Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Benchmarking LLM-generated heuristics for the multi-objective flexible job shop: an anchored comparison against exact and metaheuristic methods

Alejandro Uribe
auribev1@eafit.edu.co
Universidad EAFIT
Diego Fornies-Tabuenca
Vicomtech Foundation
Urtzi Otamendi
Vicomtech Foundation
Arkaitz Artetxe
Vicomtech Foundation
Juan Carlos Rivera
Universidad EAFIT

Research Article

Keywords: Flexible job shop scheduling, Automated heuristic design, Largelanguage models, Multi-objective optimisation

Posted Date: November 13th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11359760/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Benchmarking LLM-generated heuristics for the multi-objective flexible job shop: an anchored comparison against exact and metaheuristic methods

Alejandro Uribe[1,2*], Diego Fornies-Tabuenca[1,3], Urtzi Otamendi[1,3], Arkaitz Artetxe[1], Juan Carlos Rivera[2]

[1*]Vicomtech Foundation, Basque Research and Technology Alliance (BRTA), Donostia-San Sebastián, 20009, Spain.
[2]School of Applied Sciences and Engineering, Universidad EAFIT, Carrera 49 # 7 Sur-50, Medellín, 05001, Antioquia, Colombia.
[3]Department of Computer Sciences and Artificial Intelligence, University of the Basque Country (UPV/EHU), Manuel Lardizabal Pasealekua 1, Donostia-San Sebastián, 20018, Gipuzkoa, Spain.

*Corresponding author(s). E-mail(s): auribev1@eafit.edu.co, auribe@vicomtech.org;
Contributing authors: diegoforniestabuenca@gmail.com; uotamendi@vicomtech.org; aartetxe@vicomtech.org; jrivera6@eafit.edu.co;

Abstract

Scheduling in flexible job shops faces a trade-off between exact methods, which provide high-quality solutions at the cost of problem-specific modelling and substantial computation, and dispatching rules, which are fast but often less effective. This paper investigates whether Large Language Models (LLMs) can automatically generate executable scheduling heuristics that narrow this gap. Two frontier LLMs from different providers are used as generators and variation operators within a Non-dominated Sorting Genetic Algorithm II (NSGA-II) framework optimising makespan and total machine workload. The generated heuristics are evaluated on the Brandimarte instances against a constraint-programming solver, a multi-objective memetic algorithm, and classical dispatching rules using a common Pareto-based protocol, with training

and evaluation instances kept disjoint. Since an individual heuristic produces a single schedule while the exact solver, run once per weight of the protocol, returns a set of optimal points, comparison is performed at the portfolio level. The generated portfolio matches the memetic algorithm while requiring three to five times less computation, and outperforms classical dispatching rules under an equivalent wall-clock budget on 11 of the 15 instances ($\boldsymbol{p = 0.036}$ after Holm correction). The approach requires only 54 lines of natural-language problem specification, compared with 221 lines for the corresponding disjunctive formulation, and the generated heuristics transfer to a new objective. The two models produce results within 1.3% of each other, while most of the gain from evolutionary search appears in the first two or three generations on the instances evaluated here, improving the best heuristic by approximately 2.5% in total.

Keywords: Flexible job shop scheduling, Automated heuristic design, Large language models, Multi-objective optimisation

1 Introduction

The flexible job shop scheduling problem (FJSSP) requires deciding which of several eligible machines processes each operation, and in what order operations are sequenced on each machine [1]. Practitioners choose between method families with sharply different profiles: exact formulations, based on mixed-integer linear programming (MILP) or constraint programming (CP), return schedules of certified or near-certified quality but require problem-specific modelling and runtimes of minutes to hours, while priority dispatching rules return a schedule in milliseconds at markedly worse quality and are difficult to control when several criteria compete.

Covering the territory between these extremes has traditionally meant designing heuristics and metaheuristics by hand, which demands substantial domain expertise for modelling, algorithm design and parameter tuning, and limits transferability to new problems [2, 3]. Large Language Models (LLMs) make automating that step plausible: drawing on pre-trained knowledge rather than manual algorithm design, they can write and iteratively improve executable heuristics inside a search loop [4–6]. Evolutionary algorithms are a natural host, supplying what LLMs lack on their own: a measured, repeatable criterion by which every generated candidate is accepted or discarded [7, 8].

How good the resulting heuristics are, in terms an operations researcher would accept, remains largely unestablished. Published evaluations compare generated heuristics against other generated heuristics, handcrafted priority rules or neural constructive baselines [9, 10]; anchored comparisons against exact mathematical programming are scarce, and where they exist they concern single-machine, single-objective problems [11]. Where multiple objectives appear, they are often properties of the heuristic itself, such as solution quality traded against runtime [12], rather than competing criteria of the decision problem. That distinction matters most in industrial scheduling, where conflicting criteria must be managed explicitly rather than reduced to a single aggregated metric [1].

A meaningful comparison between automatically generated heuristics and established scheduling methods requires three conditions. First, the handcrafted baselines must be competitive; otherwise, improvements may simply reflect weak reference methods. Second, the objectives must exhibit genuine conflict so that the problem remains meaningfully multi-objective. Third, all methods must be compared under a common protocol, since an individual heuristic produces a single schedule whereas an exact solver run under a weighted protocol produces a set of optimal points. This study explicitly addresses these three issues and reports the measurements supporting each choice.

The heuristic-generation framework is used primarily as an instrument for evaluating these questions rather than as the main contribution. The study therefore focuses on four aspects:

- Benchmarking automatically generated heuristics against constraint programming, a multi-objective memetic algorithm, and strengthened classical dispatching rules under a common Pareto-based protocol.
- Quantifying modelling effort and testing whether generated heuristics can be transferred to a new objective.
- Isolating the contribution of evolutionary search by analysing performance across generations.
- Comparing two current language models as heuristic generators while separately reporting generation and execution costs.

The remainder of the paper is organised as follows. Section 2 reviews related work and Section 3 presents the generation framework. Section 4 describes the experimental setup and the baselines, and Section 5 reports the results. Section 6 concludes and states the limitations of the study.

2 State of the Art

Exact formulations based on Mixed Integer Linear Programming and Branch and Bound have been extensively studied for the FJSSP [13, 14]. The formulation used here derives from Manne's disjunctive model of the job shop [15], extended to routing flexibility by [13], which combines binary operation–machine assignment variables with pairwise precedence variables on each machine, linearised through big-M constraints; it is distinct from the sequence-position formulations in the tradition of Wagner, such as [16], and the two families are compared in [17]. Constraint Programming has emerged as a competitive alternative, performing strongly on medium-sized instances [18, 19], with interval-variable models [20] and lazy-clause-generation solvers [21] now reaching industrial sizes [22, 23].

Dispatching rules have long been valued for their computational efficiency and simplicity [24], and problem-specific heuristics improve on them by exploiting structural properties of the problem [25]. Metaheuristics extend this landscape with greater generality: trajectory-based methods such as Simulated Annealing, Tabu Search and Variable Neighbourhood Search, and population-based methods including Genetic Algorithms [26] and discrete Particle Swarm Optimization variants [27].

Hybrid population-and-local-search schemes remain the reference point against which a new heuristic for the FJSSP is judged [3, 26], which is why one is included among the baselines of this study rather than only exact methods and a fast rule.

Learning-based schedulers train a policy to construct schedules sequentially. Ho et al. [28] reduce the decision space with a graph neural network, and Echeverria et al. [29] train on CP-generated solutions and embed CP within the reinforcement learning loop; earlier work established the formulations these build on [30, 31]. All report competitive makespan results for single-objective minimisation on a subset of the benchmark instances, and all require retraining when the problem specification changes.

Designing heuristics automatically is not new: generation hyper-heuristics, and genetic programming in particular, have long evolved dispatching rules for job shop and flexible job shop scheduling [32–34], although over terminal sets that an expert must define in advance. LLMs enter this landscape from a different direction. They have been applied across the optimisation pipeline rather than only to heuristic design: as search operators embedded within an optimisation process, as designers of the algorithm itself, and as orchestration agents that interpret high-level objectives and generate executable workflows [8, 35–37]. Hybrid frameworks combining LLMs with classical solvers report gains in heuristic refinement, parameter tuning and adaptive strategy selection [9, 10]. End-to-end prompt-based solvers, which map problem descriptions directly to solutions, continue to face difficulties with scalability and constraint satisfaction [38, 39], which is what motivates embedding the model within an algorithmic framework that enforces feasibility rather than using it as a solver in its own right.

Within that pipeline, the line of work closest to this study embeds LLMs in performance-driven feedback loops over executable heuristics. AEL (Algorithm Evolution using a Large Language Model), LLaMEA (Large Language Model Evolutionary Algorithm) and LLM-GP (LLM-based genetic programming) iteratively mutate and refine heuristics or programs on the basis of empirical results [40–42], while other approaches integrate the model directly as an evolutionary operator governing exploration, exploitation or diversity [43–46]. Low-intervention frameworks add reflective feedback, allowing heuristics to be revised on the basis of past performance [6, 47], and in scheduling specifically, generated dispatching policies have been combined with evolutionary refinement for dynamic environments [48, 49].

Çetinkaya et al. [11] evaluate LLM-discovered heuristics for single-machine total tardiness against a mixed-integer programming formulation, establishing that an anchored comparison is feasible in a single-objective, single-machine setting. Extensions to multiple objectives have taken two forms. One treats the objectives of the decision problem itself [50]. The other treats properties of the generated heuristic: Yao et al. [12] evolve heuristics under a bi-objective criterion trading solution quality against the efficiency of the heuristic itself, while Huang et al. [51] use an LLM to design the operators of a multi-objective evolutionary algorithm.

Taken together, the literature demonstrates the potential of LLMs to generate heuristics for combinatorial optimisation, but the evidence remains difficult to compare across studies. Existing evaluations typically emphasise solution quality, while

differences in baselines, computational budgets, objective formulations, and generation costs can substantially affect the conclusions. In particular, it remains unclear how generated heuristics compare with strong conventional methods when evaluated under a common multi-objective protocol and comparable computational budgets, or how much modelling effort is actually replaced by automated generation.

This study addresses this gap through a controlled empirical evaluation of LLM-generated heuristics for the FJSSP. The evaluation is deliberately designed to separate three dimensions that are often considered independently: optimisation performance, computational and generation cost, and modelling effort. Generated heuristics are therefore evaluated against exact optimisation, a multi-objective memetic algorithm, and computationally matched classical rules. The study also verifies that the selected scheduling objectives exhibit genuine conflict and examines whether generated heuristics can be transferred to a different objective.

