MILP}}} - 1$	$\frac{z^{\text{LRGA}}}{z^{\text{MILP}}} - 1$	MILP	DDGA	SAGA	LRGA
1	25.85%	6.56%	27.59%	95.26	0.11	8.35	0.49
2-a	14.79%	0.67%	10.77%	0.65	0.09	9.27	0.55
2-b	43.35%	16.11%	21.84%	3600.00	0.24	21.56	0.78
3-a	15.02%	2.52%	6.04%	1.04	0.09	9.40	0.57
3-b	53.65%	18.47%	53.28%	3349.61	0.19	18.05	0.69
4-a	31.51%	6.30%	20.53%	377.90	0.10	10.31	0.57
4-b	133.92%	59.80%	126.29%	927.76	0.10	9.31	0.51
Average	45.44%	15.78%	38.05%	1193.17	0.13	12.32	0.59
Average without 4-b	30.69%	8.44%	23.34%				

Table 3: Evaluation result of all scenarios.

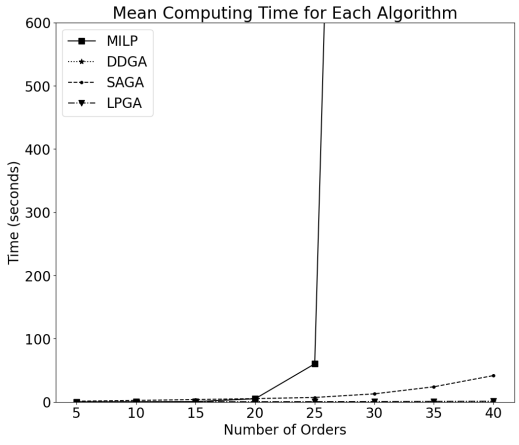

Figure 3: Mean computing time for each algorithm

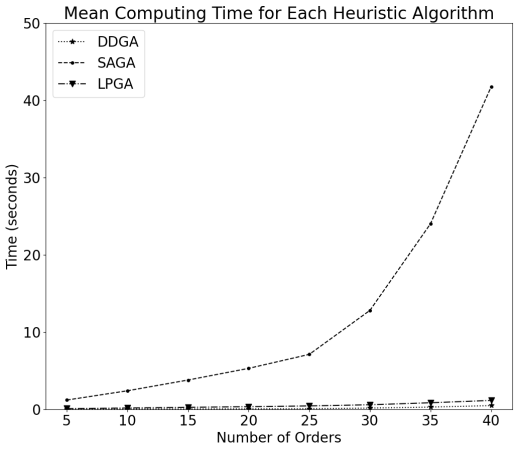

Figure 4: Mean computing time for each heuristic algorithm

Figure 3 confirms that the exact MILP solver's runtime grows exponentially (2^n), whereas all three heuristics scale in polynomial time. In Figure 4, SAGA is shown to be roughly 100× slower than DDGA and LRGA; nonetheless, even for $n = 40$, SAGA completes in under 10 minutes, which remains acceptable for daily planning. For larger order volumes, LRGA's superior scaling makes it the more efficient choice.

6 Case Study

To benchmark SAGA and LRGA against the manual planning procedures common in traditional manufacturing, we partnered with an aluminum window manufacturing company for a real-world case study. Section 6.1 provides an overview of the factory's operations. Section 6.2 describes the production data collected from our collaborator. Finally, Section 6.3 compares the schedules generated by our heuristics with the company's historical, manually planned production results.

6.1 Factory overview

The collaborating company specializes in producing high-quality aluminum windows for construction projects across Taiwan, with its main manufacturing facility located in Taichung. Given the large volumes they must deliver, precise scheduling is vital: early deliveries create extra handling and storage burdens on site, while late shipments disrupt overall construction timelines. Consequently, accurate production planning is critical to their business success.

The manufacturing process comprises five stages: cutting, drilling, assembly, mounting, and packaging. Although the factory also handles upstream and downstream tasks—such as raw-material procurement and final delivery—this case study focuses exclusively on in-factory operations.

In the cutting stage, the facility uses machines capable of both 45° and 90° angular cuts. For drilling, it employs two types of machines—air-pressure (pneumatic) and oil-pressure (hydraulic)—each designed to accommodate specific aluminum types. A detailed overview of the production flow is shown in Figure 5.

6.2 Data description

The collaborating company provided daily order records from October 2024 through mid-February 2025, covering 141 days. Each day's data is formatted as shown in Tables 8 and 9 in Appendix. However, the raw records omit several critical details—processing times for each window at every stage, which machine should each windows go through at stages with specification constraint, setup times, and how much space would one window occupy in the storage, although these factors apply in their manufacturing environment, they were not systematically documented. After cleaning and consolidating the data, we extracted 92 complete orders, including each order's, due date, actual delivery date, and the number of windows per order.

Based on interviews with our collaborating company, we adopt the following assumptions: each window requires 15 minutes of processing at every stage plus a 5-minute setup time per order; in the absence of precise data, we assume a 50% split between the two specification categories during cutting and drilling; and each window occupies 0.2 m^3 of storage, given approximately 40 m^3 of available space at each stage.

Thus, we reformatted the historical data to conform to the structure presented in Section 3 into one instance, for simplicity, we set $\alpha = 1$ and $\beta = 1$, further more $P_i = 1$ and $Q_i = 1$ for all orders i. We call this instance as the real-world instance.

6.3 Comparison with historical production plans

In Table 4, we compare SAGA and LRGA against manual planning on our real-world instance on the proportion on-time, early and tardy delivery. Both heuristics dramatically outperform the manual approach, achieving on-time delivery rates of 69.7% (SAGA) and 53.0% (LRGA) versus just 9.1% for manual planning. This improvement stems from our methods' ability to account for multiple order attributes simultaneously and schedule nearly 100 orders at once—even experienced planners might be overwhelmed.

