Job Shop Scheduling Challenge

TIG Labs - Karim Tamssaouet

January, 2026

1 Introduction

The Flexible Job Shop Scheduling Problem (FJSP) is a generalization of the classical Job Shop Problem (JSP) where operations can be processed on one of a set of eligible machines. This flexibility allows for better workload balancing and resource utilization but significantly increases the complexity of the problem, as it involves both assignment (routing) and sequencing decisions.

The FJSP is a relevant model for a wide variety of real-world manufacturing and service environments. Applications found in the literature include manufacturing (e.g., semiconductor, glass, pharmaceutical, printing) and service sectors such as healthcare and railway [1].

2 The TIG Challenge

Formally, the problem is defined by a set of n jobs $J = J_1, ..., J_n$ and a set of m machines $M = M_1, ..., M_m$. The problem is subject to the following standard assumptions:

1. Availability: All machines and jobs are available at time 0.
2. Non-preemption: Once an operation starts, it must be completed without interruption.
3. Machine capacity: Each machine can process at most one operation at a time.

The specific structure is defined as

- Each job J_i consists of a finite sequence of operations $O_{i,1}, O_{i,2}, O_{i,3}...$ that must be processed in the given order (precedence constraint).
- For each operation $O_{i,j}$, there is a set of eligible machines $\mathcal{M}_{i,j} \subset M$.
- Each operation is assigned to ate least one machine, i.e. $\mathcal{M}_{i,j} \neq \varnothing$.
- The processing time of an operation $O_{i,j}$ depends on the selected machine.

The objective is to minimize the **Makespan** ($C_{\max}$) :

$$\min C_{\max} = \min \left(\max_{1 \leq i \leq n} C_i \right) \tag{1}$$

where C_i is the completion time of the last operation of job J_i.

While the makespan is the most extensively studied criterion in scheduling literature, it is recognized that it is not often the most relevant metric in practical settings [1]. However, due to its dominance in academic research and the availability of comparable benchmarks, it remains the primary metric for this challenge.

3 The Approach to Instance Generation

3.1 Background: Academic Benchmarks

To evaluate solution approaches, researchers have developed various sets of benchmark instances over the decades [2, 1]. Well-known sets include those by [3] and [4], which extend classical job shop instances by introducing different levels of operation flexibility. Additionally, [5] generated diverse instances by varying flexibility levels and incorporating features such as reentrant flows and machine-dependent processing times. While these benchmarks have driven algorithmic development, they often lack the structural diversity required to model the broad range of practical industrial settings.

3.2 Method: Parameter-driven generation

To address the limitations of existing benchmarks, the TIG Challenge utilizes a generation method that simulates the definition of a factory's capabilities and product demand. Using a set of controllable parameters, this method constructs instances layer by layer, from the physical shop floor to the specific product recipes and orders.

3.3 Layer 1: Shop floor (Capabilities)

The foundation of the instance is the physical capability of the factory. A vocabulary of abstract operation types (e.g., drilling, cutting, painting) is created. For each operation type, a subset of machines is marked as eligible based on a flexibility parameter. This setup mimics real-world constraints where certain tasks can only be performed by specific qualified resources. A base processing time is also generated using a uniform distribution between 1 and 200, as in [6].

3.4 Layer 2: Routes (Structure)

Routes define the abstract sequence of operation types that products must follow. The complexity of these sequences is controlled by the flow structure parameter. A value of 0.0 generates a strict Flow Shop (a unique linear sequence), while a value of 1.0 generates a Job Shop (different random permutation). Intermediate values create hybrid structures. Additionally, a reentrance parameter controls the probability of a route looping back to a previously used operation type, modeling cyclic flows common in different settings such as semiconductor fabrication.

3.5 Layer 3: Product specifications (Recipes)

A Product is a specific instantiation of a Route. For each step in a product's route, a base processing time is already defined. To capture realistic variations, a machine speed variability parameter is

applied. Depending on the setting, this parameter serves different objectives: it differentiates between products that share the same route (creating unique recipes) and differentiates between the performance of various machines (modeling speed/efficiency differences).

3.6 Layer 4: Job generation (Demand)

Finally, jobs are instantiated as orders for specific products. The variety of these orders is controlled by the product mix ratio. To ensure the structural complexity of the routing is mean ingful, the value of this parameter is set to be at least equal to the flow structure parameter. This ensures that environments with complex, random routings (high flow structure) also have a sufficiently high variety of product recipes.

