Unilag Journal of Mathematics and Applications,
Volume 5 Issue 1 (2025), Pages 13–38.
ISSN: 2805 3966. URL: http://lagjma.edu.ng

TWO PARETO OPTIMUM-BASED HEURISTIC ALGORITHMS FOR MINIMIZING TARDINESS AND LATE JOBS IN THE SINGLE MACHINE FLOWSHOP PROBLEM

MATTHEW GRADWOHL[1], GUIDIO SEWA[2], OKE BLESSING OGHOJAFOR[3], RICHARD WILOUWOU[4], MUMINU ADAMU[3], AND CHRISTOPHER THRON*1

Abstract. Flowshop problems play a prominent role in operations research, and have considerable practical significance. The single-machine flowshop problem is of particular theoretical interest. Until now the problem of minimizing late jobs or job tardiness can only be solved exactly by computationally-intensive methods such as dynamic programming or linear programming. In this paper we introduce, test, and optimize two new heuristic algorithms for mixed tardiness and late job minimization in single-machine flowshops. The two algorithms both build partial schedules iteratively. Both also retain Pareto optimal solutions at intermediate stages, to take into account both tardiness and late jobs within the partial schedule, as well as the effect of partial completion time on not-yet scheduled jobs. Both algorithms can potentially be applied to scenarios with hundreds of jobs, with execution times running from less than a second to a few minutes. Although they are slower than dispatch rule-based heuristics, the solutions obtained are far better. We also attempted a neural-network solution which performs poorly, and propose reasons why neural networks may not be a suitable approach.

1. Background and motivation

1.1. **The flowshop problem: an example of a scheduling problem.** In general, a *scheduling problem* consists of organizing the execution of a sequence of jobs over time, taking into account time constraints (deadlines, sequence constraints) and constraints related to the availability of the required resources. This problem framework can be applied to a variety of practical situations, by modifying the interpretation given to jobs, machines and the necessary resources. Scheduling problems comprise a widely studied and complex combinatorial optimization problem.

2010 *Mathematics Subject Classification.* Primary: 22E30. Secondary: 58J05.
Key words and phrases. flowshop, scheduling, optimization, lateness, heuristic, Pareto, neural network .
Submitted: April 18, 2025. Revised: June 9, 2025. Accepted: July 14, 2025.
* Correspondence.

An important example of a scheduling problem is the *flowshop problem*. In general, a *flowshop* may be described as a system in which several "jobs" require processing on several sequential "machines". In different circumstances "machines" may refer to various types of processors (factory robots, line workers, editors, computer processors, etc.) that perform specified examinations or modifications of objects (automobiles, products, documents, computer programs, etc.), which are designated as "jobs". In the flowshop problem, all jobs must pass through all the machines one by one in the same sequence. A machine can only process one job at a time. In order to be processed on machine m, the job must first pass through machines $1, \ldots, m-1$, and it is not possible for two machines to work on the same job at the same time. In practice, flowshop problems occur in a wide variety of manufacturing industries including fabrics, chemicals, electronics, automotive, ([1],[2],[3]), iron and steel [4], food processing [5], ceramic tile [6], packaging and branding, pharmaceuticals, and paper manufacture [7] among others.

The flowshop problem is an example of a *matching problem*: for each machine, each job must be matched with a position. The solution to a matching problem can be expressed mathematically as a *permutation*. For example if we have three jobs, we may index them as $1, 2, 3$. If job 2 is executed first, then job 3 and finally job 1, then this job ordering may be described using the permutation $(2, 3, 1)$.

1.2. **Common variants of the single-machine flowshop problem.**

In different practical situations the flowshop managers may have different objectives. In some cases, the objective may be to minimize *makespan*, defined as the total time required to complete all jobs. Alternatively, the manager may be primarily interested in meeting the customers' requested deadlines.

There is an extensive literature on solution methods for variants of the flowshop problems, for single or multiple machines. For the makespan minimization single-machine flowshop problem, an efficient exact algorithm has been found. [8] A comprehensive review and evaluation of several heuristics and metaheuristics for the flowshop problem with total tardiness minimization may be found in [9]. Benchmarks for the permutation flowshop problem are presented in [10] and [11].

In this paper, we consider situations where all jobs have due times, and each tardy job has a fixed penalty for being tardy, plus an additional penalty that is proportional to the amount of time that it is tardy. There is no bonus for jobs that are finished early. For example if the fixed penalty is 10 and the tardiness penalty rate is 5, then a job that is 3 days tardy has cost $10 + 3 \cdot 5 = 25$.

2. Methods

2.1. **Mathematical specification of flowshop model.**

2.1.1. *Constants and variables.* Our flowshop model can be expressed mathematically in terms of the following constants and variables.

Constants: All constants listed here are positive real numbers.

- $a_1, \ldots a_N$: Arrival times for jobs at the flowshop. Jobs are listed in order of arrival, so the n'th job to arrive has index n;
- $d_1, \ldots d_N$: Due times, where d_n is the due time for the job that has arrival time a_n
- $x_1, \ldots x_N$: Processing times, where x_n is the processing time of the job with index n.
- p: fixed penalty (cost) if a job is tardy;
- q: Tardiness penalty coefficient if a job is tardy. For example, if job n arrives at time $d_n + t$, then the tardiness penalty for that job is qt (this is in addition to the fixed penalty p).

Variables:

- Tardiness: $w_k \in \mathbb{R}^+$ w_k is the tardiness of the k'th job to finish. Note that this is not the same as the job with index n. The tardiness is always nonnegative (N real variables).
- Matching variables: $u_{nk} \in \{0, 1\}$, where $1 \leq n, k \leq N$. $u_{nk} = 1$ if job n is the k'th job scheduled. Otherwise, $u_{nk} = 0$ (N^2 binary variables).
- Starting times: $s_{nk} \in \mathbb{R}^+$: Starting times, where $1 \leq n, k \leq N$. If u_{nk} is 0, then s_{nk} is also 0. Otherwise, s_{nk} is the starting time of job n (note that job n is the k'th job to execute (N^2 real variables). Under this definition, $\sum_k s_{nk}$ will be the starting time of job n, and $\sum_n s_{nk}$ is the starting time of the k'th job scheduled.
- Tardy indicators: $v_k \in \{0, 1\}$; $v_k = 1$ if the k'th job to complete is tardy, and 0 otherwise (N binary variables).

