IJMIC Vol. 3, Issue 1, 46 - 89

Research Article

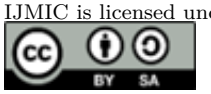
Quasi-Optimal Low-Complexity Algorithms for the Permutation Flowshop Problem with Weighted Tardiness Penalties

S. I. Josfrid Agbadogbe[1] and Christopher Thron[2*]

[1]Institute of Mathematics and Physical Sciences (IMSP), University of Abomey-Calavi, Benin
[2]Texas A&M University–Central Texas, Killeen, Texas, USA

*Corresponding author: Christopher Thron, thron@tamuct.edu

Received: [May 24, 2026]
Accepted: [July 29, 2026]
Published: [September 4, 2026]



Abstract — The permutation flowshop scheduling problem (PFSP) with N jobs and M machines is NP-hard for $M > 2$. This paper considers a generalized objective combining a fixed penalty for each tardy job with a penalty proportional to tardiness, under job-specific release times and due dates. Existing heuristics such as NEHedd address the conventional proportional-tardiness objective and do not account for this penalty structure. We propose three low-complexity heuristics: (i) a Pareto-filtered insertion heuristic extending the NEH construction, (ii) a windowed near-exhaustive selection heuristic, and (iii) a hybrid that retains the better solution from the insertion heuristic and NEHedd. The methods are evaluated on 5,100 randomly generated instances spanning three job-to-machine ratios and three traffic intensities. The insertion heuristic achieves 7.2% lower average penalty than selection while running 16 times faster, and its advantage increases from 0.6% at low traffic to 12.5% at high traffic. Insertion achieves lower average penalties than NEHedd on all tested instance sizes under low traffic, and on smaller instances under medium and high traffic. The hybrid achieves over 10% penalty reduction compared to NEHedd for smaller instances, while the relative reduction decreases with increasing configuration size. Runtime analysis indicates polynomial scaling for insertion, with $\mathcal{O}((NM)^b)$, $1.31 < b < 1.67$.

Keywords: scheduling, permutation flowshop, weighted tardiness, fixed late penalty, heuristics, NEHedd, hybrid algorithm, Pareto optimality, scalability

1. INTRODUCTION

In industrial contexts marked by increased competition, mastering deadlines and costs is a strategic challenge. Production scheduling plays an essential role, as it organizes the execution of a set of tasks on several machines in order to optimize overall system performance. This challenge is particularly present in manufacturing, logistics, and distribution, where poor scheduling leads to significant delays and additional costs.

Among the scheduling models studied, the permutation flowshop scheduling problem (PFSP) holds a central place. In this framework, all jobs must be executed in the same order on all machines, which faithfully reflects many industrial environments. The problem remains difficult to solve, especially when minimizing total weighted tardiness penalties while accounting for due dates, arrival times, and job priorities.

Despite advances in the literature, designing high-performing and scalable heuristics for large-scale instances remains a challenge. Existing methods, such as NEHedd, offer good performance, but their computational cost and adaptability to the multi-machine PFSP with a combined fixed-and-proportional weighted-tardiness penalty remain limited.

This paper addresses the following question: *How can simple heuristics, originally developed for the single-machine problem, be adapted to produce effective, fast, and scalable solutions to the multi-machine PFSP with weighted tardiness and fixed late penalties?*

To address this problem, we propose an adaptation of two single machine iterative heuristics (insertion and selection) to the multi-machine PFSP, integrating arrival times and due dates. We also present a hybrid algorithm combining the insertion heuristic and NEHedd.

Why a hybrid algorithm is needed. A natural objection is that, if one of the two heuristics were simply better overall, a hybrid combination would add computational cost for no real benefit. Our experiments (detailed in Sections 6.1–6.4) show that this is not the case: insertion and NEHedd are near-complementary rather than one dominating the other. Across all 5,100 test instances, insertion produces the better solution in 49.2% of cases and NEHedd in 50.8%; this balance shifts toward NEHedd as traffic intensifies (45.6% versus 54.4% under high traffic) and toward insertion under low traffic (52.3% versus 47.7%), and also shifts with the job-to-machine ratio and instance size (Sections 6.2–6.3). By running both heuristics and keeping the better result, the hybrid algorithm achieves the lowest penalty of all tested methods in every configuration and traffic level (Section 6.2), at the cost of the additional computation required to run both components (quantified in Section 6.5).

The heuristics are evaluated on a variety of synthetic instances, allowing comparison of performance in terms of penalty and computation time. An analysis of scalability and the trade-offs between solution quality, computational cost, and parameter tuning completes the study.

The remainder of the paper is organized as follows. Section 2 reviews related literature. Section 3 presents the mathematical formulation of the problem. Section 4 describes the adapted heuristics. Section 5 presents the experimental design. Section 6 presents and discusses the results. Section 7 summarizes contributions and directions for future work.

2. LITERATURE REVIEW

2.1 The Permutation Flowshop Problem

The permutation flowshop problem (PFSP) is a fundamental category of combinatorial optimization scheduling problems, widely studied in industrial production management [1, 2]. A set of n jobs $J = \{J_1, J_2, \ldots, J_n\}$ must be processed on m machines $M = \{M_1, M_2, \ldots, M_m\}$, each job following the same machine order from M_1 to M_m. The permutation hypothesis restricts the search space to $n!$ possible sequences.

Each job J_i is characterized by: a processing time $x_{i,j}$ on machine M_j; an arrival time a_i; a due date d_i; and a weight w_i reflecting priority. The PFSP has been studied under various optimization objectives, including minimizing makespan ($C_{\max}$), total tardiness, weighted tardiness [3], and multi-objective criteria [4].

Figure 1 illustrates a schedule for a small PFSP instance, showing how the same job order is respected on every machine while idle gaps appear whenever a machine must wait for its next job to clear the preceding stage. The instance shown is the five-job, three-machine example used again in Sections 4.1 and 4.2 to trace the heuristics step by step, so that the reader can refer back to a single concrete instance throughout the paper.

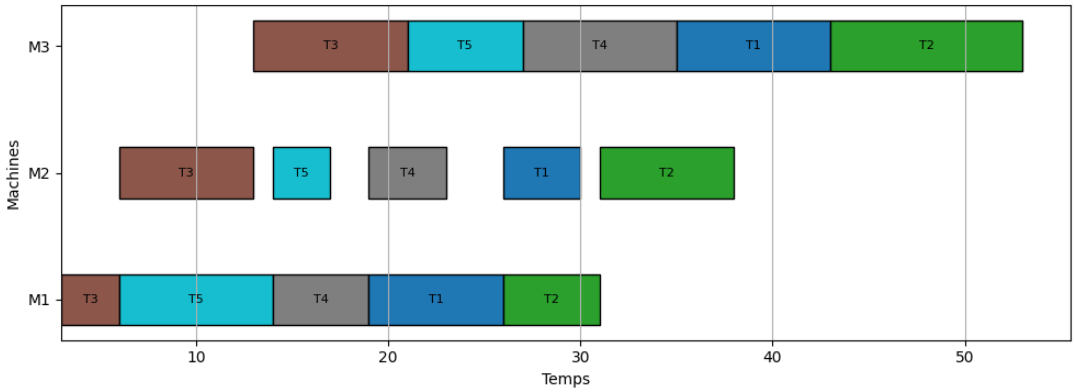


Figure 1: Gantt chart of the schedule $[1, 4, 3, 5, 2]$ for the five-job, three-machine instance of Section 4.1 (one colour per job, three machines M1–M3). Triangles on the time axis mark the arrival times a_n: no job starts on M1 before its own arrival (Constraint (8)), no job starts on a machine before completing the previous one (Constraint (6)), and the job order is identical on all three machines (Constraint (7)). The idle gaps visible on M2 and M3 are the cost of that permutation restriction.

The PFSP is NP-hard for $m \geq 3$ [5], making exact approaches impractical for large instances; exact methods such as the branch-and-bound scheme of Kim [6] remain confined to small instances, which motivates the use of efficient heuristics.

2.2 Single-Machine Heuristics

Priority rules (dispatching rules) for single-machine scheduling include: Earliest Due Date (EDD), which schedules jobs by increasing d_i and is effective for minimizing the number of late jobs; Shortest Processing Time (SPT), which minimizes average flow time; and Longest Processing Time (LPT). Comparative assessments of such dispatching rules in dynamic flow-shop settings are given by Oukil and El-Bouri [7]. These rules have $O(n \log n)$ complexity but do not account for multi-machine constraints, making them unsuitable for direct application to the PFSP. They also focus on single criteria and do not handle weighted penalties with arrival times.

2.3 NEHedd for the Multi-Machine PFSP

The NEH algorithm [1], originally designed to minimize makespan, has been adapted to due-date objectives. The NEHedd variant [2, 3] is widely recognized as a reference for minimizing total weighted tardiness. Its steps are:

1. **Initial ordering**: Sort jobs by ascending due date (EDD rule).
2. **Partial sequence construction**: Initialize with the first job. For each subsequent job k, insert it into all k possible positions of the current partial sequence.
3. **Evaluation**: For each insertion, compute $\sum_{i=1}^{k} \max(0, C_i - d_i)$, where C_i is the completion time on the last machine.
4. **Selection**: Retain the insertion minimizing total tardiness; repeat until all jobs are sequenced.

Evaluated naively, this construction requires $\mathcal{O}(N^2)$ partial-sequence evaluations at $\mathcal{O}(NM)$ each, giving $\mathcal{O}(N^3M)$ overall; for the makespan objective the accelerations of Taillard [8] reduce this to $\mathcal{O}(N^2M)$, but they do not transfer directly to due-date objectives, where each insertion changes the completion times of all downstream jobs and hence their tardiness.

NEHedd's main limitation is that it focuses exclusively on the partial sequence penalty, ignoring the completion time of partial solutions. A partial solution with low penalty but a late finish creates a cascading effect that delays subsequent jobs leading to suboptimal final solutions.

2.4 Recent Advances in Flowshop and Tardiness Scheduling

Beyond the classical NEH-family heuristics, the last five years have seen renewed interest in flowshop scheduling under due-date-related objectives. A recent systematic review by Prata et al. [9], covering 92 articles on PFSP variants with due-date-related objectives (total tardiness, earliness-tardiness, maximum lateness, number of tardy jobs), confirms that the great majority of contributions address a single continuous tardiness-type objective and rely on metaheuristic or iterated-greedy search rather than low-complexity constructive heuristics. Table 1 positions the most closely related recent contributions along the three dimensions that matter for the present work: the objective addressed, the solution approach, and whether computational complexity is characterized.

Metaheuristic approaches. Silva et al. [10] develop and compare four metaheuristics (iterated local search, iterated greedy, variable greedy, and a steady-state genetic algorithm) for a weighted quadratic tardiness PFSP objective, and Costa et al. [11] propose dispatching and improvement procedures for a related weighted squared-tardiness PFSP. Li et al. [12] propose a two-stage iterated greedy algorithm, itself built around a two-stage NEH-based construction heuristic, for a constrained PFSP with mixed hard and soft due dates and weighted tardiness. Feng et al. [13] extend tardiness-oriented PFSP heuristics, via a tabu-memory iterated greedy algorithm, to distributed heterogeneous flowshops, illustrating how the tardiness objective studied here is increasingly considered in more complex shop topologies. Fuchigami and de Athayde Prata [14] address a related multi-component due-date objective (a linear combination of earliness, tardiness, and due-date-assignment cost) using coronavirus-optimization metaheuristics together with a MILP formulation. These works consistently report strong solution quality; none, however, reports a complexity characterization of the search itself, and all require a number of iterations that grows with instance size.

Refinements of NEH-based construction. On the constructive side specifically, Li and Zhang [15] report a further practical improvement to the NEH construction – a 9.18% penalty reduction relative to standard NEH – when integrated with discrete-event simulation for satellite-industry flowshop scheduling, confirming that refining NEH-based construction remains an active and practically motivated line of research five years after the improvements of Sauvey and Sauer [16]. This is the line of work to which the present paper contributes most directly.

Learning-based schedulers. Separately, Bouška et al. [17] embed a deep neural network as a polynomial-time completion-time estimator inside a decomposition algorithm for single-machine total tardiness, achieving a 0.26% optimality gap on instances of up to 800 jobs – an accuracy that constructive heuristics do not approach, but at the price of an offline training phase tied to a particular cost structure and instance distribution.

Relative to this body of work, three gaps motivate the present study. First, most recent metaheuristic contributions – iterated greedy, genetic, and coronavirus-optimization variants among them – report strong solution quality but at a computational cost that scales unfavorably with instance size, since they typically require many iterations over full or near-full neighborhoods; by contrast, this paper targets heuristics whose complexity is characterized explicitly and shown empirically to remain low-degree polynomial in $N \cdot M$ (Section 6.6), which is a prerequisite for use on very large instances or in near-real-time settings. Second, the combined objective studied here – a fixed penalty per late job *and* a penalty proportional to tardiness, together with job-specific arrival times – is more general than the (weighted, squared, or quadratic) tardiness

Table 1: Positioning of closely related recent work. "Complexity" indicates whether the contribution reports a theoretical or empirical characterization of its running-time growth.

Reference	Objective	Approach	Shop	Complexity
Costa et al. [11]	Weighted squared tardiness	Dispatching + improvement	PFSP	No
Silva et al. [10]	Weighted quadratic tardiness	ILS, IG, VG, GA	PFSP	No
Bouška et al. [17]	Total tardiness	Deep learning + decomp.	Single mach.	Partial
Feng et al. [13]	Total tardiness	Tabu-memory IG	Distributed	No
Fuchigami and de Athayde Prata [14]	Earliness–tardiness + due date	Metaheuristic + MILP	PFSP	No
Li et al. [12]	Weighted tardiness (hard/soft)	Two-stage IG	PFSP	No
Li and Zhang [15]	Makespan (practical)	Improved NEH + DES	PFSP	No
Gradwohl et al. [4]	Tardiness + late jobs	Pareto constructive	Single mach.	Yes
This paper	Fixed + proportional tardiness	Pareto constructive, hybrid	PFSP	Yes

objectives considered in the works above, none of which combine a discrete per-job penalty with a continuous tardiness-rate penalty in the way formalized in Section 3.3. Third, where learning-based schedulers such as Bouška et al. [17]'s have been proposed for related objectives, they require substantial offline training data and retraining when the cost structure or instance distribution changes, whereas the heuristics proposed here require no training and adapt immediately to arbitrary penalty parameters (p_n, q_n) and arrival processes. This positions the insertion, selection, and hybrid heuristics developed below as a low-complexity, readily deployable alternative that complements, rather than replaces, the metaheuristic and learning-based literature when computation time is at a premium.

2.5 Gaps and Contribution

The analysis above reveals two main gaps. First, single-machine heuristics cannot be directly applied to multi-machine PFSP because they ignore machine interactions and arrival time constraints. Second, NEHedd's single-criterion greedy construction can miss good solutions because it neglects the makespan of partial solutions.

This issue parallels the single-machine problem studied by Gradwohl et al. [4], who propose Pareto-based algorithms balancing penalty and lateness. The present work extends this approach to the multi-machine setting, developing heuristics that jointly consider immediate penalty and long-term scheduling impact, while remaining low-complexity enough to scale to large instances (Section 6.6).

3. MATHEMATICAL MODEL

We schedule N jobs on M machines arranged in series, under the permutation constraint that every job follows the same processing order $M_1, M_2, \ldots, M_M$. Our objective generalizes the weighted tardiness criterion by combining a fixed penalty, charged once for every late job re-

gardless of how late it is, and a penalty proportional to the amount of tardiness.

3.1 Parameters

- N: number of jobs; M: number of machines.
- x_{nm} (time units): processing time of job n on machine m. These are job- and machine-specific and need not be equal across machines.
- a_n (time units): arrival time of job n, i.e. the earliest instant at which job n becomes available and can start processing on the first machine. Jobs are not all available at time zero; a job cannot begin machine 1 before its arrival time (this is enforced by Constraint (8)).
- d_n (time units): due date of job n, i.e. the instant by which job n is expected to complete all M machines without penalty.
- p_n (cost units, e.g. monetary units): a *fixed* penalty incurred exactly once if job n finishes after d_n, independently of the magnitude of the delay. This models fixed contractual or reputational costs that are triggered by any lateness, however small.
- q_n (cost units per time unit): the *tardiness rate* of job n – the additional cost incurred per unit of time that job n finishes after d_n. Unlike p_n, this term grows continuously with the duration of the delay.
- C_{big}: a sufficiently large constant (Big-M) used only to linearize the activation of the fixed penalty (Constraint (11)); it has no economic meaning in the model.

On the compatibility of p_n and q_n. Because p_n is expressed directly in cost units and q_n in cost units *per time unit*, the product $q_n \cdot T$ (a rate multiplied by a tardiness duration T, itself in time units) is expressed in the same cost units as p_n. This is what allows the two penalty terms to be added directly in the objective (Section 3.3) without any unit conversion, rescaling, or weighting coefficient: both $p_n v_n$ and $q_n T_n$ are, term by term, plain costs, so their sum is a well-defined total cost.

3.2 Decision Variables

The model uses *position-based* decision variables: rather than indexing schedule quantities by job identity, the model indexes them by the job's position $k = 1, \ldots, N$ in the sequence, with a separate assignment variable u_{nk} linking positions back to actual jobs. This formulation, standard for permutation scheduling MILPs, allows completion-time and precedence constraints to be written directly in terms of a single sequence position rather than requiring a full precedence matrix.

- $u_{nk} \in \{0, 1\}$: equals 1 if job n is placed at position k in the sequence, 0 otherwise.
- $S_{km} \geq 0$ (time units): start time of the job occupying position k on machine m.
- $C_{km} \geq 0$ (time units): completion time of the job occupying position k on machine m.
- $x_{km} \geq 0$ (time units): processing time of the job occupying position k on machine m – i.e., the processing time of whichever job n has $u_{nk} = 1$, on machine m.
- $T_n \geq 0$ (time units): tardiness of job n, i.e. how many time units late job n finishes relative to its own due date d_n.
- $v_n \in \{0, 1\}$: equals 1 if job n is late ($T_n > 0$), 0 otherwise. This indicator is what activates the fixed penalty p_n in the objective.

Note that the two penalty variables T_n and v_n are indexed by *job*, not by position, whereas the timing variables S_{km}, C_{km}, x_{km} are indexed by position. Keeping the penalty variables job-indexed is what allows the objective to be written with the penalty data p_n and q_n as fixed coefficients, and hence to remain linear (Section 3.3); Constraint (9) below is what links the job-indexed tardiness to the position-indexed completion times.