4 Implementation of Instance Generation

4.1 The Setup

An instance is defined by the following parameters that control the size of the instance:

- **The number of Jobs** – n
- **The number of Machines** – m
- **The number of Operation Types** – q

Currently we fix these at 50 jobs, 30 machines, and 30 operation types. There is no strict requirement for these values to be consistent across all tracks. In the future they can be customized to suit the specific problem track (Section 6). The resulting instances (1500–2000 operations) are considered medium-to-large compared to existing JSP and FJSP benchmarks. While standard benchmarks can have fewer operations, [7] proposed instances with up to 2000 operations.
Define:

- The jobs: $J = \{J_1, \ldots, J_n\}$
- The machines: $M = \{M_1, \ldots, M_m\}$
- The operation types: $T = \{T_1, \ldots, T_q\}$

As seen in the construction below the operation types get a base processing time sampled from $[1, 2, \ldots, 200]$, there is no strong justification for this specific interval; other benchmarks use different ranges (e.g., [7] uses $[1, 2, \ldots, 99]$ and [5] use $[10, 11, \ldots, 100]$). The most effective way to validate the difficulty of a particular choice is to benchmark a CP solver against a sample of generated instances.

The following four main parameters play a key role in defining the type of problem being generated, they are set by the track (see Section 6)

- `Flexibility` – controls how many machines can process each operation type.
- `Flow structure` – controls how similar or different the routes are across jobs.

- `Product mix ratio` – controls how many distinct products exist relative to the number of jobs.
- `Re-entrance level` – controls how often routes revisit operation types.

We use pseudo code to describe the generation of a challenge instance.

4.2 Part 1: Shop Floor (Assigning Operation Types to Machines)

We define the shop floor by assigning each operation type a set of compatible machines. This happens in two phases, first the base machines are assigned and then extra machines are assigned. We also assign to each operation a base processing time.

Phase 1: Base Machine Assignment (Ensure At Least One Machine per Operation Type)

- Operation types (up to the number of machines), machines are assigned from a shuffled list, so each operation type gets a unique base machine when possible.
- If there are more operation types than machines, any remaining operation types are assigned a base machine chosen at random from all machines.
- Each Operation Type is assigned a base processing time uniformly from $[1, 2, 3, \ldots, 200]$.

```
# INPUTS:
# T = list of operation types, size = q
# M = list of machines, size = m
# OUTPUT:
# base_machine[T] : a single base machine for each operation type
# eligible[T] : set of eligible machines for each operation type (initially base only)
# base_processing_time[T] : a single base processing time for each operation type.

shuffle(M) # random permutation

for i in 1..q:
    if i <= m:
        base_machine[T[i]] = M[i] # unique base machine when possible
    else:
        base_machine[T[i]] = sample_uniform(M) # if more types than machines

    eligible[T[i]] = { base_machine[T[i]] } # start compatibility set with base machine

    base_processing_time[T[i]] = UniformInteger(1, 200) #
```

Phase 2: Adding Flexibility (Extra Compatible Machines)

- For each operation type, a *target total* number of compatible machines is sampled from a normal distribution centered at the global *flexibility* parameter.

- Since each operation type already has one base machine from Phase 1, the number of *extra* machines to add is the sampled total minus 1 (clipped to be nonnegative).
- Additional machines are then selected uniformly at random from those not yet assigned to that operation type, until the target total is reached (or all machines are used).

Here, `target_total[t]` is the desired *total* number of eligible machines for operation type t (including the base machine), and

$$\texttt{target\_extra[t]} = \max(0, \texttt{target\_total[t]} - 1)$$

is the number of *extra* machines added beyond the base.

```
# INPUTS:
# flexibility (global), sigma = 0.5
# OUTPUT:
# eligible[t] updated to include extra machines

for each operation type t in T:

    # Sample TOTAL number of eligible machines (including the base machine)
    target_total[t] = abs( floor( Normal(flexibility, 0.5) ) )

    # Ensure at least 1 total (the base machine) and at most m total
    target_total[t] = max(target_total[t], 1)
    target_total[t] = min(target_total[t], m)

    # Convert to number of EXTRA machines to add beyond base
    target_extra[t] = target_total[t] - 1 # in [0, m-1]

    remaining = M \ eligible[t] # machines not already eligible

    k = min(target_extra[t], size(remaining))
    extras = sample_without_replacement(remaining, k)

    eligible[t] = eligible[t] U extras
```

4.3 Part 2: Routes

Routes define the abstract sequence of operation types that products must follow. Routes are created next, based on the *flow structure* and *re-entrance* parameters.