2.1.2. *Objective function.* In this paper, we will consider the problem of minimizing a penalty that depends both on the number of tardy jobs and on tardiness, as described in Section 1.2. In the mathematical specification, this corresponds to the following objective function:

$$\sum_{k=1}^{N} q \cdot w_k + \sum_{k=1}^{N} p \cdot v_k \tag{2.1}$$

2.1.3. *Constraints.*

- *Matching conditions* ($2N$ equality constraints):

$$\sum_n u_{nk} = 1 \ \forall \ k; \quad \sum_k u_{nk} = 1 \ \forall \ n \tag{2.2}$$

- *Lateness lower bound for kth job scheduled* (N inequality constraints):

$$-w_k + \sum_n \left(s_{nk} + u_{nk} \left(x_n - d_n\right)\right) \ \leq 0, \qquad 1 \leq k \leq N. \tag{2.3}$$

- *Positivity of tardiness for job in position k* (N inequality constraints):

$$-w_k \leq 0. \tag{2.4}$$

This constraint and the previous constraint ensure the correct value of tardiness.

- *Tardy binary indicators* (N inequality constraints):

$$w_k - C_{big} \cdot v_k \leq 0 \ \forall k, \tag{2.5}$$

  where C_{big} is a very large number. This constraint guarantees that $v_k = 1$ whenever $w_k > 0$, and otherwise $v_k = 0$.
- *Starting time follows arrival* (N constraints):

$$-\sum_k s_{nk} \ \leq -a_n. \tag{2.6}$$

- *Starting time constraint for consecutive jobs* ($N-1$ constraints):

$$\sum_n (s_{nk} - s_{n(k+1)}) + \sum_n u_{nk} x_n \leq 0, \qquad k < N. \tag{2.7}$$

  (A job cannot start until the previous job is finished.)
- *Nonnegative starting times*(N^2 constraints):

$$-s_{nk} \leq 0. \tag{2.8}$$

- *Starting times agree with matching variables* (N^2 constraints):

$$s_{nk} - C_{big} \cdot u_{nk} \leq 0, \tag{2.9}$$

  where C_{big} is a very large number. (This guarantees that s_{nk} is only positive when $u_{nk} = 1$.)

  Note that constraint (4) is actually implied by the requirement that $w_k \in \mathbb{R}^+$–we have included it in the constraint list for clarity.

2.2. **Algorithms for solution and approximate solution.** The mathematical problem described in Section 2.1 can be solved exactly using mixed-integer linear programming (MILP), since both the objective function and all of the constraints described in Section 2.1 are linear. However, the solution is computationally prohibitive except for very small systems. For larger systems, we must rely on approximate algorithms, which are typically either heuristic or stochastic. A newer alternative to these conventional methods is neural network.

2.2.1. *Existing heuristic scheduling algorithms.* The simplest and fastest classic heuristics use dispatch rules to determine the job execution order. According to [12], some dispatch rules that have been utilized include Earliest Due Time (EDD) (i.e., the job with the soonest due time runs first), Shortest Processing Time (SPT) and Longest Processing Time (LPT) (i.e. the fastest to process or slowest to process is run first, respectively). [13] identifies an additional dispatch heuristic Critical Ratio, which takes the amount of time left to process each job before its due date and divides by the processing to give each job a priority index, with higher priority given to the job with the lowest priority index.

Iterative heuristics add some number of jobs at a time and evaluate possible orders before moving on, with the order to be tried determined based on initial criteria, possibly using one of the above algorithms. Existing methods include the Palmer heuristic which calculates an index for each job and uses that to determine the initial order [14], and the NEH heuristic, which prioritizes jobs with longer executions over possibly multiple machines [15]. Until now it appears

that the Palmer heuristic has not been adapted to tardiness minimization. On the other hand, there is a modification of NEH (called "NEHedd") that can be applied to tardiness minimization problems [16]. Like NEH, NEHedd proceeds by iteratively constructing longer and longer job sequences, such that the final constructed sequence is the estimated job order. During the iterative process prescribed by NEHedd, it is often the case that more than one partial sequence meets the critierion for selection, i.e. there are "ties" between equally qualified partial sequences. Various mechanisms can be used to resolve these "ties" [17]. However, previous tie-breaking criteria do not make use of Pareto optimality to take into account the multi-criteria nature of the intermediate problems.

2.2.2. *Neural network solution for single-machine problem.*

Background

Neural networks (NN) have the potential of obtaining near-optimal solutions very quickly. The training process for a NN may be time-consuming, but once trained the NN can execute extremely rapidly, even for sophisticated, deep neural networks with millions of parameters.

We construct a NN to solve the single-machine flowshop problem as follows.

Neural network structure

The NN used is a shallow network consisting of 3 layers: an input layer, a hidden layer, and an output layer. Figure 1 displays the output of tensorflow's `summary` command showing the structure of the neural network, indicating the layers, connections, and input and output sizes. A more complete description of the network is as follows.

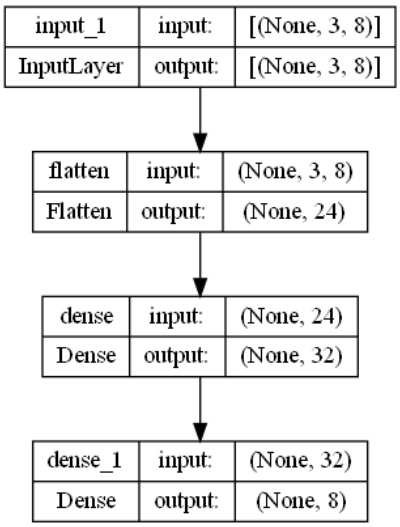

FIGURE 1. Neural network layer structure for a single-machine system with 8 jobs. Numbers in the right-hand boxes indicate the sizes of input and output arrays for each layer.