3.3 Objective Function

Minimize the total cost combining fixed and proportional tardiness penalties:

$$\min \sum_{n=1}^{N} \left(p_n \cdot v_n + q_n \cdot T_n \right). \tag{1}$$

Two properties of Equation (1) deserve comment, since they determine whether the two penalty components may legitimately be summed.

Unit compatibility. As noted in Section 3.1, both terms are expressed in a common cost unit: $p_n v_n$ is a cost (the fixed penalty p_n, switched on or off by the binary indicator v_n), and $q_n T_n$ is likewise a cost (a cost-per-time-unit rate q_n multiplied by a tardiness duration T_n in time units). Their sum is therefore a well-defined total cost, and no normalizing or weighting coefficient between the two components is required. The two terms are nevertheless not interchangeable in behaviour: $p_n v_n$ is a discontinuous step incurred at the instant a job becomes late, whereas $q_n T_n$ grows continuously thereafter. The relative magnitudes used in the experiments (p_n on the order of tens of cost units, q_n on the order of a few cost units per time unit; see Section 5.2) mean that the fixed component dominates for jobs that are only marginally late, while the proportional component dominates for jobs that are severely late – which is precisely the behaviour this objective is intended to capture.

Linearity. Because p_n and q_n are input data rather than decision variables, and because v_n and T_n are indexed by job, every term of Equation (1) is a constant multiplied by a single decision variable. The objective is therefore linear, and together with Constraints (2)–(11) below – all of which are linear – the formulation is a mixed-integer linear program that can be passed directly to a standard MILP solver. Had the penalty variables instead been indexed by position, the objective would have required products of the form $\left(\sum_n p_n u_{nk}\right) v_k$, i.e. products of two decision variables, and the model would have been bilinear rather than linear. Throughout the remainder of this paper, the value of this objective function for a given candidate sequence is referred to as its *penalty*; every "penalty" value reported in the experimental results of Section 6 is a direct evaluation of Equation (1), i.e. the fitness value of that candidate solution.

3.4 Constraints

$$\sum_{k=1}^{N} u_{nk} = 1, \qquad \forall n \tag{2}$$

$$\sum_{n=1}^{N} u_{nk} = 1, \qquad \forall k \tag{3}$$

$$x_{km} = \sum_{n=1}^{N} x_{nm} \cdot u_{nk}, \qquad \forall k, m \tag{4}$$

$$C_{km} = S_{km} + x_{km}, \qquad \forall k, m \tag{5}$$

$$S_{k,m+1} \geq C_{km}, \qquad \forall k,\ m < M \tag{6}$$

$$S_{k+1,m} \geq C_{km}, \qquad \forall k < N,\ \forall m \tag{7}$$

$$S_{k1} \geq \sum_{n=1}^{N} a_n \cdot u_{nk}, \qquad \forall k \tag{8}$$

$$T_n \geq C_{kM} - d_n - C_{\text{big}} \cdot (1 - u_{nk}), \qquad \forall n, k \tag{9}$$

$$T_n \geq 0, \qquad \forall n \tag{10}$$

$$T_n \leq C_{\text{big}} \cdot v_n, \qquad \forall n \tag{11}$$

Constraints (2)–(3) form a standard assignment structure ensuring that every job occupies exactly one position and every position holds exactly one job, together guaranteeing that u encodes a valid permutation of the N jobs. Constraint (4) carries the processing time of the job assigned to position k through to machine m. Constraint (5) is the basic completion-time identity (completion equals start plus processing). Constraint (6) enforces that the job at position k cannot start on machine $m+1$ before it has completed machine m (machine precedence). Constraint (7) enforces that, on any given machine, the job at position $k+1$ cannot start before the job at position k has completed on that same machine (this is what makes the schedule a *permutation* schedule: the same job order is respected on every machine). Constraint (8) prevents any job from starting on the first machine before it has physically arrived. Finally, Constraints (9)–(11) jointly define tardiness and switch on the fixed penalty. Constraint (9) links the job-indexed tardiness variable to the position-indexed completion times: when $u_{nk} = 1$ (job n occupies position k) the Big-M term vanishes and the constraint reduces to $T_n \geq C_{kM} - d_n$, whereas when $u_{nk} = 0$ the Big-M term makes the right-hand side sufficiently negative that the constraint is inactive. Combined with (10) and the fact that the objective is increasing in T_n, this drives T_n to exactly $\max(0,\ C_{kM} - d_n)$ at the single position k actually assigned to job n. Constraint (11) then forces $v_n = 1$ whenever $T_n > 0$; the reverse implication need not be imposed, since $p_n \geq 0$ and the objective is minimized, so no optimal solution sets $v_n = 1$ for a job that is on time. The constant C_{big} serves purely as a linearization device in (9) and (11); in practice any value at least equal to $\max_n d_n + \sum_n \sum_m x_{nm}$ is sufficient, and it should be chosen no larger than necessary, since unnecessarily large Big-M values weaken the linear-programming relaxation and slow solver convergence.

This formulation is NP-hard for $M \geq 3$ [5], motivating the heuristic approaches developed in the following section.

4. HEURISTIC ALGORITHMS

We adapt two iterative heuristics from the single-machine framework of Gradwohl et al. [4] to the multi-machine PFSP. Both algorithms iteratively build partial job sequences, accounting for partial penalty and partial makespan, while controlling complexity via parameters (S, P, W, K).

4.1 Iterative Multi-Machine Insertion Heuristic

The insertion heuristic adapts the classic NEH construction [1] to the multi-machine PFSP, using Pareto optimality to maintain a set of robust candidate solutions rather than a single sequence. Parameters S (maximum insertion positions considered) and P (maximum Pareto-optimal sequences retained) control complexity.

1. **Initialization**: Sort jobs by ascending due date d_n. Initialize with a single sequence containing the first job.
2. **Progressive insertion**: At step t ($t = 1, \ldots, N$), take the t-th job from the sorted list and:
   (a) Insert it into the last S positions of each currently retained sequence (positions near the end are preferred since jobs with later due dates are most penalized when placed early).
   (b) For each resulting sequence, compute completion times via Equations (5)–(8).
   (c) Compute each job's tardiness $T_n = \max(0, C_{kM} - d_n)$ for the position k it occupies, and the total penalty $\sum_n (p_n v_n + q_n T_n)$.
   (d) Apply Pareto filtering on (penalty, makespan): a sequence dominates another if it is strictly better on at least one criterion and at least equal on the other.
   (e) From Pareto-optimal sequences, retain at most P by evenly spaced selection after sorting by penalty then makespan.
3. **Output**: Select the retained sequence with minimum total penalty.

Restricting insertions to the last S positions reduces complexity compared to NEHedd, which evaluates all positions. Pareto filtering improves robustness by preserving solution diversity: rather than discarding all but a single best partial sequence at each step (as NEHedd does), the insertion heuristic keeps a controlled set of non-dominated alternatives, so that a partial sequence with a slightly higher penalty but a shorter partial makespan – which may lead to a better final solution once later jobs are inserted – is not prematurely discarded.

Illustrative example. Consider $M = 3$ machines and $N = 5$ jobs with:

$$X = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 2 & 3 \\ 1 & 5 & 2 \\ 3 & 1 & 4 \\ 2 & 4 & 3 \end{bmatrix}, \quad a = [0, 1, 0, 2, 1], \quad d = [8, 7, 10, 6, 9],$$

$$p = [52, 47, 44, 61, 41], \quad q = [9, 1, 4, 6, 5].$$

With parameters $S = 3$, $P = 4$, the preliminary list sorted by ascending d_n is $[4, 2, 1, 5, 3]$, so the construction starts from the single sequence $[4]$ and inserts jobs $2, 1, 5, 3$ in that order. Table 2 traces every candidate generated and its evaluation.

Two features of this trace are worth highlighting. First, Pareto filtering is not merely decorative: at step 2 the sequence $[4, 2]$ dominates $[2, 4]$ (lower penalty, equal makespan) and the latter is discarded, whereas at step 5 the sequence $[1, 4, 5, 2, 3]$ survives filtering *despite* a higher penalty than $[1, 4, 3, 5, 2]$, because its shorter makespan makes it non-dominated – exactly the diversity-preserving behaviour that NEHedd's single-sequence construction cannot reproduce. Second, the heuristic returns $[1, 4, 3, 5, 2]$ with a total penalty of 288, and exhaustive enumeration of all $5! = 120$ permutations confirms that this is *an* optimal sequence for this instance — the

Table 2: Trace of the insertion heuristic on the illustrative instance ($S = 3$, $P = 4$). At each step every candidate is evaluated on (penalty, makespan) and dominated candidates are discarded.

Step	Candidate	Penalty	Makespan	Retained
1	$[4]$	85	10	yes
2	$[4, 2]$	139	14	yes
	$[2, 4]$	159	14	no (dominated)
3	$[1, 4, 2]$	139	14	yes
	$[4, 1, 2]$	220	16	no (dominated)
	$[4, 2, 1]$	254	15	no (dominated)
4	$[1, 4, 5, 2]$	208	17	yes
	$[1, 4, 2, 5]$	225	18	no (dominated)
	$[1, 5, 4, 2]$	236	19	no (dominated)
5	$[1, 4, 3, 5, 2]$	**288**	21	yes
	$[1, 4, 5, 3, 2]$	288	21	yes
	$[1, 4, 5, 2, 3]$	292	20	yes

optimum 288 is attained by exactly two permutations, $[1, 4, 3, 5, 2]$ and $[1, 4, 5, 3, 2]$, and the heuristic retains both at step 5. On this small example the insertion heuristic is therefore not merely quasi-optimal but exact, while examining only 12 candidate sequences instead of 120.

Complexity of the insertion heuristic. The construction performs N steps. At each step at most P sequences are retained, and each generates at most S insertion candidates, so at most PS candidates are evaluated per step. Evaluating a candidate schedule from scratch requires computing C_{km} for every position-machine pair, at cost $\mathcal{O}(NM)$, which yields an overall bound of $\mathcal{O}(PSN^2M)$. If instead only the suffix affected by the insertion is re-evaluated – feasible because the prefix preceding the last S positions is fixed, and its machine-availability state can be cached – the per-candidate cost falls to $\mathcal{O}(SM)$ and the bound becomes $\mathcal{O}(PS^2NM)$. Since S and P are user-fixed constants independent of instance size, these bounds are $\mathcal{O}(N^2M)$ and $\mathcal{O}(NM)$ respectively. Expressed as a power of the instance size NM, the first bound corresponds to an exponent of exactly 1.5 for square configurations ($N = M$, so $N^2M = (NM)^{1.5}$) and tends to 1.5 for elongated ones; the exponents measured empirically in Section 6.6 fall on either side of this value ($b \in [1.31, 1.67]$), which is consistent with an implementation that re-evaluates a bounded-length suffix rather than the full schedule. Either way, the heuristic is polynomial in N and M, in contrast to the $n!$ size of the search space.

4.2 Iterative Multi-Machine Selection Heuristic

The selection heuristic starts with an exhaustive evaluation of permutations of an initial subset of W_{init} jobs, then extends iteratively using a permutation window of size W. The parameter K limits the number of retained sequences. This approach favors in-depth analysis of the initial ordering at the cost of higher computation.

1. **Initialization**: Sort jobs by a criterion (e.g., ascending d_n); set parameters W_{init}, W, K.
2. **Bootstrapping**: Generate all $W_{\text{init}}!$ permutations of the first W_{init} jobs; evaluate penalty and makespan; retain the K best Pareto-optimal sequences.
3. **Iterative extension**: For each retained prefix, generate permutations of the next W unscheduled jobs; evaluate and apply Pareto filtering; retain K sequences. Repeat until all N jobs are sequenced.

4. **Output**: Select the complete sequence minimizing total penalty (breaking ties by makespan).

Illustrative example. Using the same 5-job, 3-machine instance as above, with $W_{\text{init}} = 3$, $W = 2$, $K = 2$, the bootstrapping phase evaluates all $3! = 6$ permutations of the first three jobs of the EDD list, $[4, 2, 1]$:

Sequence	Penalty	Makespan
$[1, 4, 2]$	139	14
$[1, 2, 4]$	166	15
$[4, 1, 2]$	220	16
$[2, 1, 4]$	244	15
$[4, 2, 1]$	254	15
$[2, 4, 1]$	274	15

Here $[1, 4, 2]$ dominates every other permutation on both criteria simultaneously, so Pareto filtering leaves a single non-dominated prefix even though $K = 2$ sequences could have been retained – an early illustration of how aggressively the filter can prune when one candidate is unambiguously best. The extension step then appends both permutations of the remaining jobs $\{5, 3\}$, giving $[1, 4, 2, 5, 3]$ (penalty 317, makespan 22) and $[1, 4, 2, 3, 5]$ (penalty 326, makespan 23); the former is selected.

Comparing the two heuristics on this instance is instructive: the selection heuristic returns a solution 10.1% above the optimum (317 against 288), whereas the insertion heuristic reaches the optimum exactly. The reason is visible in the trace above – by committing to an exhaustively optimized prefix of W_{init} jobs before any of the remaining jobs are considered, selection locks in the ordering $[1, 4, 2]$, which is optimal for those three jobs in isolation but not part of any optimal complete sequence. This myopia with respect to later arrivals is the structural weakness that the aggregate results of Section 6.1 confirm at scale.

4.3 Hybrid Algorithm

Sections 6.1–6.4 show that neither the insertion heuristic nor NEHedd dominates the other across all instance configurations and traffic levels: insertion is preferable throughout the low-traffic regime and on small to moderate instances at every traffic level, while NEHedd becomes preferable on the largest instances once traffic is medium or high, the two methods being of comparable quality in the intermediate regimes. Because which of the two heuristics will be superior for a *given* instance depends jointly on N, M, and the traffic intensity of job arrivals – and cannot reliably be predicted in advance without running both – we do not attempt a predictive selection rule. Instead, the hybrid algorithm resolves the choice *a posteriori*:

1. **Independent execution**: Run the insertion heuristic (with $S = 15$, $P = 80$, the configuration identified in Sections 6.1–6.2 as giving the best quality-speed trade-off) and NEHedd independently on the given instance, producing two candidate sequences.

2. **Evaluation**: Compute the objective function value (Equation (1)) for both candidate sequences.

3. **Selection**: Retain the sequence with the lower total penalty, breaking ties by makespan.

By construction, the hybrid's total penalty is always less than or equal to the worse of the two component penalties, and equal to the better one: it can never underperform either heuristic taken alone. This is confirmed empirically in Sections 6.2–6.4, where the hybrid achieves the lowest penalty in every tested configuration and traffic level, with average gains of 5.4% over insertion and 0.4% over NEHedd for large, high-traffic instances. These gains are relative

reductions in penalty, computed instance by instance and then averaged, rather than differences of averaged penalties; the two definitions do not coincide when the penalty varies strongly across seeds, as it does here.

The asymmetry between those two figures is not an artefact but a direct consequence of the construction, and it determines how the hybrid should be assessed. Since the hybrid returns the better of the two candidates on each instance, it coincides with whichever component dominates in a given regime, so its measured gain over that component is necessarily small — on large high-traffic instances, where NEHedd wins most instances, the hybrid is close to NEHedd by construction. Reporting the smaller of the two gains would therefore understate the method as badly as reporting the larger would overstate it. The quantity that does not degenerate in this way is the *spread* between the two components: the penalty an analyst pays for committing in advance to the heuristic that turns out, on that instance, to be the worse one. That spread averages 7.5% per instance over the full benchmark, and it is exactly what the hybrid eliminates. Section 6.4 reports it by traffic level. This, rather than a uniform improvement over both components, is the claim the experiments support.

The cost of this improvement is computational: the hybrid's running time is essentially the sum of the insertion heuristic's and NEHedd's running times, since both must be executed in full before a choice can be made (Section 6.6 quantifies this for the insertion heuristic; NEHedd has classical worst-case complexity $\mathcal{O}(N^3M)$). As both components are polynomial in N and M, this sum remains polynomial in instance size, so the hybrid – while the slowest of the three methods tested – remains tractable for all instance sizes considered in this study (up to $N = 120$, $M = 30$). Because the two component runs are independent of one another, they could in principle be executed in parallel (e.g., on separate CPU cores), reducing wall-clock time toward the *maximum*, rather than the *sum*, of the two individual running times; we identify this parallelization as a direction for future work (Section 7.2).

5. EXPERIMENTAL DESIGN

5.1 Experimental Setup

Simulations were performed in Python 3.9 using NumPy and Pandas, with a fixed random seed for reproducibility, on a computer with an Intel Core i7-8650U processor (1.90 GHz base, 4.20 GHz turbo, 4 cores, 8 logical processors) and 16 GB RAM under Windows 11.

All algorithms were implemented in the same code base and share the same schedule-evaluation routine, so that reported penalties are directly comparable and reported times are not affected by differences in implementation quality. Execution time is measured as wall-clock time around the call to each heuristic, excluding instance generation and result serialization, and every algorithm is run single-threaded so that times reflect algorithmic cost rather than differing degrees of parallelism. Because all three methods are deterministic given an instance, a single run per instance suffices; the variability reported throughout Section 6 therefore reflects variability *across instances* (the 100 seeds per configuration), not run-to-run noise. For the hybrid algorithm, the reported time is the sum of its two components' times, since both must complete before a selection can be made.

5.2 Instance Generation

Instances are generated randomly to simulate diverse realistic scenarios, using the distributions summarized in Table 3. Processing times are drawn from an exponential distribution, a standard choice for modeling variable service/processing durations with a long right tail (occasional long jobs), rather than a bounded uniform distribution that would understate variability. Inter-arrival times and slack are drawn from uniform distributions whose upper bound is controlled

by a single average parameter ($iaAvg$ for arrivals, $slackAvg$ for slack), which allows the overall traffic intensity of the shop to be tuned directly (Section 5.3) while keeping the shape of within-scenario variability constant. Due dates are constructed additively from arrival time, total processing time, and slack, so that a due date is always feasible relative to a job's own processing requirements, isolating traffic intensity as the controlled source of scheduling difficulty rather than allowing arbitrarily infeasible due dates to dominate the results. The penalty parameters p_n and q_n are drawn as integers, which is both realistic (fixed contractual penalties and per-day late fees are typically quoted in whole currency units) and convenient for reproducibility. Their relative ranges are deliberately chosen so that neither penalty component is negligible: with p_n averaging about 55 cost units and q_n about 5 cost units per time unit, the fixed component dominates for jobs late by less than roughly 11 time units and the proportional component dominates beyond that.