The resulting contribution is not a new optimisation architecture, but a systematic assessment of the practical value and limitations of LLM-generated heuristics under controlled and reproducible conditions. This perspective complements existing work by moving beyond the question of whether LLMs can generate working heuristics toward the more consequential question of "when the resulting automation provides a meaningful advantage over conventional heuristic development".

3 Methodology

The framework embeds LLM-generated heuristics in a multi-objective evolutionary algorithm based on NSGA-II, which supplies the selection and diversity-preservation machinery, and confines the language model to the generation and modification of heuristic source code.

3.1 Problem definition and objectives

The FJSSP generalises the classical job shop by allowing each operation to be processed on any machine from a set of eligible alternatives [1, 52]. Let O be the set of operations and R the set of machines. For an operation $i \in O$, let $R_i \subseteq R$ be the machines able to process it and $p_{m,i}$ the processing time of i on machine $m \in R_i$. Let $\alpha_{m,i} \in \{0, 1\}$ indicate that i is assigned to m, and $t_i \geq 0$ the start time of i. A schedule assigns every operation to exactly one eligible machine and sequences the operations on each machine, respecting job precedence and machine capacity. Two objectives are minimised, the makespan

$$c_{\max} = \max_{i \in O} \Big(t_i + \sum_{m \in R_i} p_{m,i}\, \alpha_{m,i} \Big), \tag{1}$$

and the total workload consumed across all machines,

$$W = \sum_{i \in O} \sum_{m \in R_i} p_{m,i}\, \alpha_{m,i}. \tag{2}$$

The two conflict by construction: assigning every operation to its fastest eligible machine minimises W but concentrates work on those machines and inflates $c_{\max}$, while spreading work onto slower machines relieves the contention at the cost of consuming more machine time.

3.2 Overview

Figure 1 outlines the workflow. The framework is run twice, once with each of the two language models compared in this study, `claude-sonnet-5` and `gpt-5.1-codex`; everything described below is identical in the two campaigns. An initial population of executable heuristics is generated by prompting the LLM once per individual, each prompt carrying a different technical direction. Every heuristic is executed in a sandbox on a set of training instances, producing either a validated schedule per instance or a typed failure. Each objective value is divided by the lower bound of its own instance, and the resulting ratios are averaged over the instances, so that every heuristic is summarised by a two-component fitness vector: one component per objective, each expressed in multiples of the minimum attainable value. Individuals are ranked by Pareto dominance and crowding distance. The population is then partitioned into its non-dominated front and the remainder, and the model is prompted to describe what distinguishes the two; that observation is consolidated with the memory $\mathcal{R}_{t-1}$ carried over from previous generations into a single reflection $\mathcal{R}_t$. Offspring are produced by LLM-guided crossover and mutation, both drawing their parents from the ranked population by binary tournament: crossover additionally receives the reflection, while mutation receives a specific structural perturbation to apply. Parents and offspring compete together in a $\mu + \lambda$ environmental selection, and the cycle repeats for a fixed number of generations, after which a portfolio of the k best-ranked heuristics is extracted for evaluation on the held-out instances.

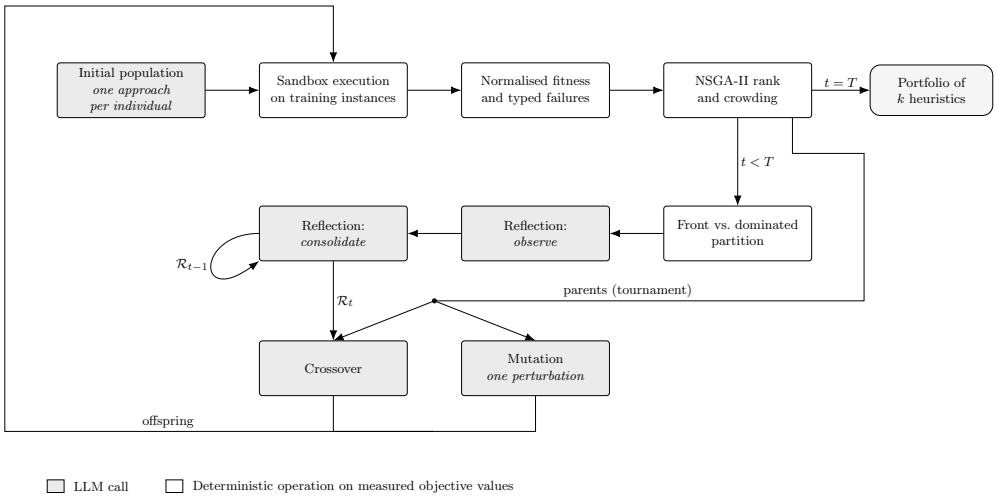


Fig. 1: Workflow of the heuristic generation framework.

3.3 Heuristic representation and evaluation

Each individual is an executable heuristic encoded as Python source code generated by the LLM, following the representation used in LLM-based evolutionary program search [5, 6]. No predefined template is imposed, which allows generated heuristics and handcrafted rules to be evaluated in the same problem environment.

The interface is fixed by an input–output contract, a fixed block of every prompt that states what the heuristic receives, what it must return and what makes a schedule acceptable and reproduced in full in Appendix A. A heuristic is a single function of one argument, whose input describes the instance (the number of jobs and machines and, for each job, its operations in precedence order with their eligible machines and processing times), and whose output is a schedule giving, for every operation, the assigned machine, its start time and the processing time used. The decision the heuristic makes is therefore the assignment and the start time of every operation; the completion time follows from the two, and is nonetheless requested explicitly so that the validator can check the three values for consistency. A schedule is rejected unless no operation is missing, every assignment is to an eligible machine, every duration matches the instance, job precedences hold and no two operations overlap on a machine. Feasibility is therefore checked by the framework rather than trusted to the model.

The contract also carries a declared time budget that the heuristic is required to respect by returning the best schedule found when it expires, while the sandbox independently enforces a hard cut-off that the model does not see. Both are needed: with only the hard cut-off, an improvement heuristic, which by construction iterates for as long as it is allowed, would be recorded as a failure rather than as a heuristic that used its budget; with only the declared budget, a generated loop without an exit condition would stall the campaign. Random number generators are seeded by the executor immediately before each call rather than by an instruction in the prompt, so that a randomised heuristic returns the same schedule every time it is run on the same instance.

A heuristic is scored on the whole training set. For each instance, each objective value is divided by a lower bound computed from that instance, so that the quantity compared is how many times the minimum attainable value rather than a raw time, and the normalised values are averaged across instances. Without this step the largest instances would dominate the mean. A heuristic that fails on some instances is penalised, not discarded. Each failed instance contributes a fixed penalty of twice its lower bound, in the same normalised units, a value of the same order as the population median and therefore deliberately mild, so that a heuristic succeeding on 11 of the 12 training instances and one succeeding on a single instance remain distinguishable by selection instead of both being eliminated at the first failure. Only individuals that fail on every instance are excluded, and even those are recorded, because the failure rate is a reported result.

3.4 Initial population, selection, crossover and mutation

The initial population is produced by prompting the model once per individual with the task specification, the objective description, the contract, a deliberately naive seed heuristic given for format only, and one technical direction drawn in round robin from a fixed list. Every prompt used in the study, together with the two lists of directions and the list of perturbations, is reproduced verbatim in Appendix A. The directions are of two kinds, and the distinction is carried through the rest of the study. *Constructive* directions describe ways of building a schedule in a single pass: orderings by processing time or remaining work, machine-selection criteria, bottleneck-first construction, gap insertion, one-operation lookahead, assignment before sequencing. *Improvement* directions describe ways of spending a time budget refining a schedule already built: tabu search on machine reassignments or on the critical path, simulated annealing, threshold accepting, iterated local search, GRASP, variable neighbourhood descent, late-acceptance hill climbing, beam search. These names state what the prompt asks for, not a verified property of the code that comes back: the framework checks feasibility and measures the objectives, and does not test whether a heuristic asked to use tabu search implements one faithfully. One in three initial individuals receives an improvement direction, alternated rather than sampled so that the proportion holds exactly, and every individual is tagged with its family. Offspring inherit the tag from a parent, so after the first generation it records lineage rather than a re-derived property of the code: a heuristic descended from a constructive one may well have acquired an improvement phase. The tag is nonetheless reported and used to separate the two lineages, because a constructive rule returns in milliseconds while an improvement heuristic spends its entire declared budget, and a runtime axis that mixes them freely becomes meaningless.

Selection follows NSGA-II without modification, with parents drawn by binary tournament on the resulting rank and crowding order and a $\mu + \lambda$ replacement. No component of this stage is generated or parameterised by the language model.

Each generation produces as many offspring as the population size; for each, mutation is chosen with probability r_m and crossover otherwise. In crossover the model receives two parent heuristics and the current reflection, and is prompted to generate a heuristic that takes what works from each, with explicit permission to drop any component it judges weak. In mutation it receives one parent and one structural perturbation drawn from a fixed list: replacing the operation-ordering rule or the machine-selection criterion, adding or removing a tie-break or a lookahead, changing the construction order, adding a left-shift repair pass, introducing randomised restarts, simplifying by removing the most elaborate component, shifting the balance towards one objective, or adding, removing or altering an improvement phase. The prompt asks for that change and no other: it forbids rewriting the heuristic from scratch and forbids making several changes at once. It also asks for the change to be made even when its effect is uncertain, which takes from the model the job of judging whether a variant is an improvement. That judgement belongs to the measurement: a variant that turns out worse is evaluated, ranked and discarded by selection like any other, so nothing is lost by trying it. This departs deliberately from the more common formulation, in which the mutation prompt asks for a version “better than” the current one: such a

prompt makes mutation a second exploitation operator acting in the same direction as selection, which removes the exploratory role mutation is there to play.

3.5 Reflection

After each generation, the model is prompted to review the solutions produced by the population and describe, in natural language, the patterns or strategies that appear to be effective. This description is then provided as additional context to the crossover operator in the next generation [6]. We use two choices to keep this process simple and consistent with the information already available from the optimisation procedure. First, the model is shown the non-dominated solutions together with the remaining population, rather than an additional clustering of the solutions. The non-dominated partition is already computed by the algorithm and has a direct interpretation in terms of Pareto dominance, whereas introducing a clustering step would require choosing an additional number of groups without a clear basis. Second, reflection is performed in two stages: the model first examines the current population and then combines this observation with the accumulated memory from previous generations. This allows information learned in earlier generations to be retained rather than replaced at each iteration.

The reflection component can be disabled and would naturally provide an ablation setting. However, this experiment was not conducted because of the token budget available for the study. We therefore do not attribute any of the reported performance to the reflection mechanism itself.