- *Input layer:* The inputs are arranged as a $3 \times N$ matrix. The three rows of the matrix contain availability times, processing times, and due times respectively, with column j representing the job with index j.
- *Hidden layer:* The hidden layer is fully connected. The number of nodes and the number of hidden layers were selected based on performance for $N = 8$ jobs.
- *Output layer:* This layer is fully-connected with N nodes, and a sigmoid activation function that produces values between 0 and 1. The estimated schedule is obtained by ordering the outputs. For example, if the NN produces the output $[0.3, 0.4, 0.2]$ then the job order should be the third job (lowest value) followed by the first job and lastly the second job (highest value). In the python code, this ordering is recovered using the command `argsort`.

2.2.3. *Pareto-based heuristics for single-machine problem.* In this paper we propose two novel heuristic methods, which we designate as "iterative insertion" and "iterative selection".

The iterative insertion algorithm is similar to the NEH algorithm that is used for makespan minimization. Modifications must be made to accommodate special difficulties posed by the tardiness-tardy time minimization criteria. In particular, since multiple criteria are used to determine "good" solutions at intermediate stages, we use Pareto optimality as a selection criterion to determine multiple candidate solutions.

On the other hand, we did not see any algorithm in the literature that is comparable to iterative selection. Iterative selection also relies on Pareto optimality to obtain a set of candidate solutions at each intermediate stage.

In the following subsections, we provide detailed descriptions of the iterative insertion and iterative selection algorithms.

Iterative insertion

Iterative insertion proceeds as follows. First, a preliminary ordered list of jobs is created, possibly using one of the simple heuristics mentioned in Section 2.2.1. During the algorithm, jobs are selected from this preliminary list one by one. The algorithm creates a series of lists of ordered lists as follows.

- The first list of ordered lists is a list with a single element containing the first preliminary job.
- The second list of ordered lists is formed as follows. Using the single list in the first list of lists (which at this stage contains a single element), insert the next preliminary job at each possible point in the list (at this point, there are two possible insertion points), to create two different lists of two elements. For each of these 2-element lists, evaluate the tardiness penalty and the finish time associated with these job orderings, and retain all of the Pareto-optimal orderings.
- Each subsequent stage of the algorithm resembles the second stage. For stage m, the m'th element of the preliminary list is selected, and inserted at each possible point in all lists in the list of lists that was created at

the $m-1$'th stage. For each of the resulting lists, evaluate the tardiness penalty and the finish time associated with the given job ordering, and retain all Pareto-optimal orderings.

- At the last stage, retain only the lists with minimal tardiness penalty.

We may give a specific example of the algorithm as follows. Suppose the preliminary order of jobs is $[4,3,2,1]$. At the first stage, the list of lists is $[[4]]$. At the second stage, the lists $[4,3]$ and $[3,4]$ are evaluated for tardiness and finish time. Suppose that both tardinessses are 0, and the finish times are equal. Then both ordered lists are retained: so at the end of the second stage, the list of ordered lists is $[[4,3],[3,4]]$. At the third stage, job 2 is inserted at each point within each list of ordered lists from the second stage. This produces a list of 6 ordered lists: $[[2,4,3],[4,2,3],[4,3,2],[2,3,4],[3,2,4],[3,4,2]]$. Of these, select those job orderings which are Pareto optimal with respect to tardiness and finish time. We may suppose for example that this reduces the list to 3 ordered lists: $[[2,4,3],[2,3,4],[3,2,4]]$. Finally, we insert job 1 at each position in each of these lists, giving rise to 12 ordered lists. From these 12 ordered lists, we select one that has minimum tardiness, since each minimum-tardiness solution is equally optimal.

In the example above, we note that an exhaustive evaluation of all possible job orderings would require $4! = 24$ evaluations for 4 jobs, considerably more than the 12 we ended up evaluating. The savings for larger numbers of jobs depends on the number of ordered lists retained from stage to stage. If for example this number is bounded by B, then the number of evaluations at stage m is $(m+1)B$, so that the total number of evaluations grows quadratically with the number of jobs (rather than exponentially).

In the above algorithm, the starting list of ordered lists contains a single element, which consists of a single job. we may define a variant of the algorithm as follows. Instead of starting with a single-element list, take the first J jobs in the preliminary list, and use as starting list the $J!$ permutations of the first J jobs. From this list, retain the Pareto optimal orderings, and then continue the algorithm as described above. J must not be chosen too large: for example, $J = 5$ means that the starting list has 120 permutations,

When calculating the penalty and finish time for the different jobs with insertion, it is possible to reduce computation time by intelligently choosing the order of calculation. For example, if job 3 is to be inserted in the list $[2,6,4,5,1]$, then one may first calculate the penalty and finish time when the new job 3 is inserted in the last place, i.e. $[2,6,4,5,1,3]$. Next, insert the new job 3 in the second to last place, i.e. $[2,6,4,5,3,1]$. Then one may reuse the calculated finish times for $2,6,4,5$, because these jobs' starting and finishing times remain unchanged. If the calculation is done this way, the complexity is reduced by a factor of 2.

Similarly, if the variant where the starting list consists of J jobs the finish times and tardinesses for the $J!$ orderings may be calculated more efficiently by ordering the calculations properly. For example, consider the case where one wants to calculate the penalties and finish times for all permutations of $[1,2,3,4]$. We may first calculate and store the penalties and finish times for each of the

single-element ordered lists, namely [1], [2], [3], and [4]. Then for example we may use the result for [1] to calculate the finish times and penalties for the 2-element ordered lists that start with 1, namely $[1, 2]$, $[1, 3]$, $[1, 4]$. Similarly, we may use [2] to calculate the finish times and penalties for the 2-element ordered lists that start with 2, namely $[2, 1]$, $[2, 3]$, $[2, 4]$. We may do the same with [3] and [4]. Then we may use the results from $[1, 2]$ to calculate results for $[1, 2, 3]$ and $[1, 2, 4]$: and so on. This method will reduce calculation by 1/2.