Number of Routes

The *flow structure* parameter determines how many distinct routes are created:

```
num_routes = max(1, int(flow_structure * n))
```

A low flow structure (close to 0) leads to few routes (approaching a flow shop), whereas a high flow structure (close to 1) leads to many distinct routes (approaching a pure job shop).

Base Routes (Before Re-entrance)

For each route:

1. Start from a list containing all operation types.
2. Randomly permute this list to obtain a sequence of length #op_types.

The result is that each route contains all operation types, but in random order.

```
# OUTPUT:
# routes[i] is a list of operation types (before re-entrance)

for i in 1..num_routes:
    routes[i] = copy(T)
    shuffle(routes[i]) # permutation of all operation types
```

For Flow Shop and Hybrid Flow Shop environments, it is standard for all jobs to include all operation types. While this is not always true for practical Job Shop environments, most academic JSP benchmarks assume each job visits every machine. At this stage of development we adhere to standard academic convention of including all operation types in each route, however this is something that could be introduced via the a new track in the future.

Re-entrance (Adding Repeated Operations)

Re-entrance is then applied to each route according to the *re-entrance level*. The following process is applied to each route:

- We scan the route from the second position onwards (step index ≥ 2).
- We continue scanning until we reach the *current end* of the route (so insertions can increase the total number of scan steps).
- At each position:
  - With probability reentrance_level, we insert a duplicate operation at this position.
  - The duplicate operation is chosen uniformly at random from the operations that appeared earlier in the route (before the current position). Repeats do not increase selection probability.
  - The chosen duplicate is inserted at the current position, and all subsequent operations are shifted one position forward.

This process creates loops in the route, where a job may return to stages it has already visited.

```
# INPUT:
# reentrance_level in [0,1]
# OUTPUT:
# routes updated in-place (route length may increase)

for each route r in routes:
```

```
j = 2
while j <= length(r): # IMPORTANT: current end, not original end

    if Bernoulli(reentrance_level) == 1:

        seen_types = distinct_set( r[1..(j-1)] ) # distinct types only
        dup_type = sample_uniform(seen_types)

        insert r at position j with dup_type # shifts r[j..end] right by 1

    j = j + 1
```

4.4 Part 3: Products

A Product is a specific instantiation of a Route. Products are defined based on the `product_mix_ratio` $\in$ [`flow_structure`, 1] and the routes from Part 2.

Number of Products

The number of distinct products is defined by the number of jobs and the product mix ratio

```
num_products = max(1, int(product_mix_ratio * n))
```

Creating Each Product

A. Route Assignment

- Each product is assigned a unique route until all routes are exhausted.
- Any remaining products are then assigned a route sampled uniformly at random from all routes.

```
INPUTS:
num_products
routes : list of available routes, size = num_routes
OUTPUT:
product_route[p] : assigned route for each product

shuffle(routes) # random permutation of routes

num_routes = size(routes)

for p in 1..num_products:
if p <= num_routes:
product_route[p] = routes[p] # unique route when possible
else:
product_route[p] = sample_uniform(routes) # reuse routes after exhaustion
```

B. Generating Processing Times (the "Recipe") We treat each position in the product route as a distinct *operation instance*. Even if two instances have the same operation type (e.g. due to re-entrance), they may have different processing times. This is a deliberate design choice. To define the processing times for each operation in the product we do:

1. Look up the operation type and its set of eligible machines (from Part 1).
2. For each eligible machine:
   - Sample a speed factor from
   $$\text{uniform}(\text{MIN\_SPEED\_FACTOR}, \text{MAX\_SPEED\_FACTOR})$$
   (e.g. between 0.8 and 1.2).
   - Compute the processing time as
   $$p_{\text{time}} = \max\big(1,\ \text{int}(\text{base\_processing\_time} \times \text{speed\_factor})\big).$$
   - Store this as step_data[machine_id] $= p_{\text{time}}$.

```
# INPUTS:
# base_processing_time[type]
# eligible[type] = set of machines compatible with that type
# min_speed_factor = 0.8, max_speed_factor = 1.2
# OUTPUT:
# recipe[p][op_index][machine] = processing time

for p in 1..num_products:

    r = product_route[p] # list of operation TYPES, after re-entrance
    recipe[p] = empty_map()

    for k in 1..length(r): # k identifies the OPERATION INSTANCE in this product

        t = r[k] # operation type at position k

        for each machine m in eligible[t]:

            speed_factor = Uniform(min_speed_factor, max_speed_factor)
            p_time = int( base_processing_time[t] * speed_factor )
            p_time = max(1, p_time)
            p_time = min(p_time, 200)

            recipe[p][k][m] = p_time

    product[p] = (route = r, recipe = recipe[p])
```

4.5 Part 4: Job Creation (Demand)

Finally, we generate the actual jobs (demands). Each job is a demand for one of the products and inherits that product's route and recipe.

Job Generation

For each job we create a `Job` object with:

- a unique job ID `j_id`, and
- a reference product chosen according to the assignment rule below.

What Each Job Contains

Each job inherits from its product:

- the route (sequence of operation types; each position in the route is treated as an operation instance), and
- the recipe (processing times for each operation on each eligible machine).

Thus, each job is a single demand instance for a particular product type.

Distribution of Jobs Over Products

- The first `num_products` jobs are assigned *deterministically*: job i is assigned to product i for $i = 1, \ldots,$ `num_products`. This guarantees that every product appears at least once.
- The remaining (`num_jobs` − `num_products`) jobs are assigned by sampling products independently and uniformly at random (with replacement).
- Since product_mix_ratio < 1.0, there are fewer products than jobs, so many jobs share the same product definition, creating clusters of jobs with identical route and processing-time structure.

```
INPUTS:
products[1..num_products]
num_jobs
OUTPUT:
jobs[i] = (job_id, chosen_product, route, recipe)

for i in 1..num_jobs:

if i <= num_products:
    chosen_product = products[i] # deterministic coverage
else:
    chosen_product = sample_uniform(products) # random remainder

jobs[i].job_id = i
jobs[i].product = chosen_product
jobs[i].route = chosen_product.route
jobs[i].recipe = chosen_product.recipe
```

5 Asymmetric Verification

Verification is computationally trivial: given a proposed schedule, checking all constraints (machine eligibility, precedence, and machine capacity) requires only simple comparisons and sorting operations, which run in polynomial time $O(n \log n)$, where n is the number of operations.

Solving, by contrast, is NP-hard: finding an optimal (or even good) schedule requires searching an exponentially large solution space, as each operation must be assigned both a machine (from its eligible set) and a start time, with choices for one operation affecting feasibility and quality of all subsequent decisions.

5.1 Solution Format

A valid solution consists of a schedule for all jobs:

- `job_schedule`: a vector of schedules, one per job.
- `job_schedule[i]` contains the schedule for job i.
- Each job schedule is a sequence of pairs `(machine_id, start_time)`.
- There is exactly one such pair for each operation in the job, in route order.

5.2 Verification Procedure

The solution verification procedure (`evaluate_makespan`) checks the following constraints in order.

1. Job Count Validation

- The solution must contain schedules for exactly `num_jobs` jobs.
- **Violation**: an error is returned if the job count does not match.

2. Product-Specific Operation Count

- Each job i is associated with a product that defines a fixed route length.
- `job_schedule[i]` must contain exactly one scheduled operation for each operation instance in that product's route.
- **Violation**: an error is returned if the operation count does not match the product's route length.

3. Machine Eligibility

- Each operation instance k of a job can only be performed on a specified set of eligible machines.
- The assigned `machine_id` for operation k must belong to this eligible set.
- Eligible machines and processing times are defined in `product.recipe[k][machine_id]`.
- **Violation**: an error is returned if an ineligible machine is assigned.

4. Precedence Constraints (Job-Level)

- Operations within a job must respect their sequential order.
- Operation k may only start after operation $k-1$ has completed.
- Formally,
$$\text{start\_time}(k) \;\geq\; \text{start\_time}(k-1) + \text{processing\_time}(k-1),$$
where processing times are taken from the job's inherited recipe.
- **Violation**: an error is returned if an operation starts before the previous one completes.