3.6 Sandbox, failure taxonomy and the output portfolio

The sandbox is the boundary between what the model said and what the benchmark accepts as a solution, and it produces the failure taxonomy reported in Section 5.9. Rather than classifying every failure as either infeasible or a code error, it distinguishes seven categories at execution time (`syntax`, `no_function`, `import`, `timeout`, `runtime`, `bad_output` and `infeasible`), and two more before execution, when the response cannot be turned into code at all: `no_code_block` and `truncated`. Two of these carry most of the information. `bad_output` and `infeasible` separate a model that cannot express a schedule in the required form from one that did not understand the problem. And `truncated` is kept apart deliberately, because it is caused by a token limit set by the experiment rather than by any deficiency of the model. The time limit is enforced by instrumenting the generated code itself, which does not interrupt a computation that passes through no loop, no comprehension and no call; this is unlikely in a scheduling heuristic, but the guarantee is not absolute.

A heuristic is deterministic once seeded: run on an instance, it returns one schedule, and that schedule evaluates to a single pair of objective values, that is, to a single point in objective space. The exact solvers return several points per instance instead, because the weighted protocol asks them for one solution per weight, and the memetic algorithm returns the non-dominated front of its final population. Comparing a point against a front and concluding that the point lacks diversity is a statement about the protocol, not about the method. The unit of comparison is therefore a *portfolio*:

k heuristics are run on the instance, the k resulting points are pooled, and the non-dominated subset is retained, at k times the cost of one heuristic. The members are selected using the rank-and-crowding order the algorithm already applies, looking only at the training instances, and are deduplicated by source code, since crossover and mutation occasionally return a parent unchanged.

The choice of a portfolio rather than a single heuristic is also forced by the selection criterion. If a single heuristic is extracted instead, environmental selection returns an extreme of the front, and since one of the two objectives is attained trivially and exactly by a construction rule, that extreme is the degenerate one: a heuristic that minimises total workload while ignoring the makespan entirely. Section 5.6 reports the measurement. A single point is not a valid summary of a multi-objective portfolio.

4 Experimental Setup

4.1 Objective selection

Six candidate second objectives were measured on the 15 Brandimarte instances. For each instance a constructive insertion rule was swept over a grid of 21 scalarisation weights crossed with five priority orders, which yields 105 schedules spread along the trade-off. On that set each candidate is summarised by its mean Spearman correlation ρ with the makespan, the mean number of non-dominated points it yields per instance, and the number of instances whose front collapses to two points or fewer, which are counted as degenerate. Table 1 shows that total workload is the only candidate that conflicts with the makespan consistently without degenerating on any instance. In contrast, maximum machine load, despite being widely used as a second objective in the FJSSP literature, produces two or fewer non-dominated solutions on 13 of 15 instances. This behaviour is expected because the load of any individual machine is bounded by the makespan, making the two objectives strongly related. We therefore select total workload as the second objective, while maximum machine load is still computed and reported as a third metric to facilitate comparison with previous studies.

Table 1: Candidate second objectives.

Candidate objective	ρ with $c_{\max}$	ND points	Degenerate	Note
Total workload W	-0.640	7.27	0/15	**selected**
Maximum machine load $L_{\max}$	$+0.878$	1.67	13/15	
Total job separation	$+0.009$	3.60	2/15	trivial bound
Total flow time	$+0.212$	3.13	2/15	used in Sec. 5.8
Load range	$+0.836$	2.47	8/15	non-linear
Load variance	$+0.820$	2.47	7/15	non-linear

4.2 Reference bounds and levels of guarantee

Every gap reported below is a distance from a reference value, one per objective and per instance. For the makespan, published best-known values exist on the Brandimarte

instances. For total workload they do not, because the objective is not reported on them. Using a published value for one objective and a value computed here for the other would make the two gaps incomparable. Both are therefore measured against an explicit three-level scheme, and the level used is reported per instance: *proven optimal* where an exact solver closed the instance, which holds for 11 of the 15 on the makespan; the *closed-form bound* $W^\star = \sum_{i \in O} \min_{m \in R_i} p_{m,i}$ for total workload, which is exact and always attainable because assigning every operation to its fastest machine is never infeasible; and *best known* for the remaining four instances, Mk06, Mk10, Mk13 and Mk15. However, since $W^\star$ is always attainable by greedily assigning operations to the fastest machine, the relevant trade-off is the additional workload required to reduce makespan. Every method compared on this objective pair attains $W^\star$ exactly on all 15 instances, with the single exception of the memetic algorithm, which misses it by less than half a percentage point on average, so the workload gap is reported but separates no two methods. The makespan is therefore the discriminating objective, and the gap of a method on an instance is the relative deviation from the makespan reference of the best schedule among those the method returns on it. Unless another objective is named, "gap" means that quantity throughout the paper.

4.3 Evaluation protocol

Individual solutions are evaluated using the gap, while solution sets are compared using hypervolume (HV) and inverted generational distance (IGD) against the non-dominated union of all evaluated solutions. As this reference set is not an optimal front, the number of non-dominated solutions is also reported. Both the reference set and the reference point used for the hypervolume are computed once, over the pooled output of every method, and held fixed thereafter. This matters where a stochastic method is reported run by run: recomputing the reference point from the separated runs would move it on most instances, and with it the hypervolume of every method in the study, including the deterministic ones.

Methods are not required to return the same number of solutions, and no attempt is made to equalise that number. What is equalised is the computational budget: every method is reported with the wall-clock time it consumed, and the number of solutions it returns is published next to its gap, so that the asymmetry is visible rather than hidden. Truncating a richer set to the size of a poorer one would require choosing which solutions to discard, a decision with no neutral answer. Cardinality does not drive the results in any case: the panel of dispatching rules returns 25 schedules per instance, more than the 10 of a generated portfolio, and is among the weakest methods on both objective pairs.

The protocol has three phases: the single-objective optima $c^\star_{\max}$ and $W^\star$, the latter given by the closed-form bound and so requiring no solver call; the two lexicographic extremes, minimising each objective subject to the other being at its optimum; and a weighted sweep over $w \in \{0.25, 0.5, 0.75\}$ of $w \cdot c_{\max}/c^\star_{\max} + (1-w) \cdot W/W^\star$, normalised by the optima so that w means the same thing on every instance. The second phase is also what resolves ties: a solver asked only to minimise the makespan returns one of the possibly many schedules that attain $c^\star_{\max}$, and the workload of that one can be arbitrarily bad unless it is pinned down in turn. Each solver therefore contributes

at most five points per instance, under a limit of 600 seconds per resolution applied equally to both. A weighted sum reaches only the points on the convex hull of the front, so a Pareto-optimal schedule lying in a non-convex region is unreachable at every weight. The protocol does not work around this, and the limitation applies equally to both solvers, which is what the comparison between them requires. The MILP model and CP-SAT, the constraint-programming solver of OR-Tools [21], inherit the same implementation of this protocol, so they solve the same phases with the same weights in the same order, and any difference between them is attributable to solver technology. Differences across methods are tested with a Friedman test over instances followed by pairwise Wilcoxon signed-rank tests with Holm correction.

4.4 Instances

The 15 Brandimarte instances [53], Mk01–Mk15, are used as the evaluation set, and are never exposed to the generated heuristics during training. Heuristics are instead generated using 12 instances from the Fattahi [16], Kacem [54], and Behnke [55] families.

Table 2 summarises the main characteristics of the benchmark families, where n is the number of instances in each family and flexibility the mean number of eligible machines per operation. The workload range measures the ratio between the total workload obtained by assigning each operation to its slowest and fastest eligible machine. A value of 1.00 indicates that machine assignment has no effect on workload, making the second objective uninformative, and the last column counts the instances of each family that are rigid in that sense.

Table 2: Benchmark families.

Family	n	Operations	Flexibility	Workload range	Rigid
brandimarte	15	55–284	1.4–4.1	1.73	0/15
barnes	21	100–225	1.1–1.3	**1.00**	**21/21**
dauzere	18	196–387	1.1–5.0	**1.02**	**9/18**
behnke	60	50–500	6.1–22.2	2.25	0/60
fattahi	20	4–48	1.5–2.7	1.32	0/20
kacem	4	12–56	5.0–10.0	7.79	0/4

Barnes [56] and Dauzère-Pérès [57] are excluded from training because their workload ranges of 1.00 and 1.02 provide almost no variation in the second objective. Including them would therefore provide little opportunity for the generated heuristics to learn machine-assignment trade-offs. The selected training families instead cover a broader range of problem sizes and flexibility than Brandimarte, supporting generalisation to unseen instances. In size the Brandimarte instances fall inside that range; in flexibility, the least flexible of them fall below it, so on those the generated heuristics are applied outside the conditions they were written for.

4.5 Baseline methods

Classical dispatching rules

The handcrafted baseline is a constructive insertion rule, deliberately stronger than the random-order greedy rule commonly used in this role. Each operation is placed in the earliest *feasible gap* on a candidate machine rather than appended to the end of its queue, so idle windows are reused; and each candidate insertion is scored by its effect on *both* objectives, normalised by their reference bounds, with w swept over $\{0, 0.25, 0.5, 0.75, 1\}$ to produce five points across the trade-off space. The rule therefore shares the normalisation scheme of the weighted phase of the exact protocol. The same machinery is run under five classical priority orders (most work remaining, shortest and longest processing time, job order and random order), so that the comparison isolates the priority rule and not the implementation; pooling all five across all five weights gives a panel of 25 points per instance. Each order is additionally run with random restarts until it has consumed the same wall-clock time as the generated portfolio, which is the equal-time control of Section 5.5.

Constraint programming

Each operation carries one optional interval variable per eligible machine, all sharing the operation's start and end variables, so that selecting a machine also fixes the duration; an `ExactlyOne` constraint over the presence literals selects one, and precedence within a job is a linear inequality between the end of an operation and the start of its successor. A single `NoOverlap` per machine replaces the $O(n^2)$ big-M disjunctions of the MILP with one global constraint. Because CP-SAT admits only integer-linear objectives, the normalised weights of the protocol are rescaled to integers by a common positive factor, which leaves the argmin unchanged. The model is solved with OR-Tools CP-SAT [21] under the protocol above.

Mixed-integer linear programming

The same protocol is solved with Gurobi, using the disjunctive formulation of [13]. Manne's original model [15] takes the machine of each operation as given and decides only the order on each machine, with one binary variable per pair of operations sharing a machine, linearised by a big-M constant. The flexible extension adds the routing decision, one binary variable per eligible operation–machine pair, and makes each disjunction conditional on both operations of the pair being assigned to the same machine. Its role is not to compete: it cross-validates the CP formulation, since two independently written models should agree on the value of an instance both can solve, and it documents the scalability wall of a Manne-type disjunctive formulation. It is run on Mk01–Mk05 only, with its computational performance and solution coverage reported in Section 5.2.