Two modifications were introduced to limit the complexity of the algorithm:

- It is possible that after several iterations, the set of Pareto optimal candidates may become unmanageably large. We counteract this effect by limiting the number of Pareto optimal solutions retained. This is done by retaining a representative subset of permutations from the Pareto optimal list for each iteration. We thus introduce a tuneable parameter P correponding to the maximum number of permutations retained. In order to maintain a variety among the sequences that are retained, the Pareto optimal candidates are sorted by completion time, and every $\lfloor L/P \rfloor$'th sequence is retained, where L is the number of candidates.
- One can also limit the insertion slots to a set number S, so that new jobs are only inserted into the last S possible positions in the previously-generated job sequences. In especially large numbers of jobs generated with a similar distribution to the training data, sorting by relative due time will put jobs that can be run later closer to the end, and they will not need to be tried at the beginning, as that would just push the other jobs to be tardy. Rather than trying those jobs at the beginning of the list, time is saved by just inserting in the specified number of insertion slots from the end.

The MILP formulation requires $O(N^2)$ variables and constraints, making exact solutions impractical for $N > 10$ (e.g., $N = 50$ yields $5,100$ variables). In contrast, the insertion heuristic reduces this to $O(P \cdot S \cdot N)$. If both of these modifications are implemented in the insertion algorithm, then at each iteration the number of sequences to be calculated is $P \cdot S$, and the length of the sequence is S (since it is not necessary to recalculate the job sequence before the first insertion point). It follows that the algorithm's execution time will be linear in the number of jobs.

Iterative selection

Instead of successively building up schedules by inserting new jobs, an alternative approach is based on appending new jobs to candidate partial schedules sequentially by a trial-and-retain procedure. For example, suppose it is desired to schedule 7 jobs $[1, 2, 3, 4, 5, 6, 7]$, and suppose a preliminary ordering (determined by a simple heuristic, such as greedy) has the jobs in order $[2, 1, 5, 6, 3, 7, 4]$. Then for the first $W = 4$ jobs, (namely $[2, 1, 5, 6]$ in this case) we may first exhaustively evaluate the penalties and finish times for all $W! = 24$ possible orderings of $[2, 1, 5, 6]$. From this list, we may retain the Pareto optimal orderings (with respect to penalty and finish times), and choose the initial job in each of these

Pareto optimal orderings. This will give us a list of single-element orderings. Suppose for example, that 2 and 5 are the initial elements of Pareto optimal orderings of $[2, 1, 5, 6]$. We then evaluate finish times and penalties for all orderings of the first five jobs in the preliminary list (namely 2,1,5,6,3) which begin with either 2 or 5. This requires evaluations for $2W!$ orderings. From these, we retain the Pareto optimal orderings, and select the first 2 elements for all of these Pareto optimal orderings. With reference to our example, suppose $[2, 6, 5, 1, 3]$, $[2, 1, 5, 6, 3]$ and $[5, 2, 6, 1, 3]$ are Pareto optimal. Then we will retain the 2-element ordered lists $[2, 6]$, $[2, 1]$, and $[5, 2]$ as candidates for the first two jobs. At the next stage, we evaluate penalties and finish times for all ordered sequences involving the jobs $[2, 1, 5, 6, 3, 7]$ that begin with either $[2, 6]$, $[2, 1]$, or $[5, 2]$: this involves calculating $3W!$ penalties and finish times. As before, we retain only the first three elements of all Pareto optimal ordered sequences. Suppose for example that all Pareto optimal sequences at this stage begin with $[2, 1, 6]$. Then at the next stage, we only consider ordered sequences of length 7 that begin with $[2, 1, 6]$.

As with iterative insertion, it is possible that the number of Pareto optimal candidates will grow too large. In this case we may use the same strategy as before: namely order the Pareto optimal candidates by completion time, and choose K regularly-spaced job sequences from among the candidates. It follows that the iterative selection algorithm also has two tuneable parameters, namely the permutation window and the number of retained sequences, which we denote as W and K, respectively.

2.3. **Training, testing, and evaluation of heuristic methods.**

2.3.1. *Construction of scheduling scenarios for training and testing.* In order to test the heuristic algorithms and train the neural network described in Section 2.2.2, it is necessary to generate a set of labeled instances for scheduling. The instances are generated with randomized arrival, execution, delay, and due times, as summarized in Table 1, which follows the notation of Section 2.1. The due times $d_n, n = 1, \ldots N$ are generated as follows:

$$d_n = a_n + z_n + x_n + D'_n, \tag{2.10}$$

where the random variable D'_n is exponentially distributed with mean $\mu_{D'}$. The role of D'_n is to allow some margin for each job, so that job n can experience a net delay of D'_n or less and still be completed on time. The parameter values used are summarized in Table 1. All algorithms are implemented using Python 3.10, with the MILP solver from Scipy 1.12. The Spyder integrated development environment (version 5.5.1) was used.

2.3.2. *Training the neural network.* The exact and heuristic single-machine algorithms do not need training. Only the neural network model needs training, and in this subsection we describe the training procedure.

$12,000$ single-machine cases with 6 jobs ($N = 6$) were generated according to the distributions in Table 1. The optimal job orders for these single-machine cases were computed using the exact MILP algorithm. These optimal orders were used to create labels for each case as follows. Let σ denote the permutation

Variable name	Distribution	Parameter
Arrival times	$a_1 \sim \text{Exp}(\mu_A);\ a_n \sim a_{n-1} + \text{Exp}(\mu_A)$	$\mu_A = 5$
Execution times	$x_n \sim \text{Exp}(\mu_X)$	$\mu_X = 5$
Information delays	$z_n \sim \text{Exp}(\mu_Z)$	$\mu_Z = 5$
Delay margins	$d'_n \sim \text{Exp}(\mu_{D'})$	$\mu_{D'} = 10$
Due times	$a_n + x_n + z_n + d'_n$	–

TABLE 1. Random variables and parameters for random generation of arrival, execution, delay times, delay margin, and due times. The index n labels jobs and runs from 1 to N.

corresponding to the correct ordering for a particular case, i.e. $\sigma[j] = k$ if the j'th job scheduled is job k. Then the label for that case is a vector L of length N with entries in $[0, 1]$ given by:

$$L[j] := \frac{k}{J-1}, \tag{2.11}$$

so that the order of the entries in L matches the order of the jobs scheduled. The metric used to evaluate the NN output is mean squared difference between the label and the output.