- **Inter-arrival times** ($iArr_n$): Uniform $U(0, 2 \cdot iaAvg)$; arrival times cumulated from $a_1 = 0$.
- **Processing times** (x_{nm}): Exponential Exp($\lambda = 1.0$) for each job-machine pair.
- **Slack** (z_n): Uniform $U(0, 2 \cdot slackAvg)$ with $slackAvg = 2.0$.
- **Due dates**: $d_n = a_n + \sum_m x_{nm} + z_n$.
- **Fixed penalty** (p_n): discrete uniform on $\{10, 11, \ldots, 99\}$.
- **Tardiness rate** (q_n): discrete uniform on $\{1, 2, \ldots, 9\}$.

Three traffic scenarios are defined by the average inter-arrival time $iaAvg$ relative to the mean processing time ($procAvg = 1.0$):

- **Low traffic**: $iaAvg = 2.0$; inter-arrivals in $U(0, 4)$.
- **Medium traffic**: $iaAvg = 1.0$; inter-arrivals in $U(0, 2)$.
- **High traffic**: $iaAvg = 0.5$; inter-arrivals in $U(0, 1)$.

Table 3: Instance generation parameters.

Variable	Distribution
Inter-arrival times ($iArr_n$)	$U(0, 2 \cdot iaAvg)$, $iaAvg \in \{2, 1, 0.5\}$
Arrival times (a_n)	$a_1 = 0$; $a_n = a_{n-1} + iArr_{n-1}$
Processing times (x_{nm})	Exp($\lambda = 1.0$)
Slack (z_n)	$U(0, 2 \cdot slackAvg)$, $slackAvg = 2.0$
Due date (d_n)	$a_n + \sum_m x_{nm} + z_n$
Fixed penalty (p_n)	Discrete uniform on $\{10, \ldots, 99\}$
Tardiness rate (q_n)	Discrete uniform on $\{1, \ldots, 9\}$

Table 4 shows an example instance with $N = 6$, $M = 2$, medium traffic (seed 42), illustrating the concrete magnitudes produced by these distributions.

5.3 Instance Configurations

Instances are organized into three series, chosen to represent three structurally distinct regimes of the job-to-machine ratio: series 1 has few jobs relative to machines ($N < M$, so each job passes through many more machines than there are competing jobs), series 2 balances the two ($N = M$), and series 3 has many jobs relative to machines ($N > M$, so machines see heavy reuse and queueing effects dominate). Comparing performance across these three regimes tests

Table 4: Example instance: $N = 6$, $M = 2$, medium traffic (seed 42).

Parameter	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6
a_n	0.000	0.494	0.870	2.556	2.644	3.191
x_{n1}	0.571	1.234	0.891	1.859	1.167	1.520
x_{n2}	0.560	1.075	0.990	1.860	1.166	1.520
d_n	1.204	3.320	2.773	9.506	6.670	7.439
p_n	52	47	44	61	41	75
q_n	9	1	4	6	5	2

whether a heuristic's advantage is structural (tied to the shape of the instance) or incidental (tied only to raw size).

- **Series 1 ($N < M$)**: $(N, M) \in \{(20, 30), (30, 45), (40, 60), (50, 75), (60, 90), (70, 105)\}$.
- **Series 2 ($N = M$)**: $(N, M) \in \{(25, 25), (35, 35), (45, 45), (55, 55), (65, 65), (75, 75)\}$.
- **Series 3 ($N > M$)**: $(N, M) \in \{(40, 10), (60, 15), (80, 20), (100, 25), (120, 30)\}$.

For each configuration and traffic level, 100 instances are generated with seeds 0–99, yielding 5,100 instances in total; using 100 independent seeds per configuration/traffic combination allows the mean and percentile statistics reported in Section 6 to be estimated with acceptably low sampling variability, rather than relying on a single realization per scenario.

5.4 Algorithm Parameters

- **Insertion heuristic**: $S \in \{15, 25\}$; $P \in \{60, 80\}$.
- **Selection heuristic**: $W_{\text{init}} = W$; $W \in \{6, 7\}$; $K \in \{15, 30, 40\}$. The size of the bootstrapping subset is set equal to the extension window, so that a single parameter W controls the depth of exhaustive search at every stage; note that the $W!$ growth of the bootstrapping phase is what makes values of W beyond 7 impractical.
- **NEHedd**: no tunable parameters.

These parameter ranges were chosen from preliminary pilot runs as representative points spanning a modest-to-generous allocation of search effort. A larger S, P, W, or K can only weakly improve solution quality, since each enlarges the set of candidates examined; in practice, as Section 6.1.4 shows, S, P and K turn out to be saturated over these ranges, so the improvement is nil while the cost is not. A fully systematic tuning of (S, P, W, K) was outside the scope of this study and is identified as a direction for future work (Section 7.2). Table 5 lists the tested parameter configurations for the three representative instance shapes used in Section 6.1–6.2.

Table 5: Tested parameter configurations.

Configuration	N	M	Insertion (S, P)	Selection (W, K)
Elongated	40	10	$(15, 60), (25, 60), (15, 80)$	$(6, 30), (7, 15), (6, 40)$
Square	25	25	$(15, 60), (25, 60), (15, 80)$	$(6, 30), (7, 15), (6, 40)$
Rectangular	20	30	$(15, 60), (25, 60), (15, 80)$	$(6, 30), (7, 15), (6, 40)$

6. RESULTS AND FINDINGS

Throughout this section, "penalty" refers to the objective function value defined in Equation (1) (Section 3.3) i.e. every penalty figure reported below is a direct fitness-function evaluation of the corresponding candidate schedule, not a proxy metric. Error bars in all figures denote the 5th–95th percentile range across the 100 seeded instances generated for each configuration/traffic combination (Section 5.3), so that both the central tendency and the variability of each method are visible.

6.1 Insertion vs. Selection

6.1.1 Low Traffic

Under low traffic the two heuristics are of essentially equal solution quality insertion's mean penalty is 0.6% below selection's with $S = 15$, $P = 80$, an advantage smaller than the spread across seeds but they are not of comparable cost: insertion runs 18 times faster (Table **??**). Selection is also markedly less stable: its standard deviation across the 100 seeds is 1.9 times insertion's at this traffic level. This gap is worth stating explicitly, because it is the one dimension on which the two heuristics differ sharply even under low traffic, where their mean penalties nearly coincide. Two methods of equal expected quality are not interchangeable if one of them produces a much wider distribution of outcomes: for a production planner, the relevant risk is the bad instance, not the average one. This pattern is consistent with the structural difference between the two heuristics: under low traffic, generous slack means that most jobs have ample room to be scheduled without becoming late, so the last-S-positions restriction used by insertion rarely excludes the positions that actually matter, while selection's more expensive windowed exhaustive search buys little additional quality for the extra computation it spends. For low traffic, insertion therefore provides the best combination of speed and solution quality among the two non-hybrid heuristics; and because the quality difference is negligible here while the time difference is an order of magnitude, low traffic is the regime in which selection is least defensible, not the one in which it is most competitive. Figures 2 and 3 compare insertion and selection for low-traffic instances across the elongated, square, and rectangular configurations of Table 5.

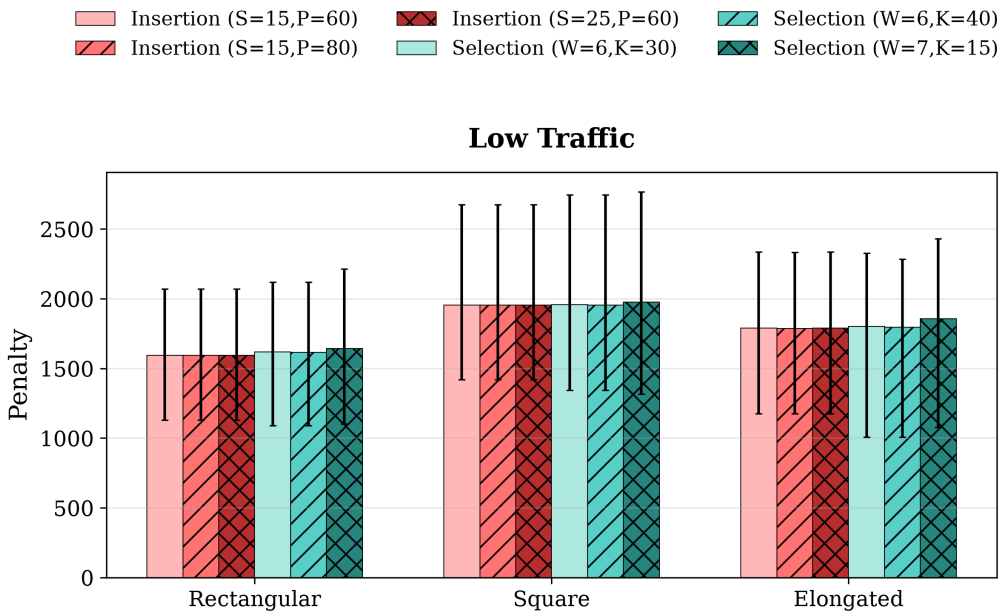


Figure 2: Total penalty, low traffic. Bars are grouped by algorithm and instance shape; within each group the three bars correspond to the three parameter settings of Table 5. Error bars span the 5th–95th percentiles over 100 seeded instances.

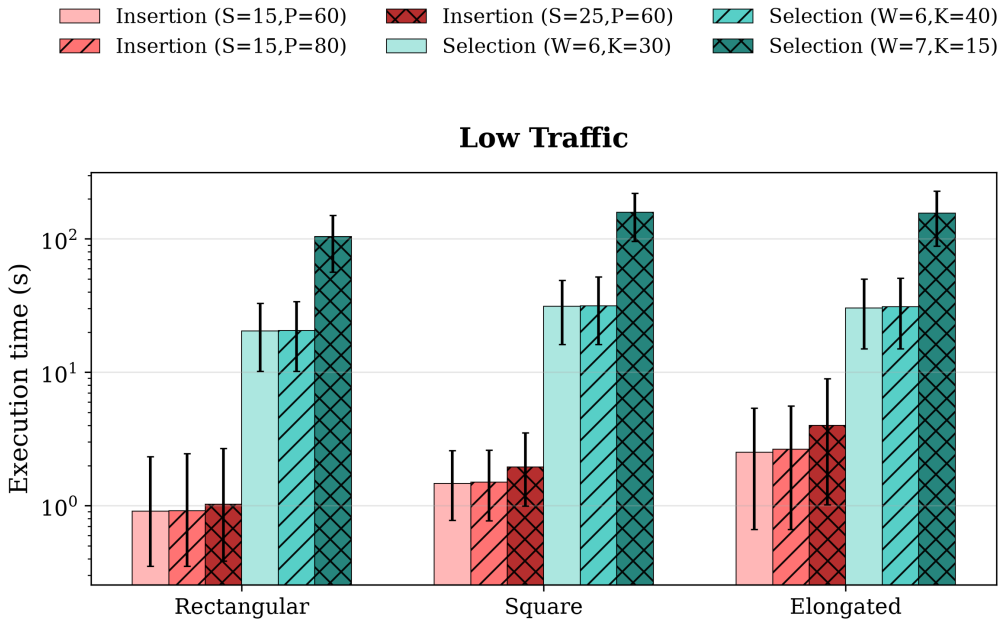


Figure 3: Execution time (s), low traffic. Same grouping as Figure 2.

6.1.2 Medium Traffic

Insertion leads selection by 8.4% in mean penalty under medium traffic while remaining 16 times faster. The three insertion parameter settings of Table 5 are indistinguishable from one another on penalty, so this advantage is a property of the heuristic and not of a particular tuning (see Section 6.1.4). Selection shows greater dispersion and suffers from a more pronounced increase in execution time as instance size grows, because the number of permutations evaluated within each window of size W grows factorially in W while the number of windows grows linearly in N; as traffic tightens, more of those windows contain jobs whose relative order materially affects

the schedule's tardiness, so selection's execution-time growth is not matched by a proportional quality gain. Insertion maintains its advantage in both quality and speed under medium traffic, as shown in Figures 4 and 5.

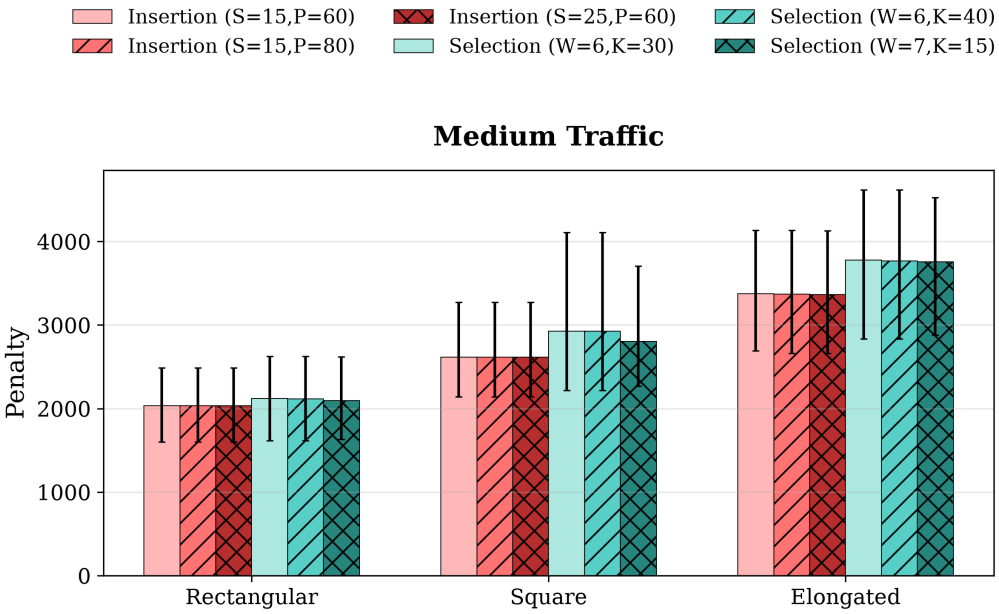


Figure 4: Total penalty, medium traffic. Same grouping as Figure 2.

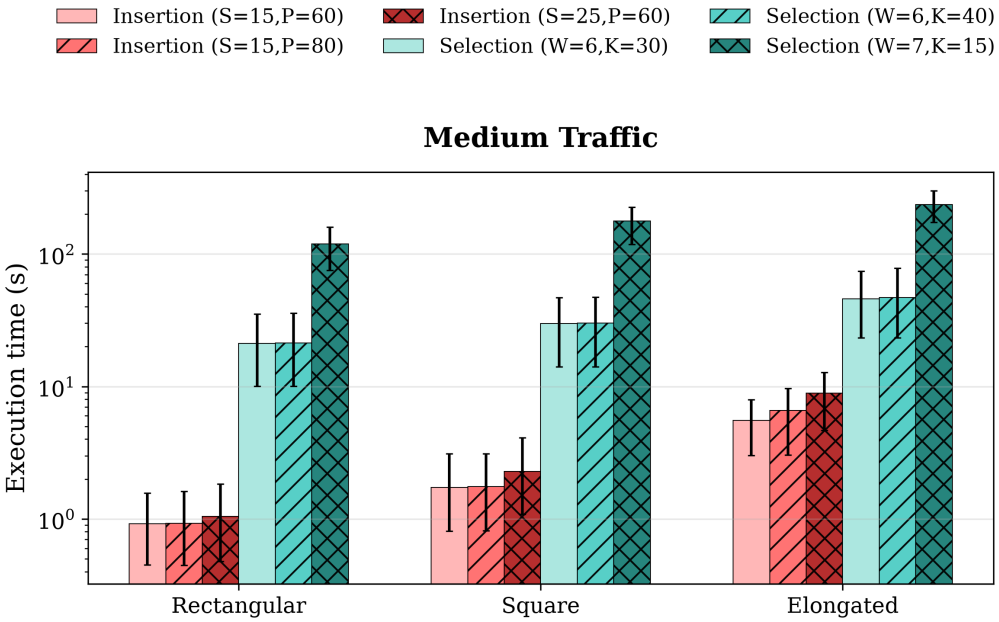


Figure 5: Execution time (s), medium traffic. Same grouping as Figure 2.

6.1.3 High Traffic

Under high traffic, insertion maintains systematically lower penalties and remains scalable, while selection becomes less competitive and significantly slower, limiting its applicability for large instances under tight scheduling conditions, as Figures 6 and 7 show. This is where the two heuristics separate most clearly: insertion's penalty advantage reaches 12.5%, its largest of the three traffic levels, while it remains 14 times faster. The mechanism is the same one that already penalised selection under medium traffic, but amplified. When inter-arrival times fall below the mean processing time, jobs queue at the first machine and almost every job becomes a candidate for lateness, so the penalty of a complete sequence depends on the relative order of jobs that are far apart in the schedule, not only on the order within a window of W consecutive jobs. Selection's exhaustive search is confined to such a window and therefore optimises a criterion that has become progressively less informative about the final objective, while still paying the $W!$ cost of that search at every step. Insertion, by contrast, re-evaluates the whole retained sequence at each construction step and keeps a Pareto set of alternatives, so the tightening of the schedule degrades its solution quality far more slowly than it degrades selection's.

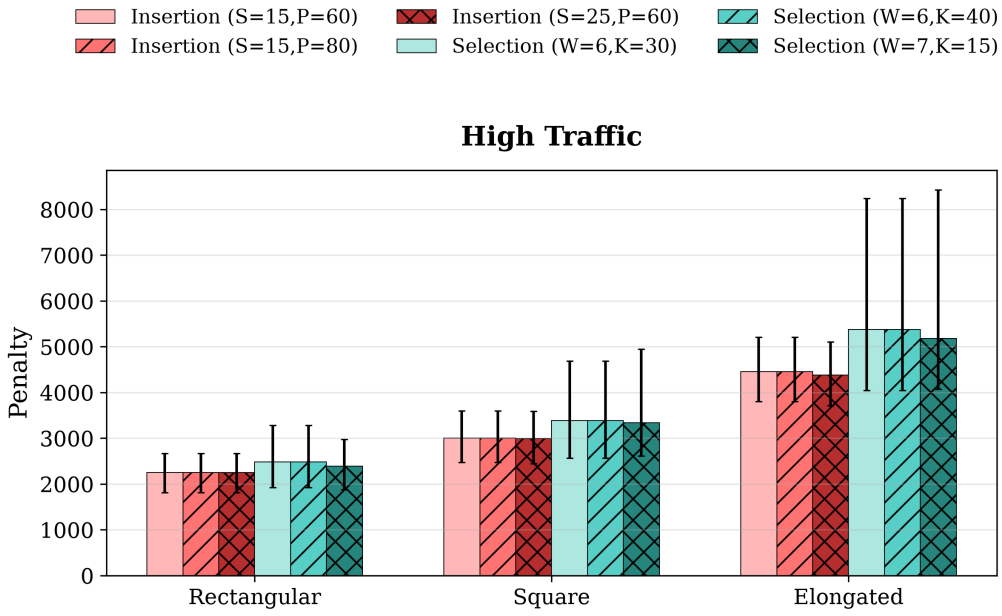


Figure 6: Total penalty, high traffic. Same grouping as Figure 2.