Multi-objective memetic algorithm

The memetic baseline is an NSGA-II with variable neighbourhood descent applied to part of the offspring, in the tradition of hybrid population-and-local-search methods for the FJSSP [26]. A chromosome is a precedence-compatible operation order together

with a machine assignment, and the decoder places each operation in the earliest feasible gap of its machine, so every chromosome decodes to a feasible schedule without repair. The descent cycles three neighbourhoods by first improvement, returning to the cheapest after every accepted move: reassigning an operation to another eligible machine, swapping two positions of the operation order, and reinserting one position elsewhere in it. The local search is intrinsically scalar, so each invocation draws a random weight and improves the correspondingly weighted sum of normalised objectives, which extends the front rather than concentrating it. It shares the selection and ranking implementation with the generation loop. Being an anytime method, it is not described by a single stopping point, so it is run at budgets of 1, 10 and 60 seconds per instance, which span the runtimes of every generated portfolio, with 10 independent seeds at each budget, and five in the transfer experiment. Each budget is reported as a row of its own, and a row describes a single run: the front it reports is the front one execution returns, and the indicators and gaps of the ten seeds are averaged, with their standard deviation given so that the run-to-run variation of a stochastic baseline is visible. The pooled front of the ten runs is reported as well, under its own label and charged its own cost of ten times the budget, since a multi-start of ten executions is a different and more expensive method rather than the same one. The contrasts of Section 5 use the single-run rows, which are the ones comparable with a portfolio run once.

The generated heuristics

The method under evaluation is the portfolio of $k = 10$ heuristics defined in Section 3.6, whose members are fixed by the rank-and-crowding order of the generation loop on the training instances alone, so that no information from the evaluation set enters the choice. The value of k counts heuristics, not solutions: the ten schedules a portfolio returns on an instance collapse to between two and eight non-dominated points, 5.5 and 4.9 on average for the two mixed portfolios. Ten members already return fewer distinct non-dominated points than members, so the portfolio is limited by redundancy among its members rather than by their number, and enlarging it would raise its cost in proportion to k without addressing that redundancy; k must still exceed one, since a single extracted heuristic is a degenerate extreme of the front. Portfolios drawn only from the constructive family and portfolios drawn from the whole population (*mixed*) are reported in separate rows throughout.

4.6 Language models and computational environment

Two models were compared as generators: `claude-sonnet-5 (anthropic)` and `gpt-5.1-codex (openai)`, each pinned to an exact version. Both were driven by a single client, through the OpenAI-compatible endpoint of OpenRouter. Any difference in behaviour is therefore attributable to the model rather than to two SDKs counting tokens and reporting errors differently. Sampling temperature was 1.0 and the output limit 16,000 tokens. The reasoning setting of each model was fixed at the cheapest level that still produced valid code, determined by prior measurement on identical prompts; the settings therefore differ between models (`effort: low` and `effort: minimal`), which is stated because presenting them as identical would be inaccurate,

and Section 5.9 reports how far those preliminary measurements underestimated the realised cost.

Each campaign used a population of 60 individuals with a mutation rate of 0.3, a declared budget of 2 seconds per heuristic evaluation, a hard sandbox limit of 30 seconds, $k = 10$ and a fixed random seed. Under the one-in-three alternation a population of 60 splits into 40 constructive and 20 improvement individuals, so every one of the 24 technical directions is instantiated at least once and each constructive direction three or four times; 60 individuals over 10 generations was in turn the largest campaign that fitted the per-model spending cap at the cost per call measured beforehand. The two campaigns were configured identically in every respect except that cap, which was \$8 for `claude` and \$5 for `openai`; each ran until it exhausted its own cap, the first completing eight generations and the second two. Neither campaign length was therefore a design choice. Neither the population size nor the mutation rate was tuned, and no search over either was performed, which is stated explicitly because the limited contribution of the evolutionary loop is one of the findings of this study.

All experiments ran on a standard personal computing environment (4-core (8-thread) Intel Core i7-1185G7 at 3.00 GHz, 16 GB RAM, no dedicated GPU, Python 3.12), so the reported runtimes reflect conditions available to a practitioner. Both exact solvers were configured with eight search threads, fully subscribing the machine's logical cores, whereas the dispatching rules and the generated heuristics run single-threaded. Runtimes for the generated heuristics were obtained in a separate pass, re-executing the stored heuristics with no model calls involved; the exact solvers and the campaigns were never run concurrently, which was verified after the fact by reconstructing each execution interval and checking for overlap.

5 Results and Discussion

5.1 Evaluation scope and reference values

The evaluation covers 44 method configurations across the 15 Brandimarte instances, yielding 660 possible method–instance pairs. Of these, 650 were executed; the 10 missing runs correspond to MILP configurations that were not launched. MILP was run on the first five instances of the set only, Mk01–Mk05. It proved optimality on none of them, and CP-SAT dominated its solutions on all five. The remaining instances are substantially larger, so MILP would have consumed an order of magnitude more time on them for the same outcome. The executed runs produced 9,383 individual solutions, with no execution errors. CP-SAT proved optimality for 11 of the 15 instances, requiring 3.74 hours of total computation time, with a median solve time of 86.1 seconds per instance.

For instances Mk06, Mk10, Mk13, and Mk15, CP-SAT did not prove optimality within the available time budget. For these four instances the reference is the best solution found within the budget rather than a proven optimum, and its guarantee level is reported per instance; CP-SAT necessarily has a gap of zero on them, since the reference is its own solution. CP-SAT is therefore never used as the control method, and is excluded from the families in which it would be compared against the reference it defines. All tests are run over the 15 instances: since the reference enters every

method's gap identically, using a best known value rather than a proven optimum rescales the whole family without affecting the comparisons between methods.

5.2 Benchmark on the Brandimarte instances

The minimum gap in the evaluation is zero, achieved only by CP-SAT. No generated heuristic or metaheuristic outperforms the exact solver on any instance. Therefore, the comparison focuses on computational cost and modelling effort rather than absolute solution quality.

The MILP reached the CP-SAT makespan on Mk01–Mk03 without proving optimality, while on Mk04 and Mk05 it reached gaps of 11.7% and 4.1%, respectively. The mean gap was 3.15%, with a median runtime of 3,452 seconds per instance, that is, the total across the six solver calls the protocol makes (the makespan optimum, the two lexicographic extremes and the three weighted resolutions), each subject to the same 600 seconds limit. Thus, the MILP results provide agreement with CP-SAT on the obtained objective values for the first three instances, but not independent optimality certificates.

Among the six methods that enter the test, a Friedman test over the makespan gap rejects equality decisively ($\chi^2 = 45.85$, $p = 9.7 \times 10^{-9}$), and Table 3 reports the ordering, with the gaps as defined in Section 4.2 and the times per instance.

Table 3: Main comparison on the Brandimarte instances.

Method	Mean rank	Mean gap	Median gap	Median time	Note
CP-SAT	—	0.00%	0.00%	86.1 s	11/15 proven optimal
MILP (Gurobi)	—	3.15%	0.00%	3,452 s	5/15, none proven
Generated, mix (openai)	1.87	6.52%	5.00%	11.5 s	2 generations
Generated, mix (claude)	1.90	6.96%	4.43%	20.1 s	8 generations
Memetic, best of 10 runs	—	8.11%	6.30%	600.5 s	pooled front, 10×60 s
Memetic, one run, 60 s	3.27	10.21%	6.46%	60.0 s	$\sigma = 1.40$ over 10 runs
Generated, constr (openai)	3.80	10.26%	9.12%	**2.1** s	2 generations
Memetic, one run, 10 s	—	12.78%	6.81%	10.0 s	$\sigma = 1.83$ over 10 runs
Generated, constr (claude)	4.97	15.58%	12.05%	6.2 s	8 generations
Insertion rule	5.20	18.15%	18.41%	0.014 s	handcrafted baseline
Memetic, one run, 1 s	—	31.44%	21.44%	1.0 s	$\sigma = 2.81$ over 10 runs

A Holm-corrected post-hoc with the insertion rule as control finds no significant difference between the two mixed portfolios ($p = 0.75$), while the two mixed portfolios, the constructive portfolio of `openai` and the memetic algorithm all improve on the insertion rule significantly (Holm ≤ 0.003). One comparison fails: the constructive portfolio of `claude` is not significantly better than the insertion rule ($p = 0.52$), which is what the ranking already suggests, since the two occupy adjacent positions at the foot of the table (4.97 against 5.20).

Against the memetic algorithm run once at 60 seconds, both mixed portfolios win on more instances than they lose, 11 to 3 for `claude` and 9 to 4 for `openai`, at raw $p = 0.008$ and $p = 0.039$. Neither difference survives Holm correction over the 15 pairs of this family (0.053 and 0.196), so it is not established; what the measurements

support is that a portfolio run once matches a memetic algorithm run once, at a third of its wall-clock time.

The statement this supports is worth giving in full: a portfolio of heuristics written by a language model matches a hand-implemented multi-objective memetic algorithm, at three to five times less computation per instance and without anyone having written the memetic algorithm. At the budget closest to the portfolios, 10 seconds, the memetic algorithm reaches a mean gap of 12.78% against their 6.52% and 6.96%; that row is reported for context and enters no contrast, because the family was declared in advance on the strongest single-run budget of the baseline.

5.3 Quality-runtime frontier

Figure 2 plots mean gap against median runtime on a logarithmic time axis, with the shape of each marker encoding the method family. Six methods are non-dominated, and two features of that frontier carry the argument. At essentially the same wall-clock budget of 2 seconds, the generated constructive portfolio sits almost four percentage points below the best a classical rule achieves with 1,000 random restarts (10.26% against 13.95%). And the `openai` mixed portfolio dominates the `claude` mixed portfolio, the 20-second restarts and the memetic algorithm at both of its reported settings: it is faster and better than a single 60-second run, and faster and better than the pooled front of ten such runs, which costs 600 seconds.