To improve the training performance of the NN, first the starting structure is adjusted. In initial testing, we tried layers with between $4N$ and $8N$ nodes, and with linear, softmin, softmax, softplus, softsign, hyperbolic tangent, SELU, elu, ReLU, and sigmoid activation functions. These functions were selected because of their common use, although other activation functions exist. More limited testing on the number of hidden layers suggested that one hidden layer with $4N$ nodes and a linear activation has the best performance using the objective function listed in 2.1.2. Parameter optimization methods tried include adadelta, adagrad, adam, adamw, adamax, RMSProp, and Stochastic Gradient Descent (SGD). The batch size was not optimized, and was set at 1. The mean squared error (MSE) loss function was used to gauge performance. The number of epochs was adjusted based on the loss curves, which leads to an epoch value of 250 where the curves begin to plateau.

During training we plot the loss curves to assist in evaluating the number of epochs. The loss curve is shown in Figure 2, which shows the flattening of the loss curve at 250 epochs.

2.4. **Performance Comparisons.** In order to evaluate the relative performance of the different algorithms, a number of tests were performed. All test scenarios were generated using the distributions and parameters described in Table 1. All

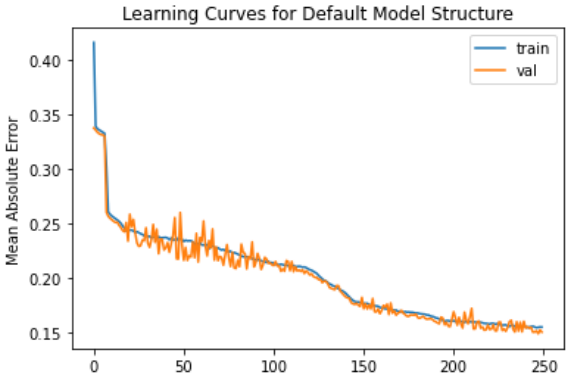

FIGURE 2. Learning loss curves for NN. The close match between training and validation curves show that overfitting has not taken place.

comparisons and training were run on a workstation with 32.0 GB of RAM and an Intel i7-1360P CPU.

First we compared numerical algorithm performance to the exact MILP solution by applying MILP, NN, and insertion to the same 2000 scenarios with 6 and 8 jobs, and comparing the objective functions obtained by the three methods. The number of jobs was limited because of the high runtimes of MILP.

Second, we compared the selection and insertion heuristic algorithms by applying both to the same 100 instances of 50 jobs. Both algorithms were run with different values for the algorithm parameters: maximum number of kept permutations and maximum number of insertion slots for the insertion algorithm; maximum number of kept permutations and selection window for selection. Several graphs comparing performance and execution times for the different algorithm variants were generated.

3. RESULTS

3.1. **Comparison of numerical algorithms with exact MILP solution for small scenarios.** Figure 3 shows the distributions of objective functions obtained using the NN, MILP, and insertion heuristic for 2000 instances of 8-job scenarios with parameters given by Table 1. For the insertion heuristic since the number of jobs was limited all permutations were kept, and all insertions were included. Clearly the insertion algorithm performance is almost identical to that of the exact MILP, while the NN is far less optimal.

Figure 4 displays the same information as Figure 3 for the 6-job case, but in the form of cumulative distributions. Here we can see more clearly the close agreement between the exact MILP solution and the heuristic solution. We conclude that at least for few enough jobs, the insertion heuristic is usually giving us an overall best solution.

3.2. **Performance comparisons between insertion and selection algorithms for larger cases.**

3.2.1. *Objective function comparisons.* Figures 5 through 11 show performance comparisons between insertion and selection heuristic algorithms with different parameter values, for 100 instances of 50 job scenarios generated using the distributions described in Table 1. The same 100 instances were run using all algorithm variants, and the best prediction from all variants was stored.

Figure 5 shows the difference between the objective values for several insertion algorithm variants with different values of P and S, compared to the best objective obtained for each instance over all insertion and selection algorithms. The figure shows mean and median differences for each variant, as well as the 5th and 95th percentile values. Maximum number of kept permutations varies from 30 to 80, while maximum number of insertions varies from 10 to 35. Altogether 30 combinations of parameter values are used. Results show little change in all performance measures when 50 or more permutations are kept ($P \geq 50$), and when insertions are limited to the last 15 or more slots ($S \geq 15$). The mean values for the different algorithm variants are consistently higher than the medians, indicated by right-skewed distributions.

Figure 6 parallels Figure 5 by showing means, medians, and 5th and 95th percentile values for objective function differences from best obtained using the selection algorithm variants with different values of K and W. The selection algorithm with $W = 7$ is far better than $W \leq 6$. Indeed, comparison with Figure 5 shows that $W = 7$ also beats all insertion variants. The number of kept permutations K has little effect on algorithm performance, regardless of the selection parameter.

Figure 7 shows the proportion of times each insertion algorithm variant matched the overall best prediction. Each variant matches the best obtained objective value for about 30 percent of instances regardless of the values of P and S. We do see a slight dip in performance for $S = 10$ compared to $S > 10$, and an even smaller dip in performane when $P < 70$. This is consistent with what was seen in Figure 5, which shows that the distribution of errors for the insertion algorithm is optimized for $S \geq 10$ and $P \geq 60$.

On the other hand, Figure 8 shows that big improvements in matching performance are obtained when the select from parameter is increased. When the select from parameter is 7, the overall best performance is matched at least 45 percent of the time across the range of kept permutation values. Selecting from 6 has similar performance to the insertion algorithms.