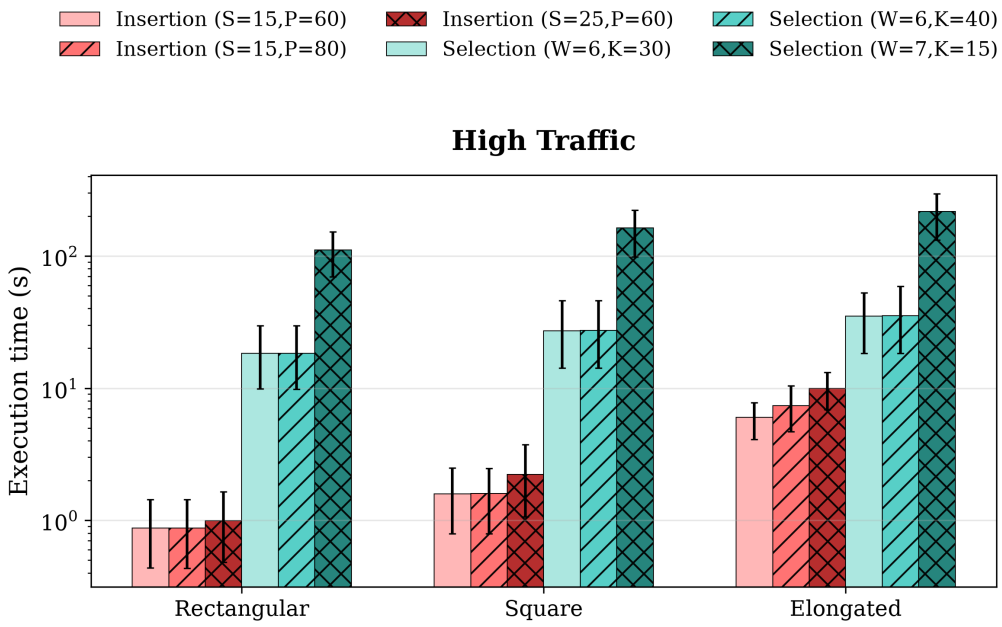


Figure 7: Execution time (s), high traffic. Same grouping as Figure 2.

6.1.4 Difference Analysis

For 20×30 low-traffic instances, Selection$(6, 40)$ obtains slightly lower penalties than Insertion (15,80) in the majority of cases, but with greater variability, including a small number of instances with markedly worse outcomes (the long right tail visible in Figure 8). This does not contradict the aggregate ranking of Table 6, and the distinction matters enough to be made explicit. Table 6 reports which method has the lower *mean* penalty over all low-traffic instances and all three instance shapes, whereas the comparison here is restricted to a single shape (20×30) and counts the *number* of instances on which each method wins. Selection can win more often while still having the higher mean, precisely because its losses, when they occur, are much larger than its gains; this is what the asymmetric tail in Figure 8 shows. The two statements are therefore consistent, and together they make the practical point that selection's median-case advantage on this one configuration is not worth its tail risk or its additional computation. Notably, increasing W from 6 to 7 in Selection produces no significant improvement in solution quality (Figure 9), indicating that this parameter increase does not justify its additional computational cost. Figures 8 and 9 show these penalty differences for 20×30 low-traffic instances, isolating the effect of the selection heuristic's window size W from the effect of switching heuristics entirely, and Table 6 summarizes the main trends across traffic levels.

The variability gap follows the same monotone pattern as the quality gap: selection's standard deviation is 1.9 times insertion's under low traffic, 2.0 times under medium traffic and 2.1 times under high traffic. Insertion's advantage over selection therefore grows with traffic intensity on both dimensions at once — lower mean penalty and tighter dispersion — while its speed advantage, though it narrows slightly, remains above an order of magnitude throughout. No traffic regime tested reverses any of these three orderings.

A second observation cuts across all three traffic levels and concerns the parameters themselves. Within each algorithm, the three settings of Table 5 produce penalties that are indistinguishable at the resolution of Figures 2–7: raising S from 15 to 25, P from 60 to 80, or K from 30 to 40 leaves the mean penalty unchanged in every scenario. The interpretation is that these three parameters are already past the point of saturation over the ranges tested. For P and K, which

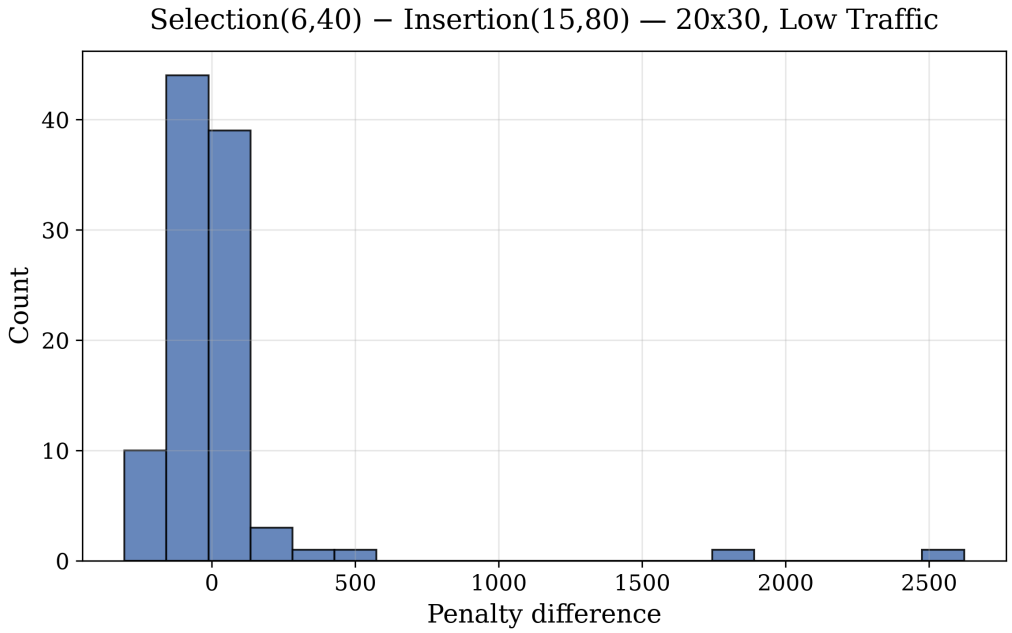


Figure 8: Penalty difference distribution for 20×30 low-traffic instances: Selection$(6, 40)$ minus Insertion$(15, 80)$.

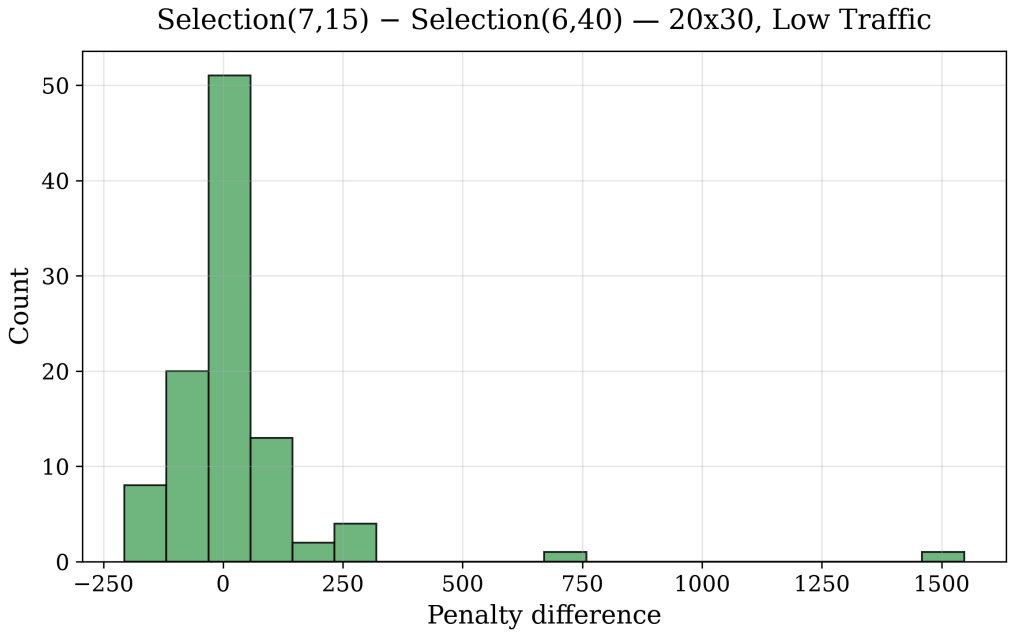


Figure 9: Penalty difference distribution for 20×30 low-traffic instances: Selection$(7, 15)$ minus Selection$(6, 40)$.

cap the number of retained Pareto-optimal sequences, this is expected: the non-dominated set is usually much smaller than the cap, so the cap is rarely binding — the illustrative instance of Section 4.2, where Pareto filtering leaves a single prefix although $K = 2$ were permitted, is the same phenomenon at small scale. This also explains why $K = 30$ and $K = 40$ yield not merely similar but identical execution times: when the cap never binds, the two settings execute the same computation. For S, saturation means that the useful insertion positions all lie within the last 15, so widening the window to 25 only adds candidates that Pareto filtering immediately discards.

The one parameter that does change behaviour is W, and it changes it in the wrong direction: raising W from 6 to 7 multiplies selection's execution time by roughly five — the $W!$ growth of the bootstrapping phase — for no measurable penalty reduction. Taken together, these two observations mean that the quality ranking reported here is robust to the parameter choices within the ranges tested, and that the practical tuning question is not how to set S, P or K but whether any value of W justifies selection's cost.

Table 6: Comparative summary: insertion vs. selection. Quantitative values for the three scenarios of Table 5 are given in Table **??**; the speed advantage of insertion ranges from 14 to 18 times, not a marginal one.

Traffic	**Best Penalty**	**Fastest**	**Stability**
Low	Insertion	Insertion	Selection ×1.9 more variable
Medium	Insertion	Insertion	Selection ×2.0 more variable
High	Insertion	Insertion	Selection ×2.1 more variable

6.2 Insertion vs. NEHedd and Hybrid

We now compare insertion ($S = 15$, $P = 80$), NEHedd, and the hybrid algorithm across the three configuration types of Section 5.3.

6.2.1 Rectangular Configurations ($N < M$)

Under low traffic, insertion attains a lower mean penalty than NEHedd at *every* configuration tested, from 20×30 to 70×105: the advantage is largest in relative terms on the smallest instances and narrows steadily as N and M grow, but it never reverses within the range studied. Under medium and high traffic the ordering does change with size: insertion still leads on small and moderate instances, and NEHedd overtakes it only on the largest configurations of the series (at 70×105 under medium traffic, and from 60×90 onwards under high traffic). The mechanism behind this reversal is that insertion's last-S-positions restriction is harmless while slack is generous, since the positions it excludes are ones no good solution would use anyway; once arrivals tighten and the sequence fills up, the best position for a job is more often outside that window, and NEHedd's exhaustive positional search recovers exactly the placements insertion cannot see. The hybrid yields the lowest penalties throughout, by construction (Section 4.3). NEHedd is significantly faster at every size; the hybrid, running both components, is the slowest. Figures 10–15 present penalties and execution times, respectively, for $(N, M) \in \{(20, 30), \ldots, (70, 105)\}$, one figure per traffic level.

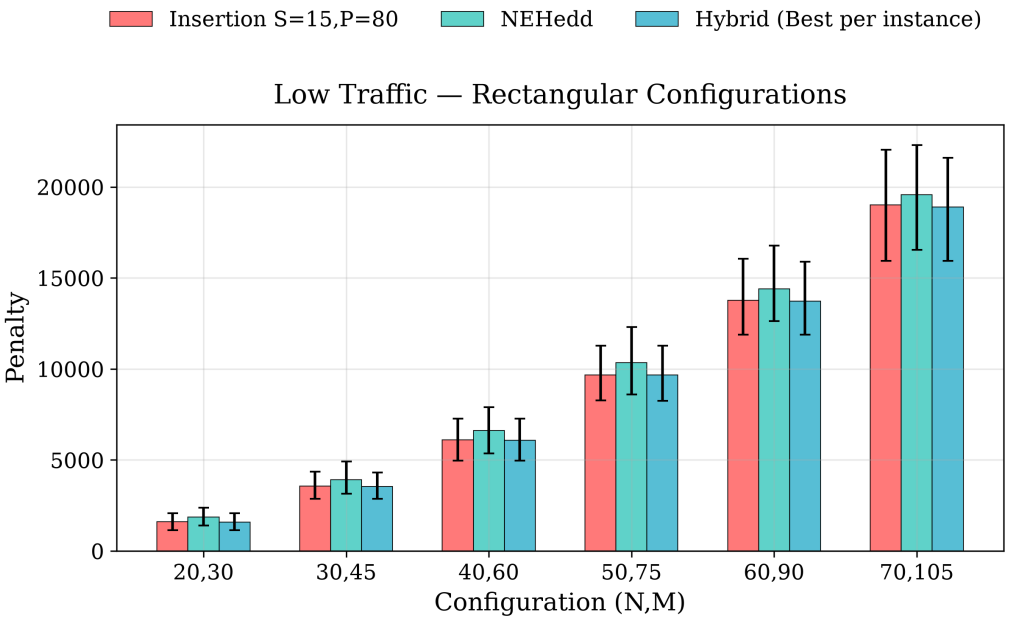


Figure 10: Penalties for rectangular configurations ($N < M$), low traffic.

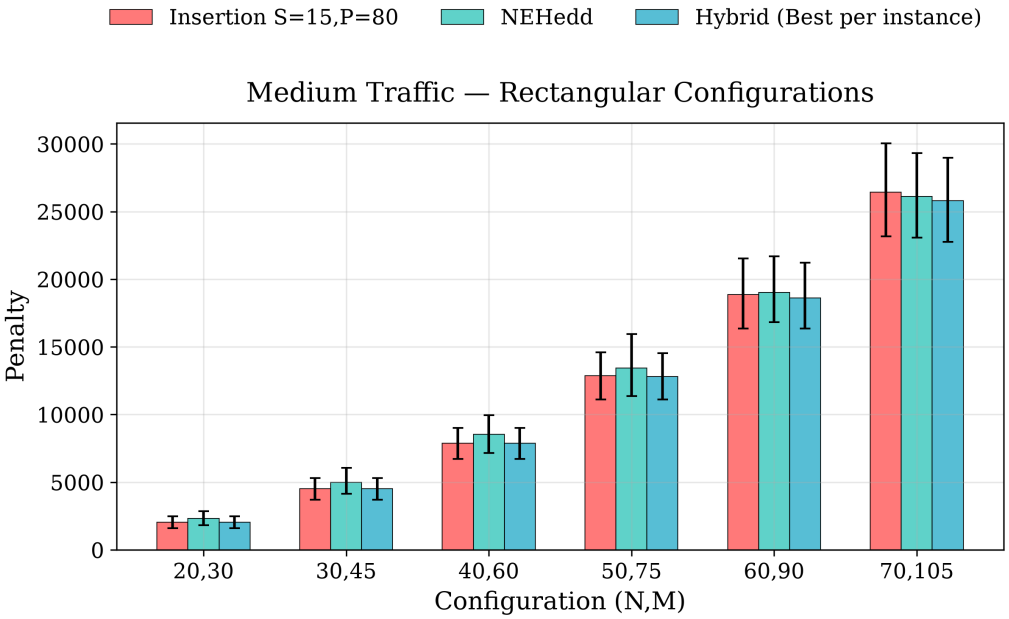


Figure 11: Penalties for rectangular configurations ($N < M$), medium traffic.

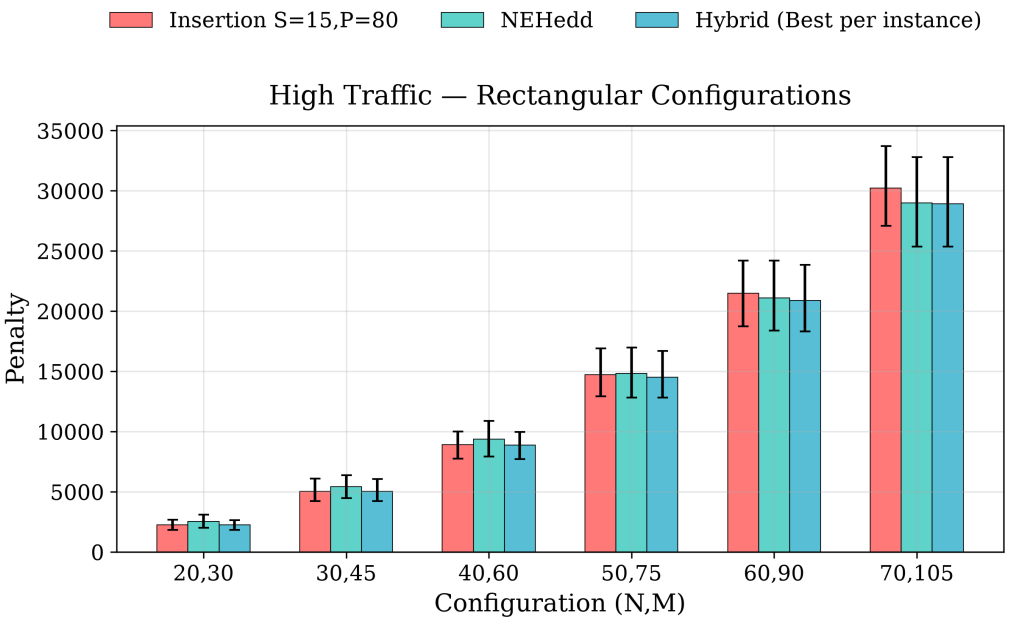


Figure 12: Penalties for rectangular configurations ($N < M$), high traffic.