However, the partner-provided dataset may not fully capture the real-world (e.g., some "working" days may in fact be national holidays, or machines may be offline for maintenance in some time period), which could cause our algorithms' performance to be overestimated. A more extensive validation under realistic operating scenarios is therefore required in future works.

7 Conclusion

In this study, we addressed a flexible flow shop scheduling problem with unrelated machines, sequence-dependent setup times, limited storage capacity, and specification constraints. We formulated the problem as a mixed-integer linear program (MILP) to generate production plans. Because solving this MILP is NP-hard, we developed a greedy allocation (GA) procedure and derived two heuristic algorithms—SAGA and LRGA—based on it. Our computational experiments demonstrate that these heuristics produce near-optimal solutions within acceptable computation time, enabling manufacturers to generate detailed daily schedules and make data-driven decisions.

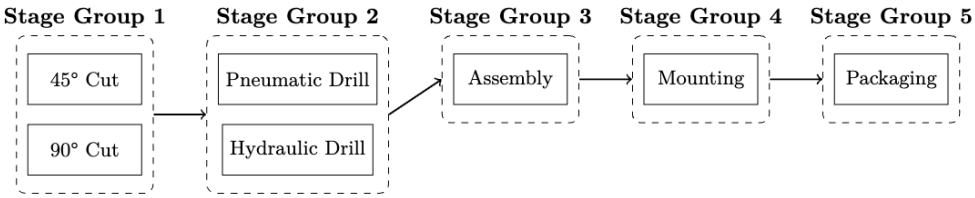

Figure 5: In-factory aluminum-window production flow, showing machine-level stage groups

Algorithm	On-time	Early	Tardy	Computing Time (seconds)
Manual Planning	9.1%	6.1%	84.8%	N/A
SAGA	69.7%	30.3%	0.0%	1342.91
LRGA	53.0%	47.0%	0.0%	225.98

Table 4: Comparison of solution performance and computation time for the real-world case study.

Our MILP model transforms specification constraints into multiple stages, which assumes that all products share the same constraint, a condition met by our industry partner. To generalize this approach, a more sophisticated formulation is needed to accommodate heterogeneous specification requirements. Future work will also focus on enhancing the GA's effectiveness in scenarios with severely constrained storage capacity, where its current performance degrades.

References

[1] D. Danneberg, T. Tautenhahn, and F. Werner. A comparison of heuristic algorithms for flow shop scheduling problems with setup times and limited batch size. *Math. Comput. Model.*, 29(9):101–126, May 1999.

[2] Henry Gabbay. Multi-Stage Production Planning. *Management Science*, 25(11):1138–1148, November 1979.

[3] F. Hillier and G. Lieberman. *Introduction to Operations Research*. McGraw Hill, USA, 10th edition, 2014.

[4] Muhammad Nawaz, E Emory Enscore Jr, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983.

[5] Yi Tan, Lars Mönch, and John W. Fowler. A hybrid scheduling approach for a two-stage flexible flow shop with batch processing machines. *J. of Scheduling*, 21(2):209–226, April 2018.

Appendix

Notation	Definition
T	The set of periods.
S	The set of stages.
I	The set of orders.
G	The set of stage groups.
Sg	The set of stages within stage group g.
δ	The set of final stages within each stage group.

Table 5: List of Sets

Notation	Definition
L	The maximum capacity that a stage can allocate in a single period.
R_{is}	The amount of capacity required to produce order i at stage s.
D_i	The period in which order i is expected to be completed.
K_{is}	Order storage space, representing the storage space required for order i at stage s.
H_g	The storage capacity limit for stage group g.
P_i	The earliness penalty for delivering order i early by one period.
Q_i	The tardiness penalty for delivering order i late by one period.
A_{is}	The cost of setting up stage s to produce order i.
M	A large constant.
α	Weight parameter for the earliness penalty in the objective function.
β	Weight parameter for the tardiness penalty in the objective function.

Table 6: List of Parameters

Notation	Definition
x_{tsi}	Capacity allocated at stage s for order i in period t.
h_{tsi}^{prev}	Completion ratio before stage s for order i at the beginning of period t. This represents the proportion of order i completed at stage s before period t begins.
h_{tsi}^{next}	Completion ratio after stage s for order i at the end of period t. This represents the proportion of order i completed at stage s by the end of period t. The completed order will then proceed to the next stage for continued production.
y_{tsi}	Binary decision variable for production at stage s for order i in period t. If the value is 1, it indicates that order i is being produced at stage s in period t.
c_{it}	Delivery decision variable for order i in period t. If the value is 1, it indicates that order i is delivered in period t.
z_i	The number of periods that order i is delivered early.
w_i	The number of periods that order i is delivered late.

Table 7: List of Decision Variables

Variable	Description	Data type
Date	The date of the record	Date
Warehouse Queue	The orders queued but not yet processed	List[Order]
Cutting Queue	The orders waiting for cutting	List[Order]
Inner Door Queue	The inner-door orders awaiting downstream processes	List[Order]
Sliding Window Queue	The sliding-window orders awaiting downstream processes	List[Order]
Pushing Window Queue	The pushing-window orders awaiting downstream processes	List[Order]

Table 8: Data format for each day's record

Variable	Description	Data type
Customer Name	Name of the customer	string
Finished	Whether the order was completed on that day	boolean
Due Date	Promised delivery date of the order	Date
Items	List of windows in the order, each as a tuple (window codename, quantity)	list[tuple(string, int)]

Table 9: Data format for each order