3.2.2. *Pairwise comparisons of algorithm performance.* Figures 9 through 11 show scatter plots that provide detailed comparisons between pairs of algorithm variants. Each point in the scatter plot represents one particular scheduling scenario. The x coordinate of the point represents the penalty achieved by one algorithm variant, and the y coordinate of the point shows the penalty achieved on the same scenario by a different algorithm variant.

Figure 9 compares insertion variants with $S = 15, 20, 30$ compared to the selection variants with $W = 5, 6, 7$ with $P = K = 50$. The 3 variations of the insertion heuristic give almost identical results (since points align closely with the 45 degree line), while the selection algorithm improves significantly as W increases.

Selection with a window of 7 performs better than the insertion variations, but insertion is better than the other two selection variations.

Figure 10 compares insertion variants with $S = 15$ compared to the selection variants with $W = 5$ with $P = 50, 60, 70$ and $K = 50, 60, 70$. The 3 variations of the insertion heuristic give almost identical results (since points align closely with the 45 degree line), while the selection algorithm improves significantly as W increases. Selection with a window of 7 performs better than the insertion variations, but insertion is better than the other two selection variations. While there is little variation over the kept permutations for either algorithm, inserting in the last 15 is consistently better than selection with a window of 5, especially with larger objective values.

Figure 11 compares insertion variants with $S = 15$ compared to the selection variants with $W = 7$ with $P = 50, 60, 70$ and $K = 50, 60, 70$. The 3 variations of the insertion heuristic give almost identical results (since points align closely with the 45 degree line), while the selection algorithm improves significantly as W increases. Selection with a window of 7 performs better than the insertion variations, but insertion is better than the other two selection variations. While there is little variation over the kept permutations for either algorithm, inserting in the last 15 is consistently worse than the selection algorithm with a window of 7, especially with smaller objective values.

3.2.3. *Runtime comparisons for insertion and selection.* Figure 12 shows runtimes for the insertion algorithm using different values for the number of kept permutations P and number of insertion slots S. The runtime is linear P for fixed S, with slope increasing with S. It appears that runtime is also linear in S for fixed P. This linearity in parameters implies that implementations with large values of P and S are feasible. However, Figure 5 indicates that large values are not necessary for good performance.

Figure 13 shows the runtime for the selection algorithm with different values for permutation window W and number of kept permutations K. The runtime is relatively constant for $K \geq 30$, whereas the runtime increases exponentially with W. Together with Figure 6, these results indicate that improvements in selection algorithm performance can be achieved by increasing W, but at relatively high computation cost.

4. Conclusions

We have shown that the insertion and selection algorithms perform nearly optimally when the number of jobs is small, and offer a substantial improvement over dispatch rules as the number of jobs increases. For both algorithms, parameters can be adjusted to affect tradeoffs between execution time and objective function performance, although with insertion the performance improvements that can be achieved by parameter adjustment are limited. In head-to-head comparisons between insertion and selection, selection and insertion algorithms have comparable accuracy when comparing variants with comparable runtime. For example, The insertion algorithm with $P = 50$ and $S = 15$ has comparable performance and runtime with the selection algorithm with $W = 6$ and $K = 30$. However, the

performance of the insertion algorithm is limited and cannot be improved by further increasing the selection window and kept permutations. On the other hand, the selection algorithm can be further improved by increasing the permutation window–but at high cost in execution time.

Although we did not do an extensive study of possible neural network architectures, the results from our simple architecture suggest that neural networks may not be well suited to this application. Preliminary investigations with much larger networks also showed very poor performance. It is reasonable that neural networks would have difficulty solving scheduling problems, due to their discontinuous nature. Neural networks training algorithms presuppose a continuous dependence of the loss function on inputs, so that gradient descent can be used to obtain incremental improvements. However, in scheduling problems the penalty is a discontinuous function of the inputs, so gradient-based parameter optimization cannot be expected to work.

Projected future work includes more extensive benchmarking. In the literature, other methods are suggested to generate scenarios [17], which may be used to test the algorithms over a wider variety of conditions.

The algorithms may also be further optimized. Further optimization can target reducing execution time and/or improving performance.

- As far as reducing execution time, not all of the speedup features for insertion and selection described in Section 2.2.3 have been implemented. There are other possibilities for speedup as well. For instance, in the selection algorithm there may be a way of selectively generating permutations, rather than generating all permutations in the permutation window.
- Performance may perhaps be improved by combining the two algorithms. It is possible, for instance, that applying selection on the schedule generated by insertion may have the effect of "fine tuning", since selection performs a local reshuffling.

Finally, the two algorithms may be generalized and applied to other problems, such as the permutation flowshop problem. This research is currently in progress.

5. DECLARATIONS

Ethics approval and consent to participate. Not applicable

Consent for publication. Not applicable

Availability of data and materials. The code used in this paper to generate data and to apply and compare scheduling algorithms is stored in GitHub at: `https://github.com/christhron/Flowshop_Iterative_Heuristics`.

Competing interests. The authors declare no competing interests.

Funding. No funding was received for conducting this study.

Authors' contributions. Conceptualization: OBO, CT,MA; **Formal analysis**: MG, GS, RW, CT; **Investigation**: MG, GS, CT; **Methodology**: MG, GS, CT; **Software**: MG, GS,CT; **Supervision**: MA, CT; **Validation**: MG, GS, CT; **Visualization**: MG, GS,CT; **Writing – original draft**: MG, OBO, RW, CT; **Writing – review & editing**: MG,CT,MA.

Acknowledgements. We wish to thank the anonymous reviewers for valuable suggestions that greatly improved the clarity of the paper.