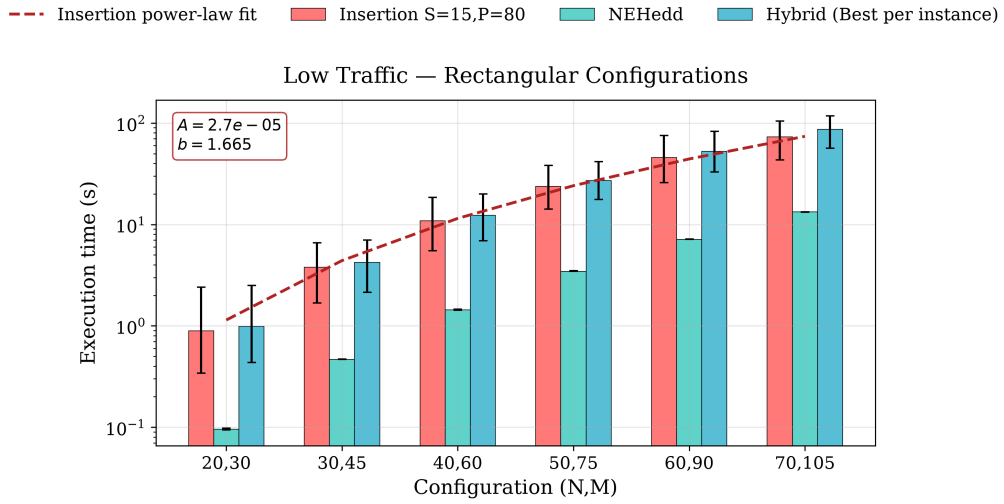


Figure 13: Execution times (log scale) for rectangular configurations, low traffic, with polynomial fit.

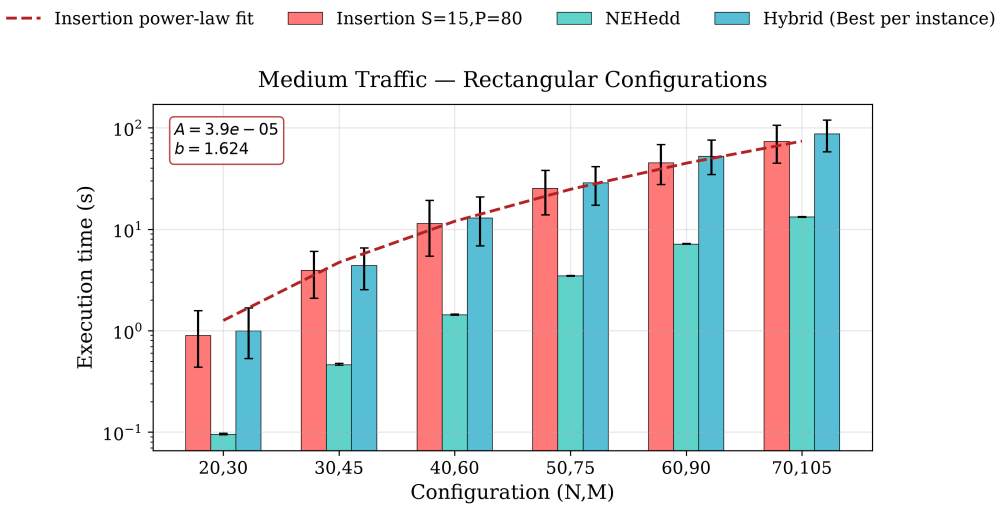


Figure 14: Execution times (log scale) for rectangular configurations, medium traffic, with polynomial fit.

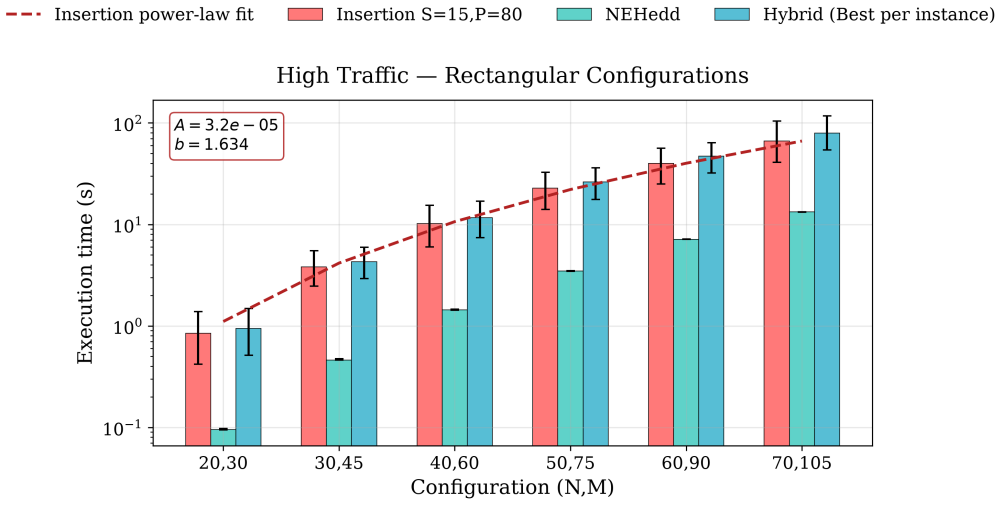


Figure 15: Execution times (log scale) for rectangular configurations, high traffic, with polynomial fit.

6.2.2 Square Configurations ($N = M$)

The square series reproduces the rectangular pattern with the crossover shifted by roughly one configuration. Insertion leads at every size under low traffic; under medium traffic NEHedd catches up only at 75×75, and under high traffic from 65×65 onwards. On the small and moderate configurations, insertion's margin over NEHedd is of the same relative order as in the rectangular series, which indicates that the advantage is driven by traffic intensity and absolute size rather than by the job-to-machine ratio as such. The hybrid maintains its penalty advantage. Result variability (P5–P95 interval) increases with instance size, reflecting the larger combinatorial search space and the correspondingly larger scope for a single early scheduling decision to cascade into materially different total penalties across random seeds, as illustrated in Figures 16–21.

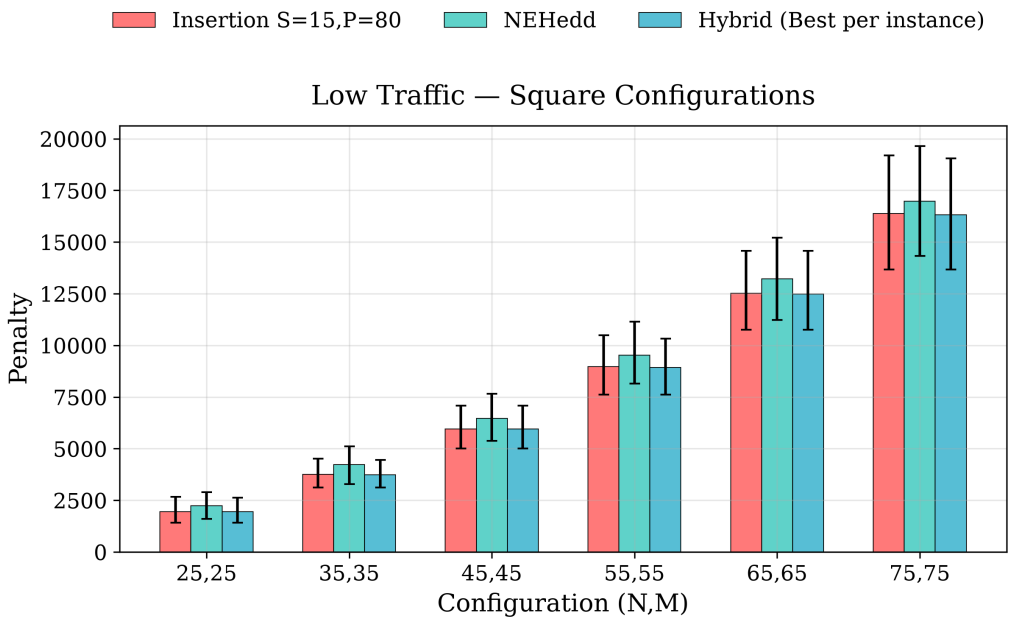


Figure 16: Penalties for square configurations ($N = M$), low traffic.

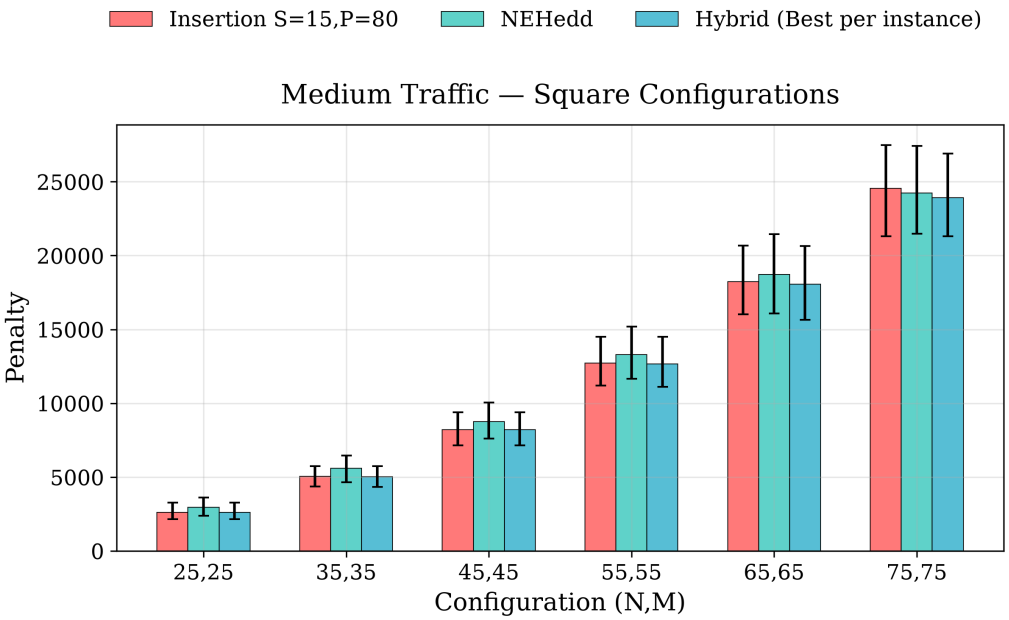


Figure 17: Penalties for square configurations ($N = M$), medium traffic.

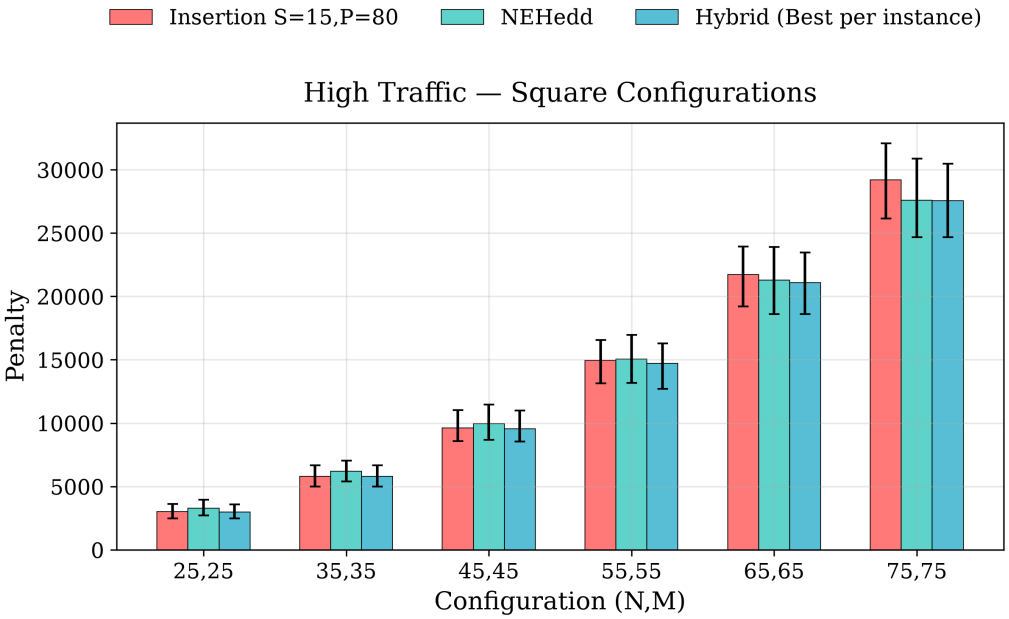


Figure 18: Penalties for square configurations ($N = M$), high traffic.

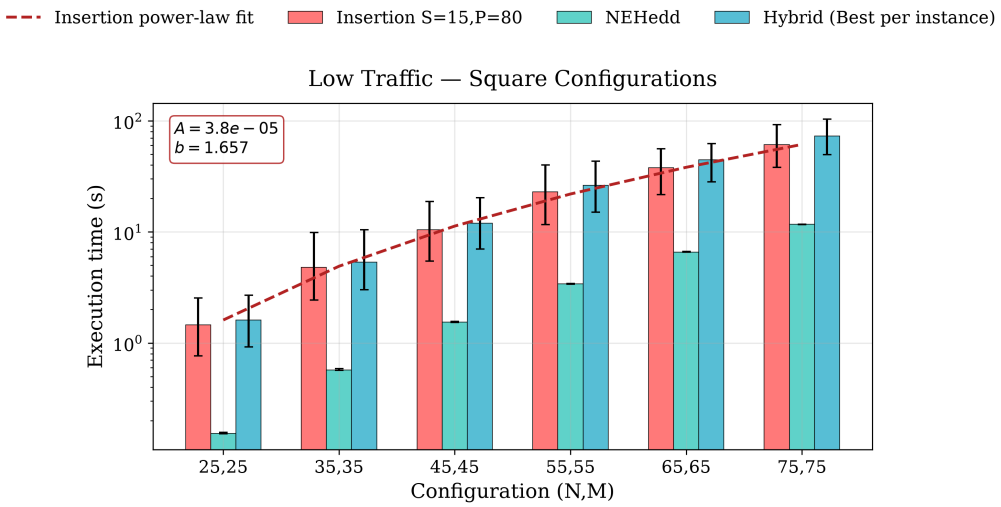


Figure 19: Execution times (log scale) for square configurations, low traffic, with polynomial fit.

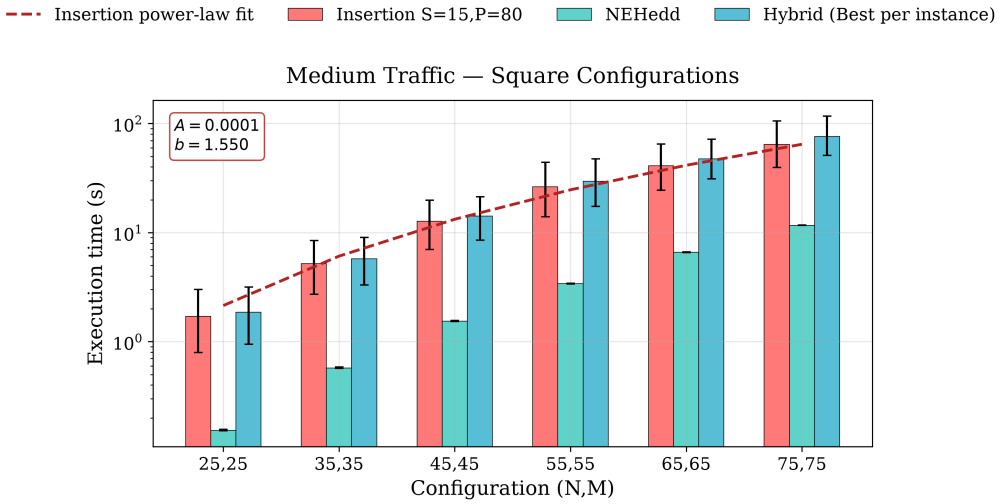


Figure 20: Execution times (log scale) for square configurations, medium traffic, with polynomial fit.

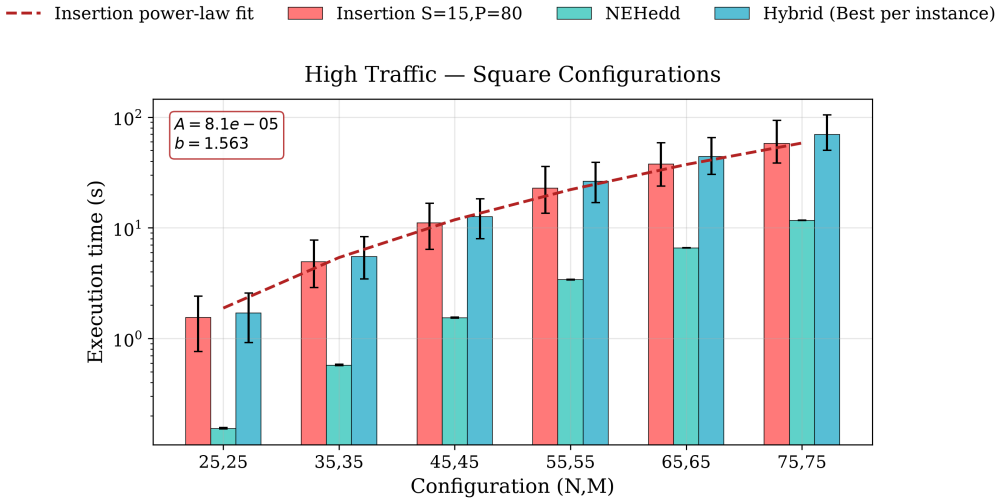


Figure 21: Execution times (log scale) for square configurations, high traffic, with polynomial fit.

6.2.3 Elongated Configurations ($N > M$)

The elongated series shows the sharpest reversal of the three. Under low traffic, insertion again leads at every configuration up to 120×30. Under medium traffic the two methods are close up to 100×25 and NEHedd takes the lead at 120×30; under high traffic the crossover occurs earlier, at 80×20, and NEHedd's margin then widens sharply at 100×25 and 120×30 — the largest gap between the two methods anywhere in the study. This is more pronounced here than in the rectangular or square cases because a large N relative to M means many jobs compete for a comparatively small number of machines, so queueing dominates and the cost of any single suboptimal placement early in the sequence is amplified by every job that follows it. The hybrid obtains the lowest penalties throughout. NEHedd maintains a clear speed advantage; the hybrid is the slowest. These effects are visible in Figures 22–27.