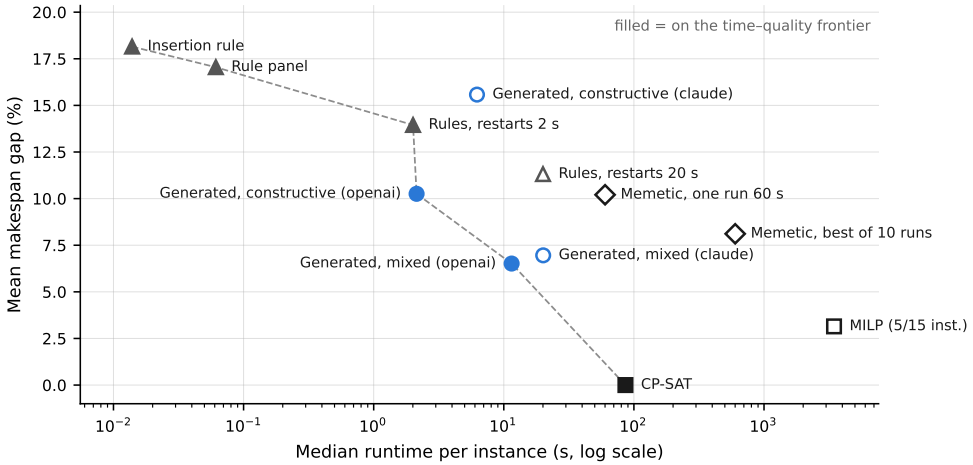


Fig. 2: Mean makespan gap against median runtime per instance, logarithmic time axis.

The constructive portfolio provides the clearest practical result. It improves the insertion rule by eight percentage points while remaining substantially faster than CP-SAT: about 40 times faster at the median and more than 1,300 times faster in the worst case.

Runtime should, however, be interpreted differently across methods. Improvement heuristics are given a fixed 2-second budget and use the available time on every instance, so a portfolio of k such heuristics has a runtime close to k times this budget. For example, the 20.1 seconds reported for the `claude` mixed portfolio is mainly determined by this design and should not be interpreted as the actual time required to generate its schedules.

The constructive portfolio behaves differently. All but one of its members terminate in less than 25 ms, while one member accounts for most of the total runtime (2.007 seconds out of 2.087 seconds). This member is labelled as constructive because it inherited the label from its parent during crossover, although the resulting heuristic also contains an improvement phase. Since the portfolio runtime is dominated by this member, its reported value is still close to the actual computation performed and is therefore suitable for comparisons based on computational speed.

5.4 Pareto-set quality

Reporting only the makespan gap would hide a real deficiency, because the indicators of Table 4 rank the methods in a different order. The LLM portfolios reach a smaller gap than the memetic algorithm (7.0% and 6.5% against 10.2%), yet the memetic algorithm approximates the whole front far better: its IGD is 0.0054 against 0.012 and 0.0223, and it returns 16.3 points per instance against 5.5 and 4.9. A gap on one objective rewards the single best schedule found and says nothing about how much of the front the method covers.

Table 4: Set-quality indicators, averaged over the 15 instances.

Method	HV	IGD	ND points
CP-SAT	0.2004	0.0175	3.27
Generated, mixed (claude)	0.1987	0.0120	5.47
Memetic, best of 10 runs	0.1982	0.0031	18.93
Generated, mixed (openai)	0.1968	0.0223	4.93
Memetic, one run, 60 s	0.1965	0.0054	16.34
Rules, restarts at 20 s	0.1958	0.0229	3.60

For hypervolume, the `claude` mixed portfolio and the memetic algorithm are close, with mean values of 0.1987 and 0.1965. The overall Friedman test across the six methods is significant ($\chi^2 = 64.60$, $p = 1.4 \times 10^{-12}$), and the portfolio wins the pairwise comparison on 11 of the 15 instances at a raw $p = 0.030$, which does not survive Holm correction ($p = 0.091$): the two are not separated.

The difference becomes clearer with IGD. The memetic algorithm obtains significantly lower IGD than all other methods (Holm-adjusted $p = 0.0009$ in all five comparisons). This difference is related to the number of solutions covering the Pareto front: the memetic algorithm produces an average of 16.3 non-dominated solutions per instance, compared with 5.5 and 4.9 for the two generated portfolios, and on Mk11 it

averages 34.5. Thus, the generated heuristics achieve similar hypervolume by finding comparable extreme solutions, but provide much less coverage of the intermediate regions of the Pareto front, as shown in Figure 3. The panels draw one run of the memetic algorithm, the seed with the median number of points, because pooling its ten runs would draw a front that no single execution returns; its run-to-run variation is reported in Table 3.

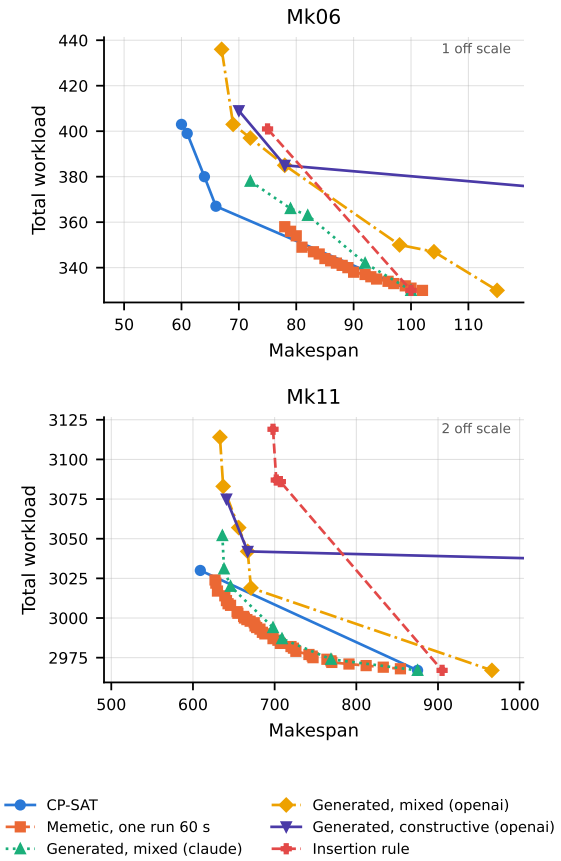


Fig. 3: Pareto fronts on Mk06 and Mk11.

This limitation is structural, not a matter of tuning. The members of a portfolio are chosen for their spread in the space of average fitness over the training instances, which says nothing about the points they produce on any single evaluation instance; once selected they run independently, each returning one or two points, and many of those points coincide.

5.5 Equal wall-clock time experiments against classical rules

The generated heuristics are also compared with classical dispatching rules at three levels of increasing computational effort.

At the first level, the rules are evaluated individually. Most work remaining gives the smallest mean gap, 18.15%, followed by random order (22.36%), longest processing time (28.67%), shortest processing time (31.55%), and job order (83.10%). As a consistency check, the most-work-remaining configuration produces exactly the same makespan as the insertion baseline on all 15 instances, confirming that both use the same implementation with different ordering rules.

At the second level, the five rules are combined across five priority weights, giving 25 solutions per instance. This portfolio requires only 0.06 seconds and achieves a mean gap of 17.05%, only slightly better than the best individual rule because most work remaining dominates the other rules on most instances. The constructive portfolio of `openai` performs better than this panel, with $p = 0.0012$ (12–3–0); the `claude` one does not (8–7–0, $p = 0.68$).

Finally, the comparison is repeated under an equivalent wall-clock budget. The classical rule is restarted with different random orders until it consumes approximately the same time as the generated portfolio, corresponding to roughly 1,000 constructions within 2 seconds and 10,000 within 20 seconds. The results are reported in Table 5, where W–L–T counts wins, losses and ties over the 15 instances. Holm correction is applied within the eight comparisons against the control method, the 20-second restarts, which are the entries marked ^{c}; the remaining comparisons do not involve the control, so for them Holm is applied over all 36 pairwise tests of this family, which is the more conservative of the two schemes.

Table 5: Generated heuristics and the memetic algorithm against classical dispatching rules.

Method	Against	Mean gap	Rules	W–L–T	p	Holm
Mixed portfolio (claude), 20.1 s	restarts at 20 s	6.96%	11.31%	11–2–2	0.007	**0.036**^{c}
Mixed portfolio (openai), 11.5 s	restarts at 20 s	6.52%	11.31%	11–3–1	0.016	0.063^c
Memetic, one run, 60 s	restarts at 20 s	10.21%	11.31%	10–4–1	0.132	0.328^c
Constructive portfolio (openai), 2.1 s	restarts at 20 s	10.26%	11.31%	7–5–3	0.388	0.388^c
Constructive portfolio (claude), 6.2 s	restarts at 20 s	15.58%	11.31%	4–10–1	0.109	0.328^c
Constructive portfolio (openai), 2.1 s	restarts at 2 s	10.26%	13.95%	10–2–3	0.034	0.444
Both mixed portfolios	rule panel	6.52–6.96%	17.05%	15–0–0	< 0.001	**0.002**
Both mixed portfolios	insertion rule (MWR)	6.52–6.96%	18.15%	15–0–0	< 0.001	**0.002**

A Friedman test over the nine comparable methods rejects equality decisively ($\chi^2 = 70.18$, $p = 4.5 \times 10^{-12}$), with mean ranks placing the two mixed portfolios first (2.17 and 2.20), then the memetic algorithm (3.77), then the 20-second restarts (4.47) and the `openai` constructive portfolio (4.67), then the 2-second restarts (6.17), the `claude` constructive portfolio (6.77), the rule panel (7.27) and most work remaining (7.53).

Against the hardest rule baseline, 10,000 random constructions per instance, only the `claude` mixed portfolio is significant after correction (Holm = 0.036). The `openai` mixed portfolio also wins on 11 of the 15 instances, at a raw $p = 0.016$, but ends at

Holm = 0.063; and the memetic algorithm, at three times the budget of that baseline, wins 10 of 15 at $p = 0.13$. The constructive portfolio does not reach significance either: its edge over the 2-second restarts does not survive Holm correction ($p = 0.44$), and against the 20-second restarts the difference is negligible (10.26% against 11.31%, $p = 0.39$). It therefore matches a rule given 10 times its budget.

Two results do not follow the overall pattern. On Mk01 and Mk14 the restart baseline outperforms the generated portfolio, reaching the proven optimum on Mk01 and coming within 0.58% of it on Mk14, against gaps of 2.50% and 2.74%. Mk01 is the smallest instance in the benchmark, with 55 operations, and Mk14 one of the least flexible, with 1.56 eligible machines per operation. In both cases the machine-assignment search space is small enough for brute-force randomisation to cover it.

The `claude` constructive portfolio also performs below the 2-second restart baseline on 9 of the 15 instances, with mean gaps of 15.58% and 13.95%, respectively ($p = 0.64$). Thus, the observed advantage of the constructive portfolio over the restart baseline is driven by the `openai` portfolio. Since only one run was performed for each model, these results should be interpreted as observations for the two evaluated portfolios rather than as general differences between the models.