5. Machine Capacity Constraints

- Each machine can process at most one operation at a time.
- No two operations assigned to the same `machine_id` may overlap in time.
- For any two operations on the same machine with intervals $[s_1, f_1)$ and $[s_2, f_2)$, one of the following must hold:
$$s_2 \geq f_1 \quad \text{or} \quad s_1 \geq f_2.$$
- Verification method:
  - Collect all operations assigned to each `machine_id`.
  - Sort them by `start_time`.
  - Check that consecutive operations do not overlap.
- **Violation**: an error is returned if overlapping operations are detected on any machine.

5.3 Solution Quality Calculation

If all constraints are satisfied, the **makespan** is calculated as the maximum completion time over all jobs.

Solution quality is measured using the baseline score. Let C_{base} denote the baseline makespan and C the submitted solution's makespan. The quality score is:

$$\text{better\_than\_baseline} = \frac{C_{\text{base}} - C}{C_{\text{base}}}.$$

Higher scores correspond to stronger performance relative to the baseline. This incentivizes meaningful algorithmic innovation and consistent performance improvements. The baseline algorithm is based on dispatching rules, selecting operations by Most Work Remaining and assigning machines via Earliest End Time.

6 Challenge Tracks

The challenge comprises five tracks, each representing a specific special case of the Flexible Job Shop Scheduling Problem (FJSP). Across all tracks, the following parameters are fixed: the number of jobs is set to 50, the number of machines to 30, the number of operation types to 30, the base processing times are generated uniformly in the range $[1, 200]$, and the machine speed factors are drawn uniformly from the interval $[0.8, 1.2]$.

Track	**Flow Structure**	**Flexibility**	**Mix Ratio**	**Reentrance**
1. Flow Shop	0.0	1.0	0.5	0.2
2. Hybrid Flow Shop	0.0	3.0	0.5	0.2
3. Job Shop	0.4	1.0	1.0	0.0
4. FJSP (Medium)	0.4	3.0	1.0	0.2
5. FJSP (High)	1.0	10.0	1.0	0.0

Table 1: Parameter configurations for the five challenge tracks.

Table 1 details the five challenge tracks, each designed to test specific algorithmic capabilities through distinct parameter combinations. Track 1 (Flow Shop) and Track 2 (Hybrid Flow Shop) both enforce a rigid, unidirectional flow structure (0.0) with low product mix (0.5), but differ in flexibility: Track 1 allows only one machine per operation, testing basic sequencing decisions, while Track 2 increases flexibility to 3.0, requiring effective load balancing across parallel machines.

Track 3 (Job Shop) introduces routing complexity through a moderate flow structure value (0.4) and a high product mix (1.0), while maintaining strict machine assignments (flexibility 1.0), thereby focusing primarily on sequencing under diverse routing constraints. Track 4 (Flexible Job Shop – Medium) represents a more realistic manufacturing environment, combining moderate flow structure (0.4) and flexibility (3.0) and allowing for re-entrant flows. Finally, Track 5 (Flexible Job Shop – High) simulates a highly chaotic setting with a completely random flow structure (1.0) and extreme flexibility (10.0), shifting the dominant challenge from sequencing to optimal load balancing across a vast and complex search space.

References

[1] Stéphane Dauzère-Pérès et al. "The flexible job shop scheduling problem: A review". In: *European Journal of Operational Research* 314.2 (2024), pp. 409–432.

[2] Dennis Behnke and Martin Josef Geiger. "Test instances for the flexible job shop scheduling problem with work centers". In: (2012).

[3] Paolo Brandimarte. "Routing and scheduling in a flexible job shop by tabu search". In: *Annals of Operations research* 41.3 (1993), pp. 157–183.

[4] Johann Hurink, Bernd Jurisch, and Monika Thole. "Tabu search for the job-shop scheduling problem with multi-purpose machines". In: *Operations-Research-Spektrum* 15.4 (1994), pp. 205–215.

[5] Stéphane Dauzère-Pérès and Jan Paulli. "An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search". In: *Annals of Operations research* 70.0 (1997), pp. 281–306.

[6] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. "Benchmarks for shop scheduling problems". In: *European Journal of Operational Research* 109.1 (1998), pp. 137–141.

[7] Eric Taillard. "Benchmarks for basic scheduling problems". In: *european journal of operational research* 64.2 (1993), pp. 278–285.