6. Figures

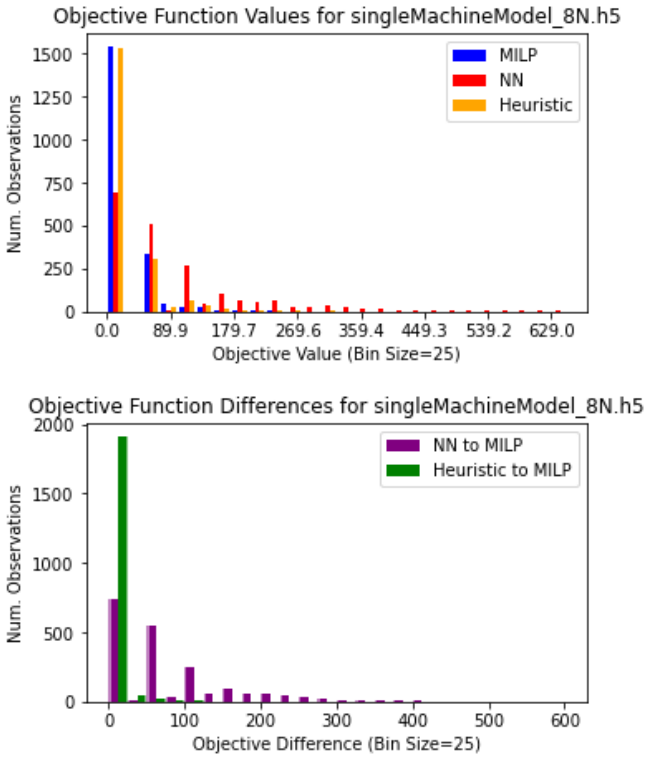

Figure 3. Distributions of objective function differences (NN minus MILP and insertion heuristic minus MILP), applied to 2000 instances of 8 jobs with distributions according to Table 1.

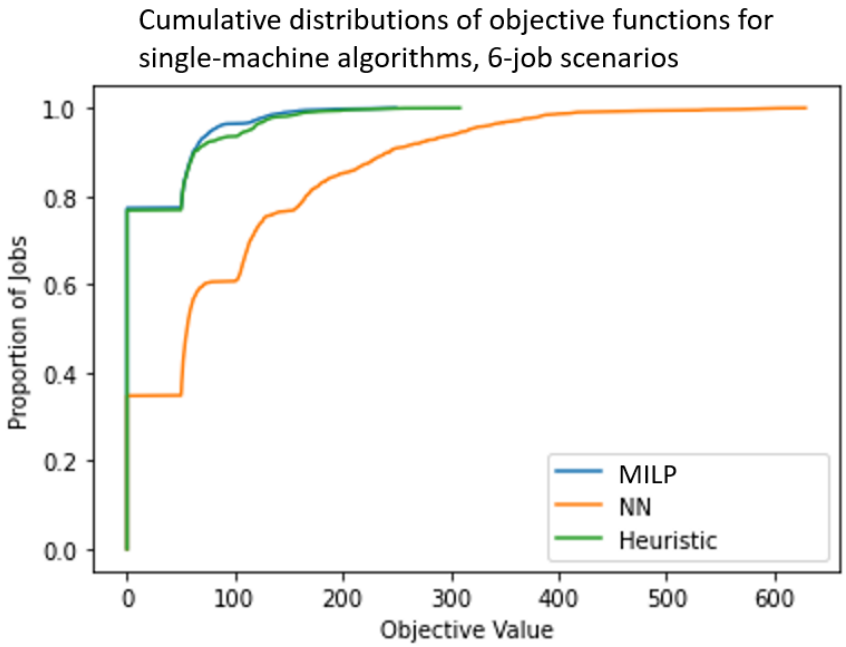

FIGURE 4. Cumulative distributions of objective functions for MILP, NN and insertion heuristic single-machine algorithms, applied to 2000 instances of 8 jobs with distributions according to Table 1.

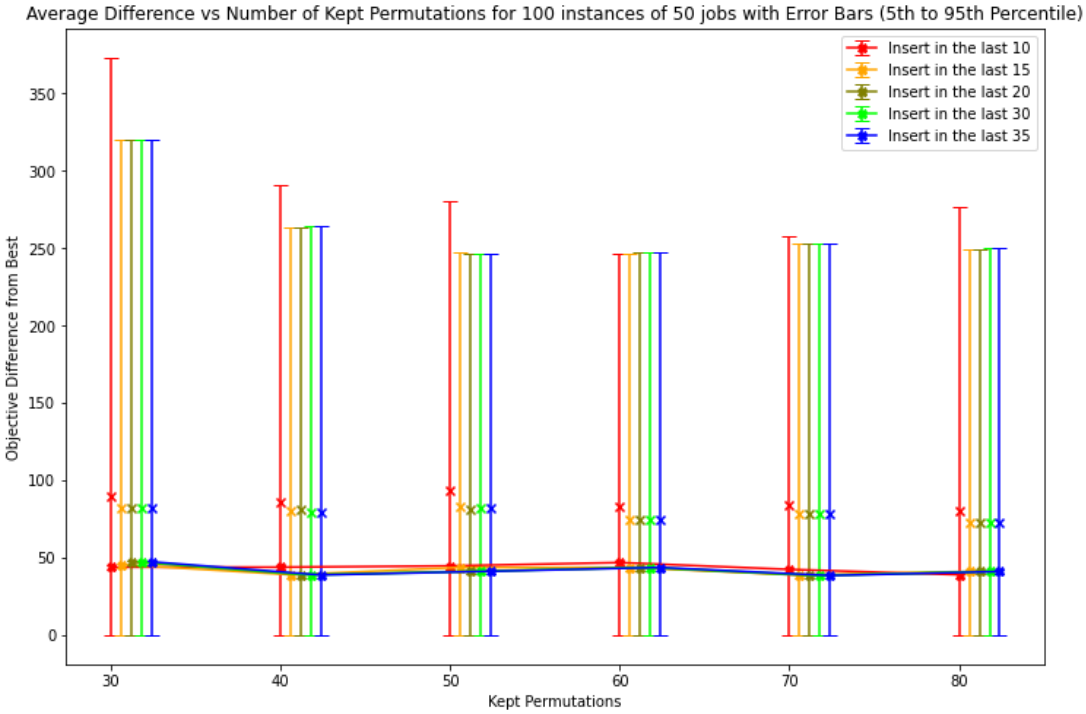

FIGURE 5. Performance comparison for 30 insertion algorithm variants using different parameter values (number of kept permutations P and number of insertion slots S), for 100 50-job scenarios. The number of insertion slots (S) is listed on the x axis, while different values of P correspond to the different series shown in the legend. The y axis shows the difference between the objective function obtained by the algorithm and the lowest overall objective function obtained from all 30 insertion algorithm variants shown in this figure and the 18 different selection algorithm variants shown in Figure 6 for each 50-job scenario. Error bars show the 5th and 95th percentiles for each algorithm variant. Medians for each algorithm variant are plotted with circles and are joined with lines, while means are plotted with $\times$'s. Variants with $S \geq 10$ and $P \geq 60$ have similar performance statistics.