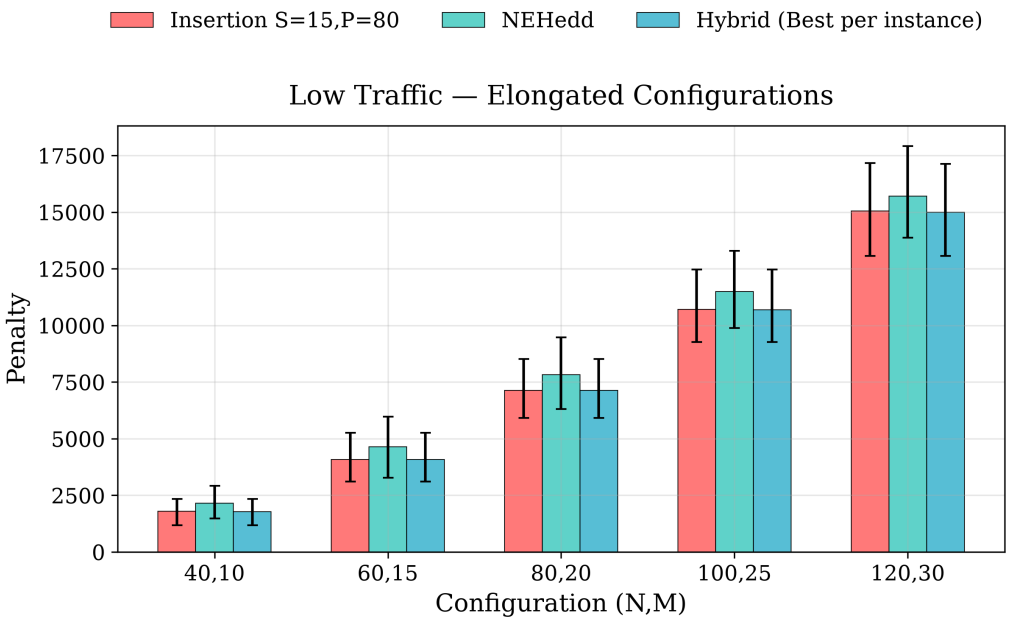


Figure 22: Penalties for elongated configurations ($N > M$), low traffic.

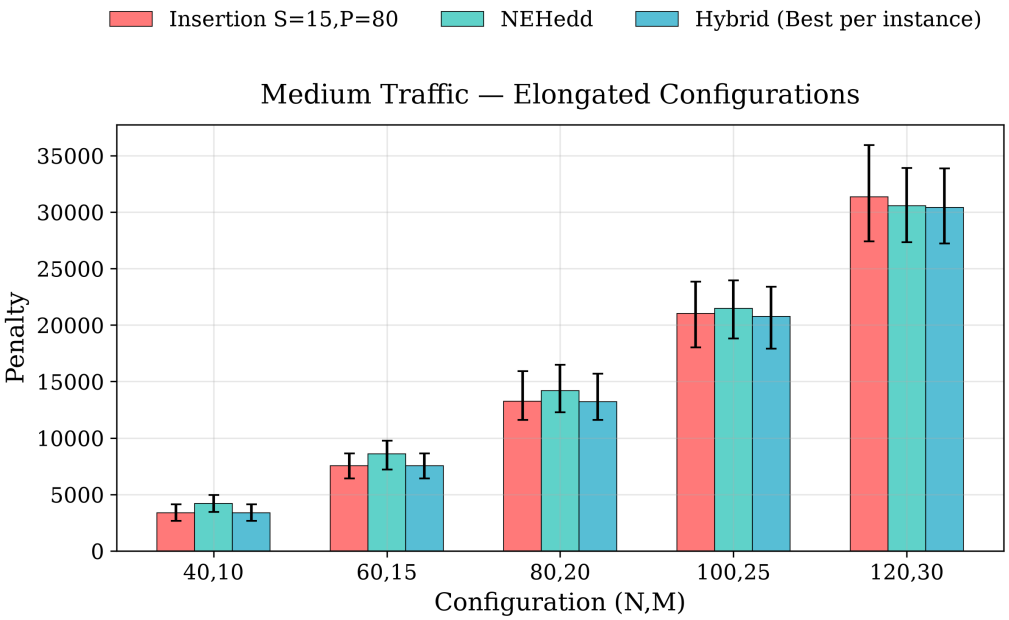


Figure 23: Penalties for elongated configurations ($N > M$), medium traffic.

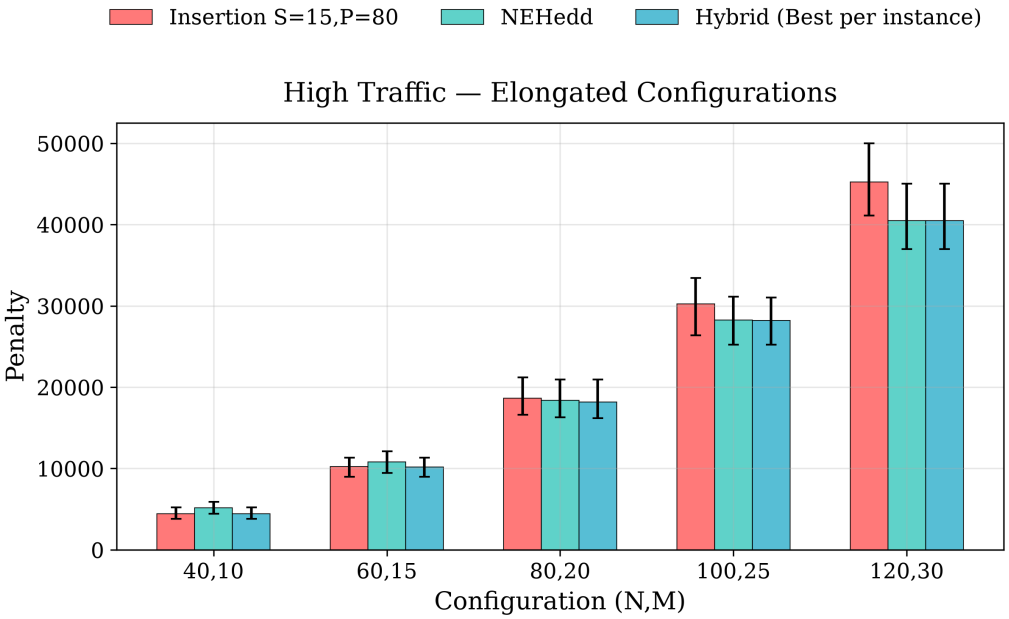


Figure 24: Penalties for elongated configurations ($N > M$), high traffic.

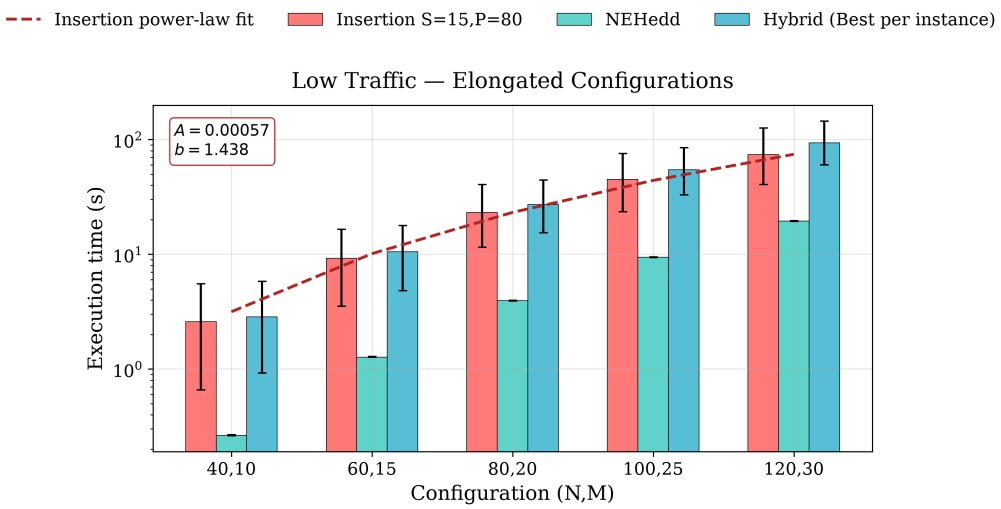


Figure 25: Execution times (log scale) for elongated configurations, low traffic, with polynomial fit.

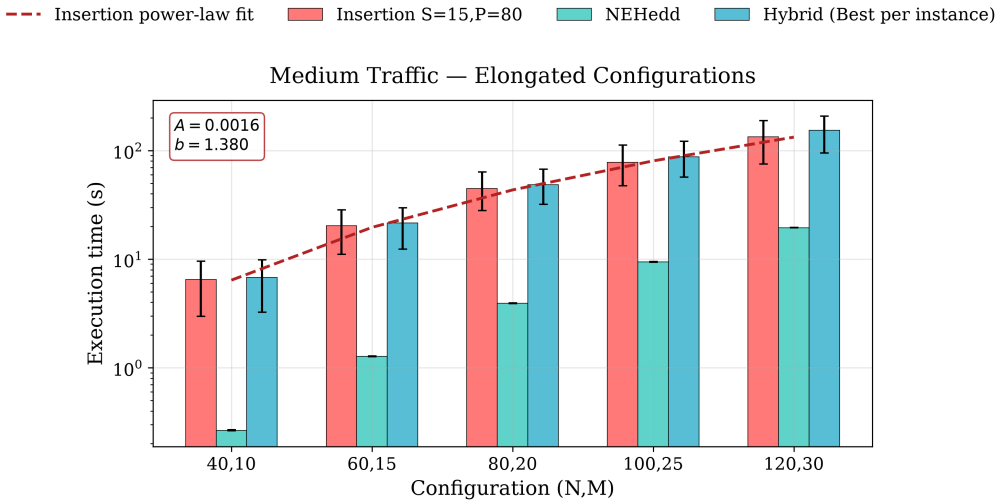


Figure 26: Execution times (log scale) for elongated configurations, medium traffic, with polynomial fit.

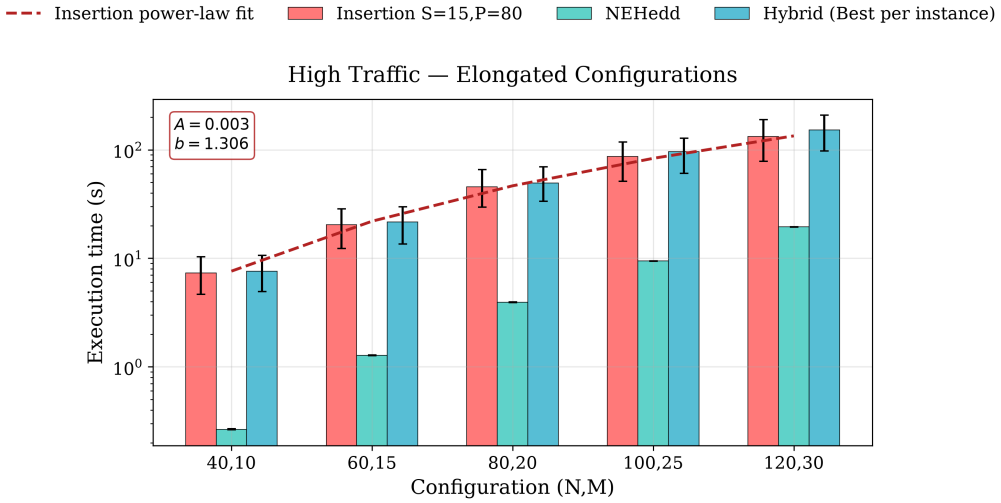


Figure 27: Execution times (log scale) for elongated configurations, high traffic, with polynomial fit.

6.3 Penalty Gap Analysis

Across all three configuration types the mean per-instance difference (Insertion − NEHedd) is distinctly negative — insertion better — on the small and moderate configurations, and it stays negative at every size under low traffic. It crosses zero only under medium and high traffic, and then only on the largest one to three configurations of each series, becoming most strongly positive for the elongated series at large N, consistent with the mechanism discussed in Section 6.2.3.

Two features of Figures 28–36 deserve emphasis because they are what justifies the hybrid. First, the sign of the mean difference changes gradually with size and traffic rather than switching at an identifiable threshold, so no simple rule of the form "use NEHedd above $NM = c$" would separate the two regimes cleanly. Second, and more importantly, the P5–P95 interval straddles zero at almost every point plotted, including those where the mean is clearly of one sign: even in configurations where one heuristic wins on average, the other still wins on a substantial minority of individual instances. This dispersion, and not the position of the means, is the quantitative content of the near-50/50 split reported in Table 7, and it is why the hybrid is presented not as a competing third heuristic but as a practical resolution of the trade-off documented in this subsection: it captures the better of the two on every instance, including those in the tails where a fixed choice would lose the most. Figures 28–36 show these per-instance penalty differences, which make explicit how the balance between the two heuristics shifts continuously with instance size and traffic, rather than switching abruptly at some threshold.

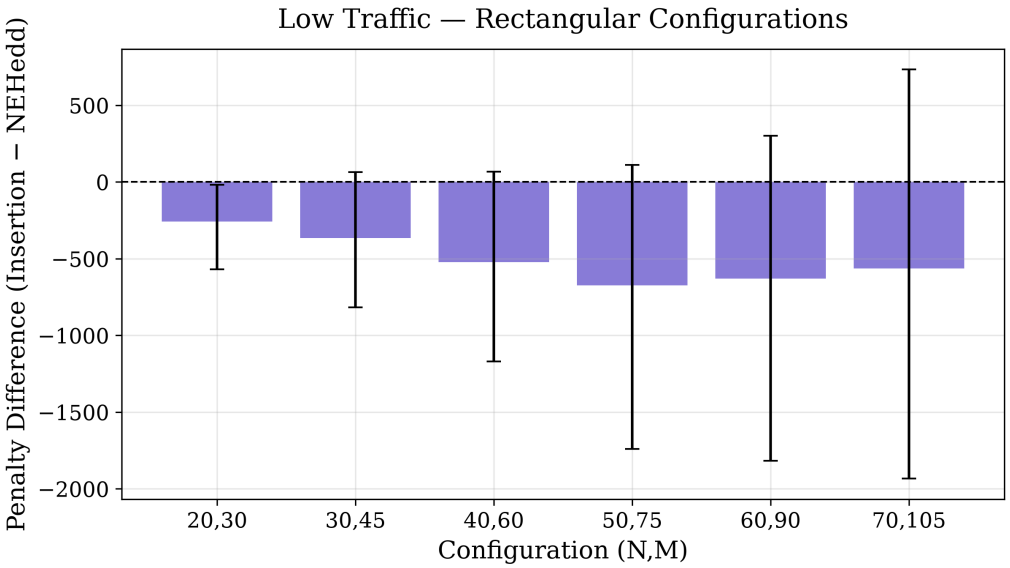


Figure 28: Penalty differences (Insertion − NEHedd) for rectangular configurations, low traffic.

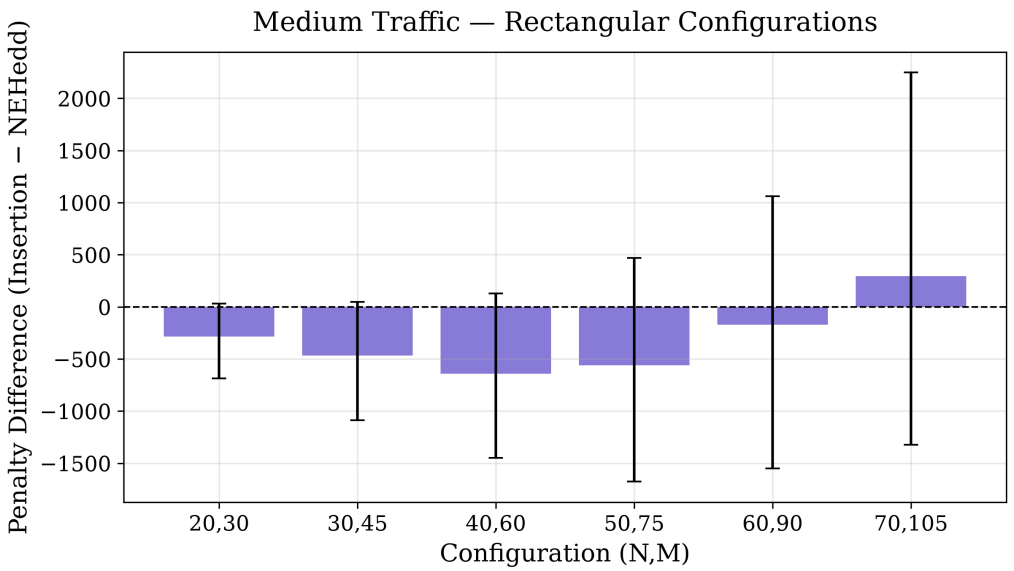


Figure 29: Penalty differences (Insertion − NEHedd) for rectangular configurations, medium traffic.

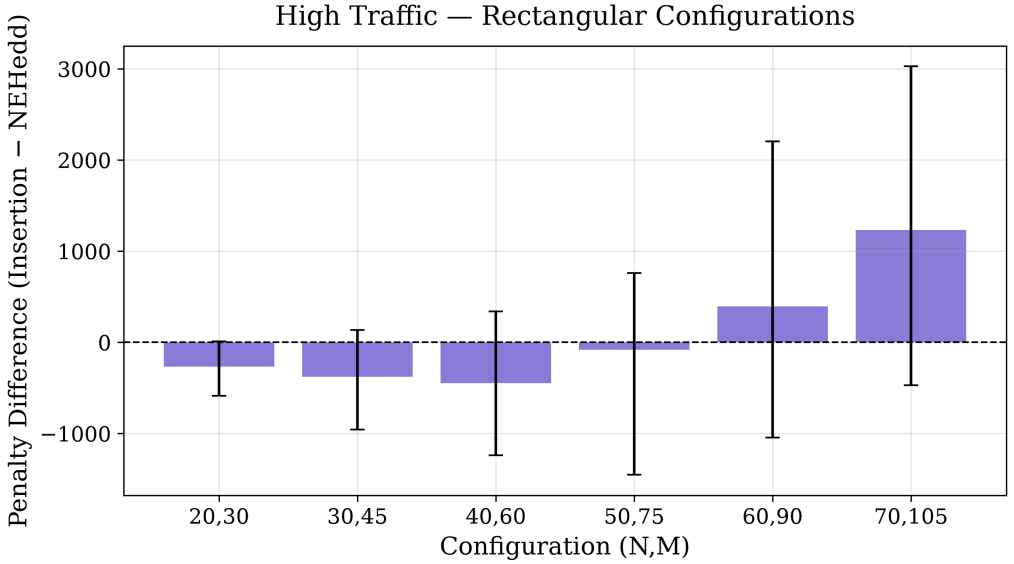


Figure 30: Penalty differences (Insertion − NEHedd) for rectangular configurations, high traffic.