5.6 Evolutionary loop contribution

Most of the gain appears in the first two or three generations (Table 6). The values are makespans divided by the lower bound of their own instance and averaged over the training set, so 1.6563 means 66% above that bound. Generation zero is the initial population, produced by prompting alone before any crossover or mutation has taken place, and every later generation adds one round of variation followed by environmental selection; the last generation listed for each campaign is the last it ran before reaching its spending cap. The short campaign gained in two generations what the long one gained in eight (2.5% against 2.4% in best makespan): it is not a truncated long run, it captured the whole effect. Beyond the third generation the best makespan still falls, but at less than half the earlier rate, and the gain no longer reaches the held-out instances: the portfolios extracted at generations two and eight of the same campaign give 6.94% and 6.96%. That comparison also controls for the different lengths of the two campaigns: within one model, cutting at generation two or at generation eight gives the same result on the held-out instances, so the comparison between the two models is not confounded by one of them having run four times longer. What the remaining five generations buy is coverage, improving the IGD from 0.0271 to 0.0120. What the loop raises is the floor, not the ceiling: the population median gains 7.7% and 10.3% while the best gains around 2.5%.

One extreme of the front is free: in both campaigns the best total workload equals the closed-form bound from generation zero, because one of the 12 constructive directions assigns each operation to its fastest machine, which attains that objective exactly. That extreme is therefore not an achievement of the method. Since one of the two objectives is already at its optimum from generation zero, the only room left for improvement is on the makespan, which is part of why the hypervolume barely moves. It is also why a single heuristic cannot stand in for the portfolio: asked for one

Table 6: Normalised makespan over the training instances, by generation.

Campaign	Gen	Best	Median
`claude-sonnet-5`	0	1.6563	1.9614
	3	1.6286	—
	8	1.6161	1.8109
`gpt-5.1-codex`	0	1.6803	2.1661
	2	1.6379	1.9433

individual, environmental selection returns the heuristic that sits at that free extreme, with a mean makespan gap of 521% against a workload gap of 0%.

Finally, the two models converge to nearly the same place, 1.6161 and 1.6379, a difference of 1.3%. Two frontier models from different laboratories, prompted identically, arrive at equivalent heuristics. The practical consequence of this subsection is direct: for this problem two or three generations suffice, and the 60-heuristic initial population guided by 24 technical directions does the greater part of the work.

5.7 Modelling effort

The second part of the comparison concerns modelling effort, which is measured through two indicators: problem-specific expert decisions and code lines. An expert decision is a modelling choice that requires knowledge of the problem, such as defining a chromosome, neighbourhood, or big-M constant. The memetic algorithm requires nine such decisions, MILP eight, CP-SAT and the dispatching rules five each, and the generated heuristics three.

Problem-specific code follows a similar pattern: 250, 221, 202, and 146 lines for the memetic algorithm, MILP, CP-SAT, and dispatching rules, respectively, compared with 54 lines for the generated heuristics. Code length is only a proxy for modelling effort, but all methods were implemented by the same researchers using common interfaces. The remaining 570 lines of shared problem infrastructure and 986 lines of generation infrastructure are excluded from this comparison.

Thus, the result is not simply less code overall, but substantially less problem-specific code and fewer expert modelling decisions.

5.8 Extending to different objectives

The portfolios generated for the makespan and total-workload pair were reused, at zero cost in model calls, and evaluated on total flow time, selected by the procedure of Table 1. Table 7 reports the result. Neither exact solver was run on this objective, so the gap is measured against a lower bound rather than a proven optimum, and "Solutions" counts the schedules each method produces per instance, from which the best total flow time is taken.

A Friedman test over the eight comparable methods rejects equality decisively ($\chi^2 = 97.19$, $p = 4.1 \times 10^{-18}$). The generated portfolios beat the rules restarted at

Table 7: Transfer to total flow time.

Method	Mean gap	Time	Solutions
Memetic, best of 5 runs	135.6%	300.2 s	20.0
Memetic, one run, 60 s	140.8%	60.0 s	4.0
Generated portfolio (openai)	144.6%	16.3 s	10.0
Generated portfolio (claude)	147.8%	18.4 s	10.0
Rules, restarts at 20 s	180.1%	20.0 s	5.0
Rule panel	203.4%	0.07 s	25.0
Random order	212.1%	0.01 s	5.0
Most work remaining	245.6%	0.02 s	5.0

equal wall-clock on all 15 instances ($p < 0.001$, Holm $= 0.0004$), beat the instantaneous rule panel on all 15, and beat the strongest single rule on all 15. The difference between the two models is again not significant ($p = 0.42$), indicating that the method does not depend on the provider.

The most informative row is the last. Most work remaining, the rule that performed best on the original objective pair, is beaten by random order on all 15 instances (245.6% against 212.1%, $p < 0.001$). The cause is structural. Total flow time changes only when a job's last operation is placed; everywhere else the incremental effect is zero and identical across machines, so the choice falls to an arbitrary tie-break. The rule does not optimise, it guesses. Repairing it would mean redesigning it around an estimate of future completions, precisely the modelling effort the previous section measures. The memetic algorithm run once is still ahead of both portfolios on 10 of the 15 instances, but neither difference is significant ($p = 0.08$ and $p = 0.39$); it too re-points for free, because it receives the objectives as a parameter. The asymmetry this experiment establishes is therefore not that generated heuristics beat the memetic algorithm, but that metaheuristics and generated heuristics re-point at no cost while dispatching rules and exact formulations do not.

5.9 Generation cost and reliability

The two generation campaigns made 540 and 180 model calls at costs of at least \$8.84 and \$5.45, respectively, and both stopped at their spending caps rather than because of errors. These counts cover the initialisation, crossover and mutation calls, and exclude the two reflection calls issued per generation. The realised cost per call was 1.9 and 6.9 times higher than estimated from an initial probe, mainly because most campaign calls were crossover prompts containing two parent solutions and substantially more reasoning tokens. The spending cap was therefore an important part of the experimental design.

Generation reliability was high: 98.7% and 95.6% of generated individuals were valid in the two campaigns. Truncation was the most common single failure category for `openai` (2.2%), against 1.1% for infeasible schedules and 1.1% for runtime errors. Since truncation resulted from the experiment's output limit, it is reported separately from model failures. Among genuine failures, `infeasible` and `runtime` were the main categories, while `bad_output` was negligible. Five of the nine failure categories

were never observed, including syntax errors, missing functions, unavailable imports, malformed responses, and time-limit violations. Overall, the failures were therefore mainly related to the feasibility or execution of the generated schedules rather than to violations of the required output format.

6 Conclusions

This paper set out to measure how far automatically generated heuristics close the gap between exact solvers and dispatching rules for the flexible job shop under two conflicting objectives, under a protocol in which every method is evaluated identically, the handcrafted baseline is deliberately strong, and the objectives were verified to conflict before being used.

No generated heuristic, and no metaheuristic, improves on the exact solver on any instance. The contribution of automated heuristic design in this setting is therefore not solution quality but what is obtained per unit of computation and of modelling effort, and on both axes the result is favourable: a portfolio of generated heuristics matches a hand-implemented memetic algorithm at three to five times less computation, improves on classical rules given 10,000 random constructions per instance, and required 54 lines of natural-language problem description against 221 lines of disjunctive formulation. That improvement over the rules holds on 11 of the 15 instances for both models, and is significant after correction for one of them. Three further results stand independently of that comparison and deserve to be carried forward.

First, the evolutionary loop contributes far less than is usually assumed. It improves the best heuristic by around 2.5% in total, most of it in the first two or three generations, and raises the median of the population considerably more than its best member. What does the work is the initial population of 60 heuristics guided by 24 distinct technical directions. The cheap experiment that would establish whether this holds elsewhere is simply to report generation zero.

Second, the method does not depend on the model. Two frontier models from different laboratories, prompted identically, converged to heuristics differing by 1.3%, and showed no significant difference when their portfolios were transferred to a new objective. That matters for the durability of results in a field whose instruments are replaced every few months. Each model was run once, however, so where the two portfolios do differ, as the constructive ones do, the difference cannot be separated from run-to-run variation: what this supports is the agreement between the models, not a ranking of them.

Third, the advantage of a generated heuristic is clearest exactly where a handcrafted rule is blindest. Total flow time changes only when a job's last operation is placed, so an incremental dispatching rule has no signal to act on for most decisions; the directed rule loses to its own randomised version on all 15 instances. Generated heuristics are not constrained to an incremental decision rule and do not inherit that blindness.

The portfolio's one clear weakness is front coverage: it reaches extreme points equivalent to those of the memetic algorithm but covers the front some three times less densely, because a set of independently generated heuristics carries no diversity

pressure among its members. Three directions follow. The first is to make the portfolio produce its points jointly instead of independently: selecting the pooled points for spread, or giving each member a region of the front to cover, puts the diversity pressure where it is now missing. Since the gain is concentrated early, the budget it consumes is better spent on a larger and more diverse initial population, or on several seeds, than on further generations, which would also supply the error bars this study could not afford. And one capacity that would distinguish the approach most sharply from an exact formulation remains untested: absorbing a sequence-dependent extension of the problem through textual respecification alone, which was not attempted here and is therefore not claimed.

Appendix A Prompts

The prompts are reproduced verbatim. Text in braces marks a placeholder substituted at run time; everything else is fixed and identical for both models. Four blocks (the system message, the problem statement, the objective description and the input–output contract) are shared by every call that asks for code, so that the model always receives the same definition of the problem and the prompts differ only in the task.

Table A1: Composition of each call.