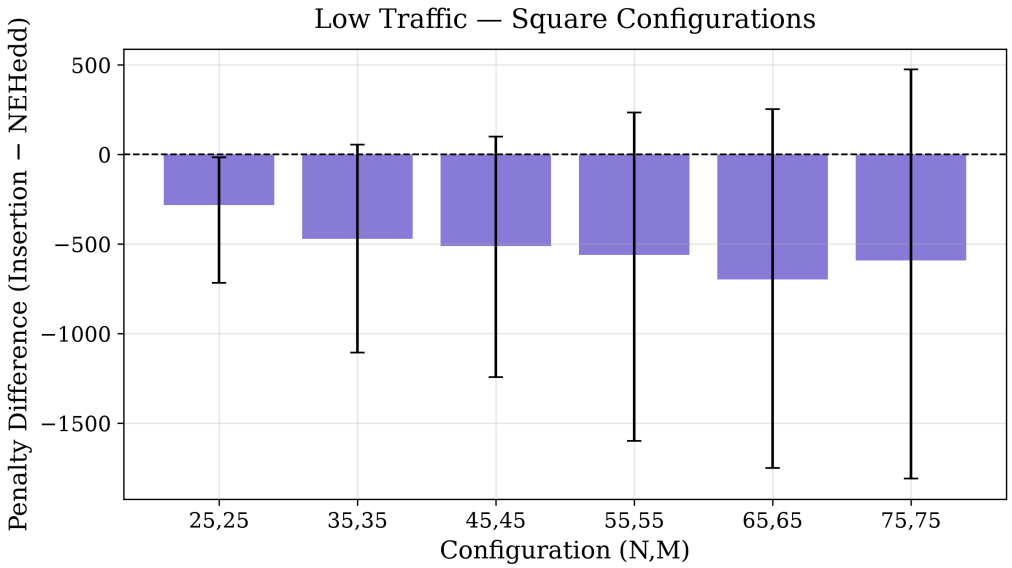


Figure 31: Penalty differences (Insertion − NEHedd) for square configurations, low traffic.

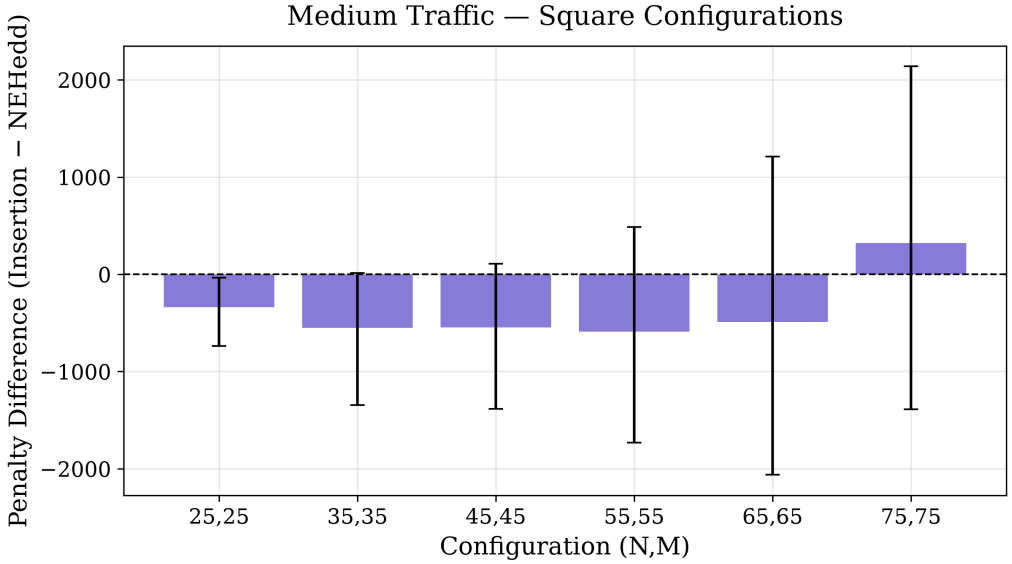


Figure 32: Penalty differences (Insertion − NEHedd) for square configurations, medium traffic.

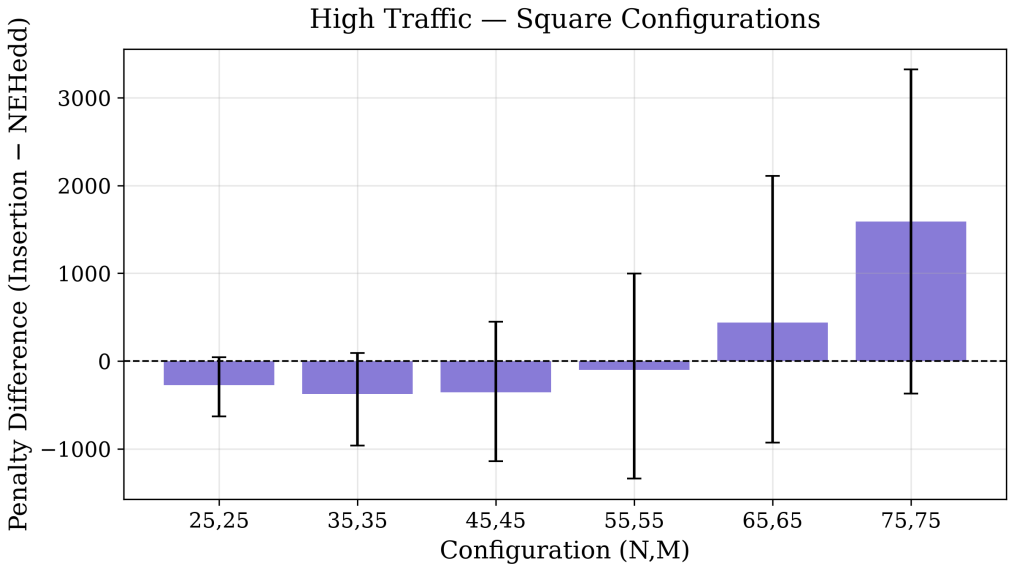


Figure 33: Penalty differences (Insertion − NEHedd) for square configurations, high traffic.

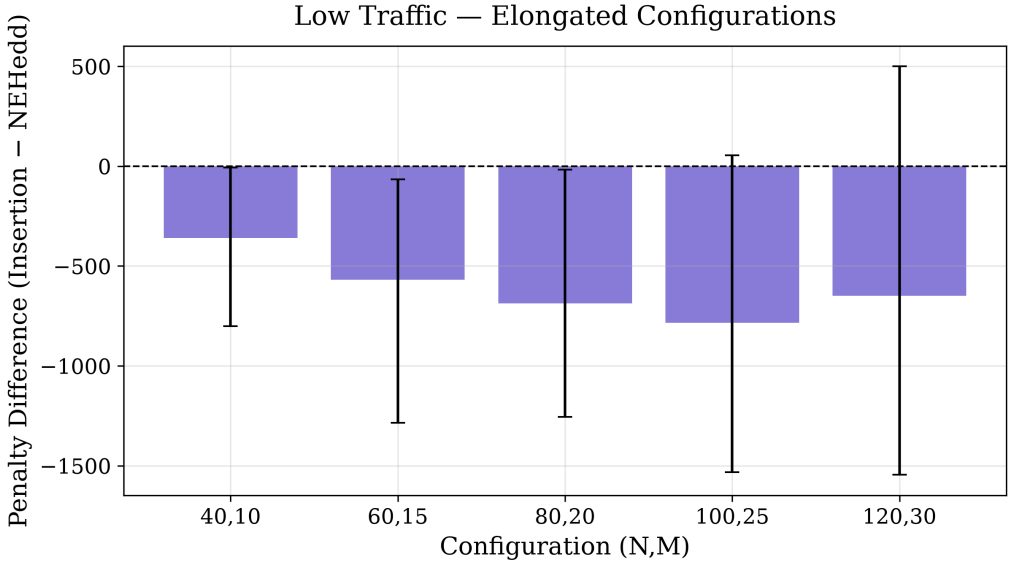


Figure 34: Penalty differences (Insertion − NEHedd) for elongated configurations, low traffic.

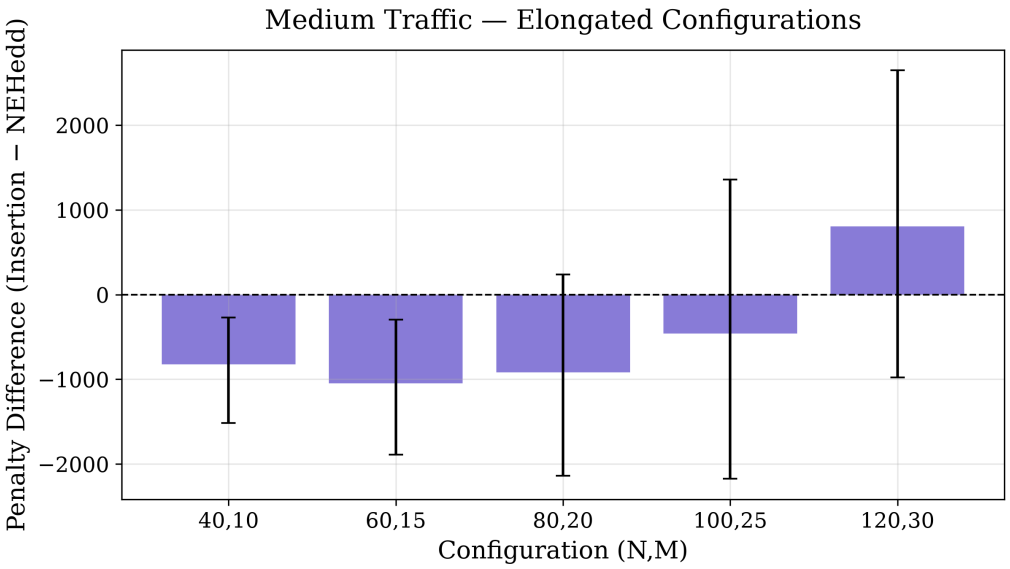


Figure 35: Penalty differences (Insertion − NEHedd) for elongated configurations, medium traffic.

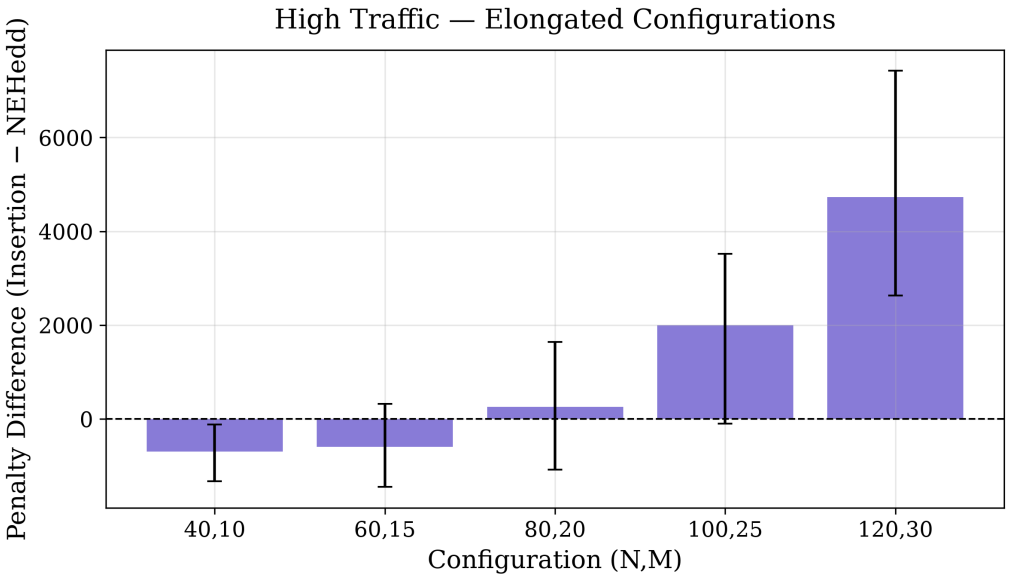


Figure 36: Penalty differences (Insertion − NEHedd) for elongated configurations, high traffic.

6.4 Hybrid Algorithm Statistics

Table 7 reports, for each traffic level, the proportion of instances where insertion or NEHedd produced the better solution (and was therefore selected by the hybrid). This table is the quantitative basis for the complementarity argument introduced in Section 1 and used to justify the hybrid design in Section 4.3.

Table 7: Hybrid algorithm: fraction of instances where each component is selected.

Traffic	Insertion (%)	NEHedd (%)	Instances
Low	52.3	47.7	1700
Medium	49.8	50.2	1700
High	45.6	54.4	1700
Total	49.2	50.8	5100

Under low traffic, insertion is selected slightly more often (52.3%), reflecting its competitiveness in less constrained conditions. Under high traffic, NEHedd dominates (54.4%). Overall, the two algorithms are nearly equal (50.8% versus 49.2%), confirming their complementarity and justifying the hybrid approach: no fixed rule of the form "always use insertion" or "always use NEHedd" would capture the better solution in more than about 55% of instances, whereas the hybrid captures it in 100% of instances by construction.

Frequency alone, however, does not measure what is at stake, since it counts a narrow loss and a heavy one alike. The relevant magnitude is the penalty separating the two candidates on the instances where they disagree — what an analyst forfeits by having committed to the one that turns out to be worse. Averaged per instance over the whole benchmark, that spread is 7.5% of the worse candidate's penalty, and 5.8% on the large high-traffic instances specifically, and the hybrid recovers all of it by construction. Note that this relative spread is not simply increasing in instance size: large instances carry much larger absolute penalties, so a given absolute discrepancy between the two candidates represents a smaller fraction of the total there than it does on the small instances of the same series. Two consequences follow. First, the case for the hybrid does not rest on the near-50/50 split alone: even a rule that picked correctly 55% of the time would still forfeit a substantial share of this spread. Second, the appropriate benchmark for the hybrid is this spread rather than its margin over each individual component, because the latter necessarily collapses towards zero against whichever component dominates a given regime — as it does against NEHedd on the large high-traffic instances of Section 6.2.3, where the hybrid and NEHedd almost coincide.

6.5 Overall Performance Summary

Tables 8–10 report, for every configuration and traffic level, the mean penalty of each method, the relative percentage deviation (RPD) of insertion and NEHedd from the hybrid — computed per instance and then averaged, following the usual convention in the scheduling literature — the mean execution time, and the p-value of a paired Wilcoxon signed-rank test of the insertion–NEHedd difference. They give in numerical form the information that Figures 10-36 present graphically, so that every quantitative claim made in Sections 6.2 and 6.3 can be read off a table rather than estimated from a bar chart. Table 11 then summarizes comparative performance across traffic levels.

Table 8: Aggregated results, rectangular configurations: mean penalty (Insertion, NEHedd, Hybrid), RPD of Insertion and NEHedd relative to the Hybrid, mean execution time (s), and Wilcoxon p-value for the Insertion–NEHedd penalty difference (paired, per instance).

(N, M)	Traffic	$\bar{P}_{\text{Ins}}$	$\bar{P}_{\text{NEH}}$	$\bar{P}_{\text{Hyb}}$	RPD_{Ins}	RPD_{NEH}	$\bar{t}_{\text{Ins}}$	$\bar{t}_{\text{NEH}}$	$\bar{t}_{\text{Hyb}}$	p
20,30	Low	1595	1852	1592	0.2%	17.0%	0.89	0.095	0.99	¡0.001
20,30	Medium	2031	2316	2026	0.2%	14.7%	0.90	0.095	0.99	¡0.001
20,30	High	2254	2522	2250	0.2%	12.4%	0.84	0.095	0.94	¡0.001
30,45	Low	3553	3919	3543	0.3%	10.8%	3.77	0.466	4.24	¡0.001
30,45	Medium	4527	4995	4519	0.2%	10.7%	3.92	0.463	4.39	¡0.001
30,45	High	5049	5426	5036	0.2%	8.0%	3.82	0.462	4.28	¡0.001
40,60	Low	6098	6621	6089	0.1%	8.9%	10.84	1.433	12.28	¡0.001
40,60	Medium	7886	8526	7870	0.2%	8.4%	11.41	1.432	12.85	¡0.001
40,60	High	8910	9359	8874	0.4%	5.5%	10.19	1.434	11.62	¡0.001
50,75	Low	9676	10351	9658	0.2%	7.2%	23.72	3.457	27.18	¡0.001
50,75	Medium	12879	13439	12805	0.6%	5.0%	25.23	3.459	28.69	¡0.001
50,75	High	14735	14819	14503	1.6%	2.2%	22.82	3.468	26.29	0.630
60,90	Low	13760	14390	13718	0.3%	5.0%	45.68	7.137	52.82	¡0.001
60,90	Medium	18850	19020	18610	1.3%	2.3%	45.17	7.145	52.32	0.037
60,90	High	21495	21101	20890	2.9%	1.0%	39.86	7.144	47.01	¡0.001
70,105	Low	19002	19565	18883	0.6%	3.7%	73.32	13.264	86.59	¡0.001
70,105	Medium	26425	26129	25794	2.5%	1.3%	73.46	13.210	86.67	0.044
70,105	High	30227	28996	28914	4.6%	0.3%	66.29	13.257	79.55	¡0.001

Table 9: Aggregated results, square configurations: mean penalty (Insertion, NEHedd, Hybrid), RPD of Insertion and NEHedd relative to the Hybrid, mean execution time (s), and Wilcoxon p-value for the Insertion–NEHedd penalty difference (paired, per instance).

(N, M)	Traffic	$\bar{P}_{\text{Ins}}$	$\bar{P}_{\text{NEH}}$	$\bar{P}_{\text{Hyb}}$	RPD_{Ins}	RPD_{NEH}	$\bar{t}_{\text{Ins}}$	$\bar{t}_{\text{NEH}}$	$\bar{t}_{\text{Hyb}}$	p
25,25	Low	1955	2238	1951	0.2%	15.8%	1.46	0.154	1.61	¡0.001
25,25	Medium	2617	2956	2613	0.2%	13.6%	1.71	0.154	1.86	¡0.001
25,25	High	3002	3276	2993	0.3%	9.9%	1.55	0.154	1.70	¡0.001
35,35	Low	3744	4216	3734	0.3%	13.0%	4.78	0.573	5.36	¡0.001
35,35	Medium	5038	5589	5032	0.1%	11.1%	5.19	0.574	5.76	¡0.001
35,35	High	5812	6187	5797	0.3%	6.9%	4.94	0.573	5.52	¡0.001
45,45	Low	5959	6472	5943	0.3%	9.0%	10.43	1.537	11.96	¡0.001
45,45	Medium	8218	8766	8200	0.2%	7.0%	12.68	1.540	14.22	¡0.001
45,45	High	9611	9970	9545	0.7%	4.5%	11.11	1.538	12.65	¡0.001
55,55	Low	8964	9525	8929	0.4%	6.8%	22.92	3.395	26.32	¡0.001
55,55	Medium	12705	13296	12654	0.4%	5.2%	26.23	3.400	29.63	¡0.001
55,55	High	14964	15066	14724	1.7%	2.4%	22.89	3.403	26.30	0.405
65,65	Low	12512	13210	12478	0.3%	5.8%	37.93	6.585	44.51	¡0.001
65,65	Medium	18225	18718	18063	0.9%	3.7%	40.94	6.581	47.53	¡0.001
65,65	High	21729	21292	21099	3.1%	0.9%	37.61	6.586	44.20	¡0.001
75,75	Low	16381	16973	16309	0.4%	4.1%	61.21	11.685	72.90	¡0.001
75,75	Medium	24546	24224	23922	2.6%	1.2%	64.47	11.670	76.14	0.005
75,75	High	29185	27598	27548	6.1%	0.2%	58.07	11.684	69.75	¡0.001

Table 10: Aggregated results, elongated configurations: mean penalty (Insertion, NEHedd, Hybrid), RPD of Insertion and NEHedd relative to the Hybrid, mean execution time (s), and Wilcoxon p-value for the Insertion–NEHedd penalty difference (paired, per instance).

(N, M)	Traffic	$\bar{P}_{\text{Ins}}$	$\bar{P}_{\text{NEH}}$	$\bar{P}_{\text{Hyb}}$	RPD_{Ins}	RPD_{NEH}	$\bar{t}_{\text{Ins}}$	$\bar{t}_{\text{NEH}}$	$\bar{t}_{\text{Hyb}}$	p
40,10	Low	1787	2147	1783	0.2%	21.2%	2.58	0.264	2.84	¡0.001
40,10	Medium	3371	4195	3371	0.0%	25.6%	6.52	0.265	6.79	¡0.001
40,10	High	4458	5152	4457	0.0%	16.1%	7.31	0.264	7.58	¡0.001
60,15	Low	4077	4646	4073	0.1%	14.1%	9.25	1.272	10.53	¡0.001
60,15	Medium	7539	8585	7536	0.0%	14.1%	20.33	1.273	21.60	¡0.001
60,15	High	10228	10828	10190	0.4%	6.4%	20.37	1.275	21.64	¡0.001
80,20	Low	7137	7825	7129	0.1%	10.0%	23.19	3.930	27.12	¡0.001
80,20	Medium	13256	14173	13224	0.2%	7.3%	44.70	3.930	48.63	¡0.001
80,20	High	18650	18393	18158	2.8%	1.3%	45.50	3.933	49.43	0.004
100,25	Low	10704	11489	10690	0.1%	7.6%	45.00	9.438	54.44	¡0.001
100,25	Medium	21022	21481	20756	1.3%	3.6%	78.10	9.441	87.55	¡0.001
100,25	High	30233	28235	28209	7.2%	0.1%	86.47	9.433	95.90	¡0.001
120,30	Low	15058	15706	15001	0.4%	4.8%	73.91	19.463	93.37	¡0.001
120,30	Medium	31366	30562	30401	3.2%	0.5%	133.77	19.488	153.25	¡0.001
120,30	High	45225	40497	40497	11.7%	0.0%	133.03	19.487	152.52	¡0.001

Table 11: Comparative summary of algorithms by traffic level.

Traffic	**Best Penalty**	**Fastest**	**Stability**
Low	Hybrid > Insertion > NEHedd	NEHedd	Insertion more stable
Medium	Hybrid > Insertion ≈ NEHedd	NEHedd	NEHedd more variable
High	Hybrid > NEHedd > Insertion	NEHedd	NEHedd more variable

6.6 Temporal Complexity of the Insertion Heuristic

We fitted a power-law model to execution time data:

$$\text{time} = A \cdot (N \cdot M)^b, \tag{12}$$

estimating A and b by nonlinear regression for each configuration type and traffic level. Tables 12–14 report the fitted parameters.

Table 12: Fitted complexity parameters for rectangular configurations.

Traffic	A	b
Low	2.7×10^{-5}	1.6652
Medium	3.9×10^{-5}	1.6243
High	3.2×10^{-5}	1.6344

Table 13: Fitted complexity parameters for square configurations.

Traffic	A	b
Low	3.8×10^{-5}	1.6566
Medium	1.0×10^{-4}	1.5497
High	8.1×10^{-5}	1.5627

Table 14: Fitted complexity parameters for elongated configurations.

Traffic	A	b
Low	5.72×10^{-4}	1.4381
Medium	1.643×10^{-3}	1.3801
High	3.037×10^{-3}	1.3063

The exponents $b \in [1.31, 1.67]$ confirm polynomial complexity: approximately $\mathcal{O}((NM)^{1.4})$ for elongated configurations and $\mathcal{O}((NM)^{1.6})$ for rectangular and square configurations. These values bracket the theoretical exponent of 1.5 derived in Section 4.1 for the full-evaluation bound $\mathcal{O}(PSN^2M)$, which is the expected outcome for an implementation that re-evaluates only a bounded-length suffix at each insertion: the fitted exponent should lie between the $\mathcal{O}(NM)$ incremental bound ($b = 1$) and the $\mathcal{O}(N^2M)$ full-evaluation bound ($b = 1.5$ for square configurations), and modestly above the latter once the constant-factor overheads of Pareto filtering and candidate sorting – both of which grow with the number of retained sequences – are absorbed into the fit. The systematically lower exponent for elongated configurations is consistent with the insertion heuristic's own construction: with $N > M$, most of the sequence-building steps insert into a partial sequence bounded by a comparatively small and roughly constant machine count M, so per-step evaluation cost grows more slowly with N than it does in the rectangular case, where M itself grows with the instance. Exponential complexity is excluded, as it would be incompatible with the observed times (e.g., 73.32 seconds for $N = 70$, $M = 105$). For moderate-size instances ($NM \leq 2400$) the execution time remains below 12 seconds, confirming practical efficiency.

6.7 Optimality Gap on Small Instances

The comparisons above are between heuristics; they do not establish how far any of them lies from the optimum. To quantify this, the MILP of Section 3 was solved exactly with CBC

on small instances, and the penalty returned by each heuristic compared against the certified optimum. Instances were drawn from the generator of Section 5.2 with $N \in \{6\}$, $M \in \{2, 3\}$ and 20 seeds per traffic level, giving 120 instances all solved to proven optimality. Of these, 42 admit a schedule with zero total penalty (every job on time), for which a relative gap is undefined; the statistics below are therefore computed on the remaining 78 instances. Table 15 reports the mean relative gap to the optimum and the proportion of instances on which each heuristic attains the optimum exactly.

Table 15: Mean relative gap to the certified optimum, and proportion of instances solved exactly, on 78 small instances with non-zero optimal penalty. "NEH-LPT" denotes the same insertion procedure initialized by descending total processing time instead of EDD, reported for reference.

	Insertion		**NEHedd**		**NEH-LPT**	
Traffic	Gap	Exact	Gap	Exact	Gap	Exact
Low ($n = 17$)	2.45%	94%	2.45%	94%	28.00%	65%
Medium ($n = 26$)	8.84%	81%	18.68%	50%	48.66%	38%
High ($n = 35$)	7.69%	66%	15.37%	57%	36.36%	37%
All ($n = 78$)	**6.93%**	**77%**	13.66%	63%	38.64%	44%

Three observations follow. First, the insertion heuristic is quasi-optimal in a measurable sense at this scale: it attains the exact optimum on 77% of instances and its mean gap is under 7%, which substantiates the characterization used in the title. Second, the gap widens as traffic intensifies – from 2.45% under low traffic to roughly 8% under medium and high traffic – confirming that the difficulty of the instance, and not merely its size, drives the loss of quality; under low traffic the generous slack means most jobs can be scheduled on time by any reasonable ordering, and the heuristic is then near-exact. Third, insertion has a smaller gap than NEHedd at these sizes (6.93% against 13.66%; paired Wilcoxon signed-rank test, $p = 0.009$), which is consistent with the small-instance advantage reported in Sections 6.2.1–6.2.3 and provides an independent, optimum-anchored confirmation of it.

The selection heuristic is omitted from Table 15 because at $N = 6$ with $W_{\text{init}} = W = 6$ its bootstrapping phase enumerates all 6! permutations, making it exact by construction; its gap is therefore identically zero and carries no information at this scale. Two further caveats apply. The instances used here are far smaller than those of Section 6.2, so these gaps should not be extrapolated to $N = 120$: the exact MILP becomes intractable well before that size, which is precisely why heuristics are needed. And the sample size is modest. Extending this analysis to $N \leq 12$ with more seeds, which is within reach of a commercial solver, would tighten the estimates considerably.

6.8 Discussions

Three aspects of these results deserve comment beyond the per-configuration observations above.

Which method to deploy. The results do not identify a single best heuristic, and this is the practically useful finding rather than an inconclusive one. Where computation time is the binding constraint – online or near-real-time rescheduling, or very large instances – NEHedd is the appropriate choice: it is the fastest of the three methods at every instance size and traffic level tested, and its quality disadvantage relative to the hybrid is modest. Where solution quality is paramount and scheduling is performed offline — weekly or shift-level production planning, where a few minutes of computation is immaterial against the cost of late deliveries — the hybrid is preferable, since it is guaranteed never to be worse than either component and empirically achieves the lowest penalty in every scenario tested. The value of that guarantee is the 7.5% spread quantified in Section 6.4: it is what an analyst would forfeit, on average,

by fixing one heuristic in advance and being wrong. Whether that is worth roughly doubling the computation is a question about the relative cost of tardiness and of computing time in the application at hand, and the figures above are what allow it to be answered rather than assumed. The insertion heuristic alone occupies the middle ground, and is the best single-method choice for small to moderate instances under low traffic.

Comparison with the metaheuristic literature. The gains reported here are smaller than those typically claimed for metaheuristic approaches to related objectives; Li et al. [12] and Silva et al. [10], for instance, report larger improvements over constructive baselines for weighted and quadratic tardiness respectively. This is expected and not a weakness of the present approach: iterated-greedy and population-based methods invest orders of magnitude more evaluations per instance, and the comparison is therefore not like for like. The relevant contrast is that the methods proposed here complete in seconds on instances with NM up to 2400 and admit an explicit polynomial complexity characterization (Sections 4.1 and 6.6), which the cited metaheuristics do not. The two families are complementary: a natural composition would use the hybrid's output to initialize a metaheuristic, as discussed in Section 7.2.

Threats to validity. Three limitations qualify the above. First, all instances are synthetic and drawn from the specific distributions of Table 3; in particular, exponential processing times and uniform slack may understate the structure present in real production data, where processing times are often clustered by product family. Second, the parameter values for S, P, W, K were selected from pilot runs rather than systematic tuning, so the reported comparison is between reasonable rather than optimally tuned configurations, and the ranking of insertion against selection in particular could shift under thorough tuning. Third, the optimality gaps of Section 6.7 are established only at $N = 6$, where the MILP remains tractable; the gap at the instance sizes of Section 6.2 is extrapolated from the trend across traffic levels rather than measured, and remains the principal quantitative uncertainty in this study.

7. CONCLUSIONS

7.1 Summary of Contributions

This paper addressed multi-machine permutation flowshop scheduling to minimize a weighted combination of fixed late penalties and proportional tardiness, subject to arrival times and due dates. The main contributions are:

1. A clear mathematical model integrating arrival times, due dates, variable fixed penalties, and tardiness coefficients, with an explicit justification (Section 3.1) that its two penalty components are unit-compatible and can be summed directly.

2. Multi-machine adaptations of two iterative heuristics (insertion and selection) from the single-machine framework of Gradwohl et al. [4], using Pareto filtering to maintain solution diversity.

3. A hybrid algorithm combining insertion and NEHedd that achieves the lowest penalty in every configuration and traffic level tested. Its contribution is best stated as the elimination of the penalty spread between the two components — 7.5% per instance on average over the full benchmark — rather than as a uniform improvement over both, since by construction it tracks whichever component dominates a given regime. Its necessity is supported by the near-50/50 split in Table 7 together with the magnitude of that spread.

4. A comprehensive empirical study over 5,100 instances covering three configuration types and three traffic levels, with a polynomial time complexity characterization of the insertion heuristic.

Key findings are: the insertion heuristic offers an excellent quality-speed trade-off for low-to-medium traffic and smaller instances; NEHedd is faster and preferable for large instances under high traffic; and the hybrid is ideal when solution quality is paramount and the extra computation of running both components is acceptable. The complementarity of insertion and NEHedd – each being selected in approximately half of all instances – justifies the hybrid design.

7.2 Limitations and Future Work

Several limitations suggest directions for future research:

1. **Parameter optimization.** The parameter ranges tested for S, P, W, K (Section 5.4) were chosen from preliminary pilot runs rather than a systematic search; a more exhaustive exploration, e.g. via grid search, Bayesian optimization, or automated tuning frameworks, could further improve the quality-time trade-off. The results of Section 6.1.4 suggest where such a search would and would not pay: S, P and K are already saturated over the ranges tested, so little is to be gained there, whereas W is the one parameter with a large effect on cost and it should be explored downwards — towards $W < 6$, and towards non-uniform window sizes — rather than upwards, since $W = 7$ already costs roughly five times $W = 6$ for no measurable quality gain.

2. **Hybrid parallelization.** As discussed in Section 4.3, the two components of the hybrid algorithm are independent and could be executed concurrently on separate cores or threads. Because the hybrid's running time is currently the sum of its two components (Section 6.6), parallel execution would reduce wall-clock time toward the maximum of the two, making the hybrid viable for near-real-time use without sacrificing its penalty advantage.

3. **Integration of metaheuristics.** Combining the hybrid with genetic algorithms, tabu search, or iterated local search (Section 2.4) would allow broader exploration of the solution space beyond what a single constructive pass can reach, potentially closing part of the residual gap to optimal or near-optimal solutions on the largest instances, at an additional but likely still polynomial computational cost.

4. **Validation on real data.** All experiments in this study use synthetic instances generated from the distributions in Table 3. Testing on industrial data (e.g., logs from actual production lines, where processing times, arrival patterns, and due-date policies may deviate substantially from the exponential/uniform assumptions used here) would strengthen the practical applicability claims made in Section 6.

5. **Extension to other models.** Applying the heuristics to hybrid flowshop (where some stages have parallel machines) or job shop variants (where job routings differ) would broaden their scope beyond the strict permutation-flowshop assumption used throughout this paper.

ACKNOWLEDGMENT

The authors gratefully acknowledge Texas A&M University–Central Texas for providing travel funding essential to the completion of this work, and the 3HD Foundation of Porto Novo for logistical support.

CODE AND DATA AVAILABILITY

The implementation of the three heuristics, the instance generator, and the scripts producing all figures and tables in this paper are available at `https://github.com/agbadogbesamuel-hub/`

`Manuscript_Final_Version.git`. All instances used in the experiments are fully reproducible from the seeds 0–99 and the generation parameters of Table 3.

REFERENCES

[1] M. Nawaz, E. E. Enscore, and I. Ham, "A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem," *Omega*, vol. 11, no. 1, pp. 91–95, 1983.

[2] E. Vallada, R. Ruiz, and G. Minella, "Minimising total tardiness in the m-machine flow-shop problem: A review and evaluation of heuristics and metaheuristics," *Computers & Operations Research*, vol. 35, no. 4, pp. 1350–1373, 2008.

[3] V. Fernandez-Viagas and J. M. Framinan, "NEH-based heuristics for the permutation flowshop scheduling problem to minimise total tardiness," *Computers & Operations Research*, vol. 60, pp. 27–36, 2015.

[4] M. Gradwohl, G. Sewa, O. B. Oghojafor, R. Wilouwou, M. Adamu, and C. Thron, "Two Pareto optimum-based heuristic algorithms for minimizing tardiness and late jobs in the single machine flowshop problem," *Unilag Journal of Mathematics and Applications*, vol. 5, no. 1, pp. 13–38, 2025.

[5] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117–129, 1976.

[6] Y.-D. Kim, "A new branch-and-bound algorithm for the permutation flowshop with total tardiness objective," *Computers & Operations Research*, vol. 20, no. 8, pp. 829–838, 1993.

[7] A. Oukil and A. El-Bouri, "Ranking dispatching rules in multi-objective dynamic flow shop scheduling: A multi-faceted perspective," *International Journal of Production Research*, vol. 59, no. 2, pp. 388–411, 2021.

[8] E. Taillard, "Some efficient heuristic methods for the flow shop sequencing problem," *European Journal of Operational Research*, vol. 47, no. 1, pp. 65–74, 1990.

[9] B. de Athayde Prata, L. Abreu, and V. Fernandez-Viagas, "A systematic review of permutation flow shop scheduling with due-date-related objectives," *Computers & Operations Research*, vol. 175, p. 106989, 2025.

[10] A. F. Silva, J. M. Valente, and J. E. Schaller, "Metaheuristics for the permutation flowshop problem with a weighted quadratic tardiness objective," *Computers & Operations Research*, vol. 140, p. 105691, 2022.

[11] A. Costa, J. M. Valente, and J. E. Schaller, "Efficient procedures for the weighted squared tardiness permutation flowshop scheduling problem," *Flexible Services and Manufacturing Journal*, vol. 32, no. 4, pp. 850–874, 2020.

[12] Q.-Y. Li, Q.-K. Pan, L. Gao, H.-Y. Sang, X.-X. Zhang, and W.-M. Li, "The constrained permutation flowshop problem: An effective two-stage iterated greedy algorithm to minimize weighted tardiness," *Swarm and Evolutionary Computation*, vol. 91, p. 101738, 2024.

[13] X. Feng, F. Zhao, G. Jiang, T. Tao, and X. Mei, "A tabu memory based iterated greedy algorithm for the distributed heterogeneous permutation flowshop scheduling problem with the total tardiness criterion," *Expert Systems with Applications*, vol. 237, p. 121790, 2023.

[14] H. Y. Fuchigami and B. de Athayde Prata, "Coronavirus optimization algorithms for minimizing earliness, tardiness, and anticipation of due dates in permutation flow shop scheduling," *Arabian Journal for Science and Engineering*, vol. 48, no. 8, pp. 10 839–10 857,

2023.

[15] G. Li and L. Zhang, "Discrete-event simulation integrates an improved NEH algorithm for practical flowshop scheduling problems in the satellite industry," *Applied Sciences*, vol. 14, no. 21, p. 9755, 2024.

[16] C. Sauvey and N. Sauer, "Two NEH heuristic improvements for flowshop scheduling problem with makespan criterion," *Algorithms*, vol. 13, no. 5, p. 112, 2020.

[17] M. Bouška, P. Šůcha, A. Novák, and Z. Hanzálek, "Deep learning-driven scheduling algorithm for a single machine problem minimizing the total tardiness," *European Journal of Operational Research*, vol. 308, no. 3, pp. 990–1006, 2023.