Call	Fixed blocks	Substituted at run time
Initialisation	A.1 + seed	one technical direction
Crossover	A.1	two parents, consolidated reflection
Mutation	A.1	one parent, one perturbation
Reflection, observe	—	group summary, code samples
Reflection, consolidate	—	previous memory, current observation

A.1 Fixed blocks

System message

> ```
> You are an expert in the design of constructive heuristics for production
> scheduling. You write Python code and nothing else. Return exactly one code
> block, formatted as ‘‘‘python ... ‘‘‘, containing a single self-contained
> function. Do not explain the code outside the block.
> ```

Problem statement

> ```
> The Flexible Job Shop Scheduling Problem (FJSSP).
>
> A set of jobs must be processed. Each job is an ordered sequence of operations
> that must run one after another: an operation cannot start before the previous
> operation of the same job has finished. Each operation can be processed by any
> machine from its own set of eligible machines, and the processing time depends
> ```

```
on which machine is chosen. A machine processes at most one operation at a time
and cannot be interrupted.

A solution assigns every operation to one eligible machine and fixes its start
time, so that job precedences hold and no two operations overlap on the same
machine.
```

Objectives

```
Two objectives are minimised, and they conflict:

  1. MAKESPAN - the time at which the last operation finishes.
  2. TOTAL WORKLOAD - the sum of the processing times actually used, across all
     machines.

Putting every operation on its fastest machine minimises total workload but
creates contention on those machines and inflates the makespan. Spreading work
onto slower machines relieves the contention and shortens the makespan, at the
cost of consuming more machine time.

You do not need to balance them in any particular way: a heuristic that leans
towards either objective is useful, as long as it is deliberate about it.
```

Input–output contract

```
The function must be named 'heuristic' and take one argument.

INPUT - a dictionary with:
  - n_jobs     : number of jobs.
  - n_machines : number of machines.
  - time_budget : seconds the function is allowed to spend, as a float. A
                  heuristic that only constructs a schedule can ignore it. A
                  heuristic that iterates MUST respect it: read the clock with
                  time.perf_counter() at the start, check it inside the loop,
                  and return the best schedule found so far once the budget is
                  spent. A function that overruns it is discarded, however good
                  its schedule was going to be.
  - jobs       : dictionary. Each key is a job number (jobs start at 1).
                  Each value is the list of that job's operations, in the order
                  they must be executed. Each operation is a tuple of two lists:
                    * the eligible machine numbers (machines start at 0),
                    * the processing time on each of those machines, in the
                      same order.

OUTPUT - a dictionary with:
  - one key per job number, using the same numbering as the input,
  - and as value the list of that job's operations, each one a dictionary with:
      'Operation'      : position of the operation within the job, from 1,
      'Assigned Machine' : the chosen machine number,
      'Start Time'     : integer start time,
      'End Time'       : integer end time,
      'Processing Time' : the processing time of the chosen machine.

The schedule is rejected unless every operation is present, every machine is
eligible, every duration matches the input, job precedences hold, and no two
operations overlap on a machine.
```

```
These libraries are available and need no import: math, random, itertools,
functools, operator, statistics, collections, heapq, copy, time, numpy (as np).
No other library can be imported.

The random generators are seeded before the function is called, so a
randomised heuristic gives the same result every time it runs on the same
instance. Do not reseed them.
```

Seed heuristic

Supplied only in the initialisation call, and only to fix the format. It is deliberately naive: it schedules job by job and places each operation on the machine that can start it earliest.

```
def heuristic(input_data):
    schedule = {}
    machine_free = {m: 0 for m in range(input_data['n_machines'])}
    for job, operations in input_data['jobs'].items():
        schedule[job] = []
        job_ready = 0
        for position, (machines, times) in enumerate(operations):
            options = sorted((max(machine_free[m], job_ready), m, times[k])
                             for k, m in enumerate(machines))
            start, machine, duration = options[0]
            schedule[job].append({
                'Operation': position + 1,
                'Assigned Machine': machine,
                'Start Time': start,
                'End Time': start + duration,
                'Processing Time': duration,
            })
            machine_free[machine] = start + duration
            job_ready = start + duration
    return schedule
```

A.2 Generation and variation

Initialisation

```
{system}

{problem}

{objectives}

{io_contract}

{seed}

Write a different and better heuristic than the reference above, respecting the
input and output contract exactly.

Take this direction: {approach}

Include a docstring of at most 40 words stating the key idea. Output only the
code block.
```

```
Keep the whole function under 60 lines. A heuristic that needs more than that
is doing too much: the schedule is built in one pass over the operations and
the decision rules are a handful of comparisons. An improvement heuristic may
use its full budget in iterations, but its code stays this short too.
```

Crossover

```
{system}

{problem}

{objectives}

{io_contract}

### Heuristic A
```python
{parent_a}
```

Heuristic B
```python
{parent_b}
```

What has been learned so far
{reflection}

Write a new heuristic that takes what works from A and from B. You are not
required to keep any particular part of either: if a component of one is
clearly weak, drop it.

Include a docstring of at most 40 words stating what you took from each and
why. Output only the code block.
Keep the whole function under 60 lines. A heuristic that needs more than
that is doing too much: the schedule is built in one pass over the
operations, and the decision rules are a handful of comparisons.
```

Mutation

The reflection is deliberately absent here; see Section 3.4.

```
{system}

{problem}

{objectives}

{io_contract}

### Current heuristic
```python
{parent}
```

Write a variant of this heuristic with the following change:

    {perturbation}
```

```
Make the change deliberately, even if you are not sure it will help: the
population is evaluated afterwards and a worse variant is informative too. Do
not rewrite the heuristic from scratch and do not make several changes at once.

Include a docstring of at most 40 words stating what you changed. Output only
the code block.

Keep the whole function under 60 lines. A heuristic that needs more than that
is doing too much: the schedule is built in one pass over the operations and
the decision rules are a handful of comparisons. An improvement heuristic may
use its full budget in iterations, but its code stays this short too.
```

A.3 Reflection

Observe

The two groups are the non-dominated front and the remainder of the population. {`performance`} carries the size, mean and best value of each group on both objectives, normalised by the instance lower bound, together with its mean failure rate; {`heuristics`} carries the source code of the three best-scoring members of each group.

```
You are analysing heuristics for the Flexible Job Shop Scheduling Problem.

Below is a group of heuristics with similar performance on the two objectives,
makespan and total workload. Their average position is:

{performance}

Scores are standardised: positive is above the population average, negative is
below.

### Heuristics
{heuristics}

In at most 50 words, state what these heuristics have in common structurally
and how that relates to where they sit on the two objectives. No preamble, no
restating the question.
```

Consolidate

{`previous`} is the reflection carried over from the previous generation and {`observations`} the output of the call above. The result becomes the {`reflection`} of the next generation's crossover.

```
You are advising on the design of heuristics for the Flexible Job Shop
Scheduling Problem, where makespan and total workload are minimised together.

### What was concluded in previous generations
{previous}

### What the current population shows
{observations}

In at most 150 words, give design advice for the next generation. Identify
```

```
which structural choices are paying off on each objective and which are not.
Do not refer to groups, clusters or specific heuristics by name; write advice
that stands on its own.
```

A.4 Technical directions

One direction is drawn in round robin per initial individual, substituted as {approach}. Each family has its own counter, so both lists are traversed completely.

Constructive (12)

1. Order the operations by shortest processing time first.
2. Order the operations by longest processing time first.
3. Give priority to the jobs with the most remaining work.
4. Give priority to the jobs with the least remaining work.
5. Assign each operation to the machine that finishes it earliest.
6. Assign each operation to its fastest machine whenever contention allows it.
7. Assign each operation to the least loaded machine among the eligible ones.
8. Build the schedule machine by machine rather than job by job.
9. Insert each operation into the earliest idle gap that can hold it, not only at the end of a machine's queue.
10. Identify the bottleneck machines first and schedule around them.
11. Use a one-operation lookahead: consider the effect on the next operation of the same job before choosing.
12. Decide all machine assignments first, then sequence the operations on each machine.

Improvement (12)

13. Build one schedule with a greedy rule, then improve it with a tabu search on machine reassignments, keeping a short tabu list of recently moved operations and stopping when the budget runs out.
14. Build one schedule with a greedy rule, then improve it with a tabu search that only moves operations lying on the critical path of the current makespan.
15. Build one schedule with a greedy rule, then improve it with simulated annealing: accept a worse neighbour with probability exp(-delta/T), cool T geometrically, and return the best schedule ever seen.
16. Build one schedule with a greedy rule, then improve it with threshold accepting: accept any neighbour that is not worse than the current one by more than a threshold, and lower the threshold as the budget is spent.
17. Use iterated local search: descend to a local optimum, perturb it by rescheduling a few random operations, descend again, and keep the perturbation only if it paid off.
18. Use GRASP: repeatedly build a schedule with a randomised greedy rule that picks from a restricted candidate list, apply a local descent to each, and keep the best.
19. Use variable neighbourhood descent: cycle through two or three different neighbourhoods and only move to the next one when the current one is exhausted.
20. Use random-restart hill climbing: descend from several different randomised starting schedules and return the best local optimum found.
21. Use late-acceptance hill climbing: compare each candidate against the solution from a fixed number of iterations ago instead of the current one.
22. Use a beam search over the construction: keep several partial schedules alive at each step and expand only the most promising ones.

23. Improve an initial schedule with a local search whose move is to swap two consecutive operations on the same machine, spending the whole budget on it.
24. Improve an initial schedule with a local search that alternates between reassigning one operation to another eligible machine and reordering the queue of the machine it left.

A.5 Structural perturbations

One perturbation is drawn in round robin per mutation, substituted as {`perturbation`}.

1. Replace the rule that decides the order of the operations with a different one.
2. Replace the criterion used to choose the machine for an operation.
3. Add a tie-breaking rule where the current one chooses arbitrarily.
4. Remove a tie-breaking rule and see whether the heuristic still holds up.
5. Change the construction order: job by job, operation by operation, or machine by machine.
6. Add a lookahead step that considers the consequence of a choice before committing to it.
7. Remove the lookahead, if there is one, and make the heuristic purely greedy.
8. Add a repair pass that runs after construction and left-shifts operations into earlier gaps.
9. Introduce randomisation with several restarts, keeping the best schedule found.
10. Simplify: remove the most elaborate component and check what is actually carrying the performance.
11. Shift the balance deliberately towards makespan, at the cost of total workload.
12. Shift the balance deliberately towards total workload, at the cost of makespan.
13. Add a local search that spends the remaining time budget improving the schedule the heuristic just built.
14. Remove the iterative improvement phase, if there is one, and keep only the construction.
15. Change the acceptance rule of the improvement phase: strictly improving, or tolerant of small worsenings.
16. Change the neighbourhood of the improvement phase: move one operation to another machine, or swap two operations on the same machine.
17. Spend the time budget on several randomised restarts instead of one long descent, or the other way round.

References

[1] Dauzère-Pérès, S., Ding, J., Shen, L., Tamssaouet, K.: The flexible job shop scheduling problem: A review. European Journal of Operational Research **314**(2), 409–432 (2024)

[2] Tang, J., Liu, G., Pan, Q.: A review on representative swarm intelligence algorithms for solving optimization problems: Applications and trends. IEEE/CAA Journal of Automatica Sinica **8**(10), 1627–1643 (2021)

[3] Martí, R., Sevaux, M., Sörensen, K.: Fifty years of metaheuristics. European Journal of Operational Research **321**(2), 345–362 (2025)

[4] Zhong, R., Hussien, A.G., Yu, J., Munetomo, M.: Llmoa: A novel large language model assisted hyper-heuristic optimization algorithm. Advanced Engineering Informatics **64**, 103042 (2025)

[5] Liu, F., Tong, X., Yuan, M., Lin, X., Luo, F., Wang, Z., Lu, Z., Zhang, Q.: Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model (2024)

[6] Ye, H., Wang, J., Cao, Z., Berto, F., Hua, C., Kim, H., Park, J., Song, G.: ReEvo: Large Language Models as Hyper-Heuristics with Reflective Evolution (2024)

[7] Wu, X., Wu, S.-h., Wu, J., Feng, L., Tan, K.C.: Evolutionary Computation in the Era of Large Language Model: Survey and Roadmap (2024)

[8] Huang, S., Yang, K., Qi, S., Wang, R.: When large language model meets optimization. Swarm and Evolutionary Computation **90**, 101663 (2024)

[9] Ros, F.D., Soprano, M., Gaspero, L.D., Roitero, K.: Large language models for combinatorial optimization: A systematic review (2025)

[10] Ghanbarzadeh, R., Mirjalili, S.: A structured review of large language models in metaheuristic optimisation. Decision Analytics Journal **15**, 100587 (2025)

[11] Çetinkaya, İ.O., Büyüktahtakın, İ.E., Shojaee, P., Reddy, C.K.: Discovering heuristics with large language models (LLMs) for mixed-integer programs: Single-machine scheduling. Computers & Operations Research **186**, 107325 (2026)

[12] Yao, S., Liu, F., Lin, X., Lu, Z., Wang, Z., Zhang, Q.: Multi-objective Evolution of Heuristic Using Large Language Model (2025)

[13] Özgüven, C., Özbakir, L., Yavuz, Y.: Mathematical models for job-shop scheduling problems with routing and process plan flexibility. Applied Mathematical Modelling **34**(6), 1539–1548 (2010). Cited by: 276

[14] Roshanaei, V., Azab, A., Elmaraghy, H.: Mathematical modelling and a meta-heuristic for flexible job shop scheduling. International Journal of Production Research **51**(20), 6247–6274 (2013). Cited by: 86

[15] Manne, A.S.: On the job-shop scheduling problem. Operations Research **8**(2), 219–223 (1960)

[16] Fattahi, P., Saidi-Mehrabad, M., Jolai, F.: Mathematical modeling and heuristic approaches to flexible job shop scheduling problems. Journal of Intelligent Manufacturing **18**(3), 331–342 (2007)

[17] Demir, Y., İşleyen, S.K.: Evaluation of mathematical models for flexible job-shop scheduling problems. Applied Mathematical Modelling **37**(3), 977–988 (2013)

[18] Meng, L., Zhang, C., Ren, Y., Zhang, B., Lv, C.: Mixed-integer linear programming and constraint programming formulations for solving distributed flexible job shop scheduling problem. Computers and Industrial Engineering **142**

(2020). Cited by: 241

[19] Ham, A.M., Cakici, E.: Flexible job shop scheduling problem with parallel batch processing machines: Mip and cp approaches. Computers and Industrial Engineering **102**, 160–165 (2016). Cited by: 71

[20] Laborie, P.: Ibm ilog cp optimizer for detailed scheduling illustrated on three problems. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 6th International Conference, CPAIOR 2009 Pittsburgh, PA, USA, May 27-31, 2009 Proceedings 6, pp. 148–162 (2009). Springer

[21] Perron, L., Didier, F.: CP-SAT (2024)

[22] Da Col, G., Teppan, E.C.: Industrial size job shop scheduling tackled by present day cp solvers. In: Principles and Practice of Constraint Programming: 25th International Conference, CP 2019, Stamford, CT, USA, September 30–October 4, 2019, Proceedings 25, pp. 144–160 (2019). Springer

[23] Yuraszeck, F., Montero, E., Canut-De-Bon, D., Cuneo, N., Rojel, M.: A constraint programming formulation of the multi-mode resource-constrained project scheduling problem for the flexible job shop scheduling problem. IEEE Access **11**, 144928–144938 (2023)

[24] Jeong, K.-C., Kim, Y.-D.: A real-time scheduling mechanism for a flexible manufacturing system: Using simulation and dispatching rules. International Journal of Production Research **36**(9), 2609–2626 (1998). Cited by: 117

[25] Alvarez-Valdes, R., Fuertes, A., Tamarit, J.M., Giménez, G., Ramos, R.: A heuristic to schedule flexible job-shop in a glass factory. European Journal of Operational Research **165**(2), 525–534 (2005). Cited by: 71

[26] Gao, J., Sun, L., Gen, M.: A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. Computers and Operations Research **35**(9), 2892–2907 (2008). Cited by: 491

[27] Ding, H., Gu, X.: Improved particle swarm optimization algorithm based novel encoding and decoding schemes for flexible job shop scheduling problem. Computers and Operations Research **121** (2020). Cited by: 111

[28] Ho, K.-H., Cheng, J.-Y., Wu, J.-H., Chiang, F., Chen, Y.-C., Wu, Y.-Y., Wu, I.-C.: Residual scheduling: A new reinforcement learning approach to solving job shop scheduling problem. IEEE Access **12**, 14703–14718 (2024)

[29] Echeverria, I., Murua, M., Santana, R.: Leveraging constraint programming in a deep learning approach for dynamically solving the flexible job-shop scheduling problem. Expert Systems with Applications **265**, 125895 (2025)

[30] Song, W., Chen, X., Li, Q., Cao, Z.: Flexible job-shop scheduling via graph neural network and deep reinforcement learning. IEEE Transactions on Industrial Informatics **19**(2), 1600–1610 (2022)

[31] Lei, K., Guo, P., Zhao, W., Wang, Y., Qian, L., Meng, X., Tang, L.: A multi-action deep reinforcement learning framework for flexible job-shop scheduling problem. Expert Systems with Applications **205**, 117796 (2022)

[32] Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Qu, R.: Hyper-heuristics: a survey of the state of the art. Journal of the Operational Research Society **64**(12), 1695–1724 (2013) https://doi.org/10.1057/jors.2013.71

[33] Nguyen, S., Mei, Y., Zhang, M.: Genetic programming for production scheduling: a survey with a unified framework. Complex & Intelligent Systems **3**(1) (2017) https://doi.org/10.1007/s40747-017-0036-x

[34] Zhang, F., Mei, Y., Nguyen, S., Zhang, M.: Evolving scheduling heuristics via genetic programming with feature selection in dynamic flexible job-shop scheduling. IEEE Transactions on Cybernetics **51**(4), 1797–1811 (2021) https://doi.org/10.1109/TCYB.2020.3024849

[35] Pallagani, V., Muppasani, B.C., Roy, K., Fabiano, F., Loreggia, A., Murugesan, K., Srivastava, B., Rossi, F., Horesh, L., Sheth, A.: On the prospects of incorporating large language models (LLMs) in automated planning and scheduling (APS). Proceedings of the International Conference on Automated Planning and Scheduling **34**, 432–444 (2024)

[36] Sartori, C.C., Blum, C., Bistaffa, F., Rodríguez Corominas, G.: Metaheuristics and large language models join forces: Toward an integrated optimization approach. IEEE Access **13**, 2058–2079 (2025)

[37] Zhang, S., Gong, C., Wu, L., Liu, X., Zhou, M.: AutoML-GPT: Automatic Machine Learning with GPT (2023)

[38] Jiang, X., Wu, Y., Li, M., Cao, Z., Zhang, Y.: Large Language Models as End-to-end Combinatorial Optimization Solvers (2025)

[39] Liu, S., Chen, C., Qu, X., Tang, K., Ong, Y.-S.: Large language models as evolutionary optimizers. In: 2024 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2024)

[40] Liu, F., Tong, X., Yuan, M., Zhang, Q.: Algorithm Evolution Using Large Language Model (2023)

[41] Stein, N., Bäck, T.: Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. IEEE Transactions on Evolutionary Computation **PP**, 1–1 (2024)

[42] Hemberg, E., Moskal, S., O'Reilly, U.-M.: Evolving Code with A Large Language Model (2024)

[43] Brahmachary, S., Joshi, S.M., Panda, A., Koneripalli, K., Sagotra, A.K., Patel, H., Sharma, A., Jagtap, A.D., Kalyanaraman, K.: Large Language Model-Based Evolutionary Optimizer: Reasoning with elitism (2024)

[44] Liu, S., Chen, C., Qu, X., Tang, K., Ong, Y.-S.: Large Language Models as Evolutionary Optimizers (2024)

[45] Meyerson, E., Nelson, M.J., Bradley, H., Gaier, A., Moradi, A., Hoover, A.K., Lehman, J.: Language Model Crossover: Variation through Few-Shot Prompting (2024)

[46] Bradley, H., Fan, H., Galanos, T., Zhou, R., Scott, D., Lehman, J.: The openelm library: Leveraging progress in language models for novel evolutionary algorithms, 177–201 (2024)

[47] Dat, P.V.T., Doan, L., Binh, H.T.T.: HSEvo: Elevating Automatic Heuristic Design with Diversity-Driven Harmony Search and Genetic Algorithm Using LLMs (2024)

[48] Huang, J., Li, X., Gao, L., Liu, Q., Teng, Y.: Automatic programming via large language models with population self-evolution for dynamic job shop scheduling problem (2024)

[49] Yatong, W., Yuchen, P., Yuqi, Z.: TS-EoH: An Edge Server Task Scheduling Algorithm Based on Evolution of Heuristic (2024)

[50] Liu, F., Lin, X., Wang, Z., Yao, S., Tong, X., Yuan, M., Zhang, Q.: Large Language Model for Multi-objective Evolutionary Optimization (2024)

[51] Huang, Y., Wu, S., Zhang, W., Wu, J., Feng, L., Tan, K.C.: Autonomous Multi-Objective Optimization Using Large Language Model (2024)

[52] Xiong, H., Shi, S., Ren, D., Hu, J.: A survey of job shop scheduling problem: The types and models. Computers & Operations Research **142**, 105731 (2022)

[53] Brandimarte, P.: Routing and scheduling in a flexible job shop by tabu search. Annals of Operations Research **41**(3), 157–183 (1993)

[54] Kacem, I., Hammadi, S., Borne, P.: Pareto-optimality approach for flexible job-shop scheduling problems: hybridization of evolutionary algorithms and fuzzy logic. Mathematics and Computers in Simulation **60**(3–5), 245–276 (2002)

[55] Behnke, D., Geiger, M.J.: Test instances for the flexible job shop scheduling problem with work centers. Research Paper RR-12-01-01, Helmut-Schmidt-Universität, Universität der Bundeswehr Hamburg, Hamburg,

Germany (2012)

[56] Barnes, J.W., Chambers, J.B.: Flexible job shop scheduling by tabu search. Technical Report Series ORP96-09, Graduate Program in Operations Research and Industrial Engineering, The University of Texas at Austin, Austin, TX, USA (1996)

[57] Dauzère-Pérès, S., Paulli, J.: An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. Annals of Operations Research **70**, 281–306 (1997)