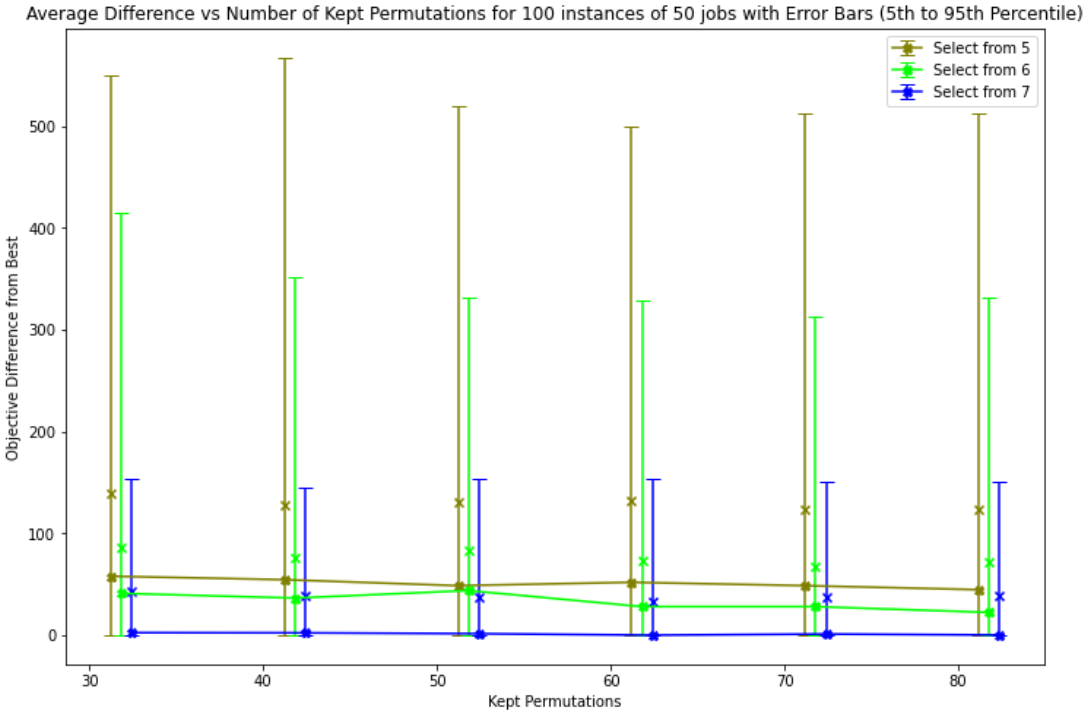

FIGURE 6. Comparison of 5th to 95th percentiles of objective minus best values for the selection algorithm run with different parameter values (number of kept permutations K and permutation window W) for 100 50-job scenarios. Best overall values were taken as the lowest objective function obtained from all insertion and selection algorithm variants for the corresponding 50-job scenario. Medians are plotted with dots and connected by lines, and means are plotted with $\times$'s. Increasing W produces notable reductions in error, while increasing K above 30 has little effect.

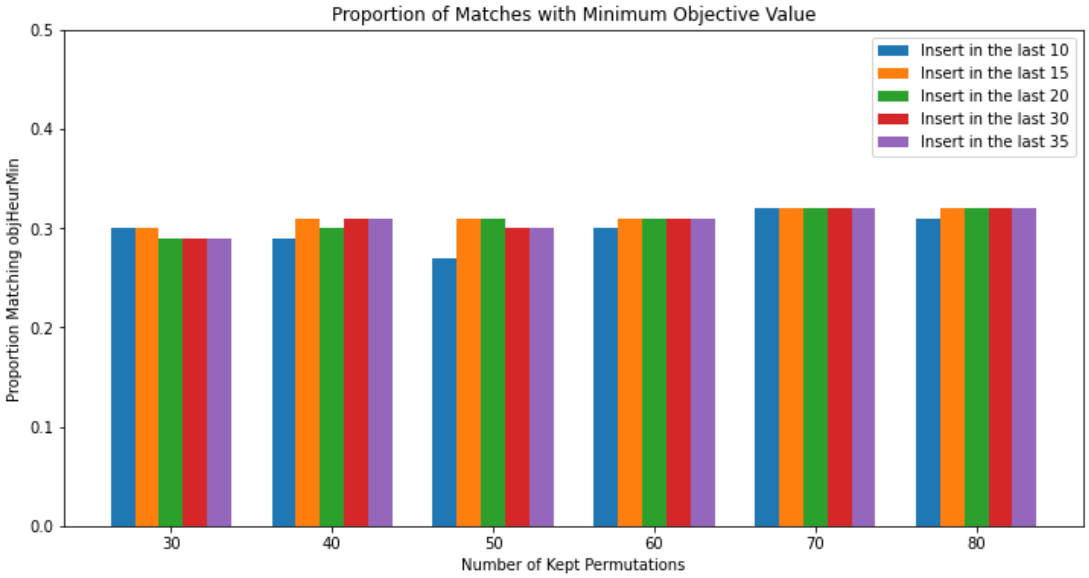

FIGURE 7. Proportion that each insertion algorithm attains the minimum penalty out of all obtained values across 50 job instances. The proportion evens out at around 0.3.

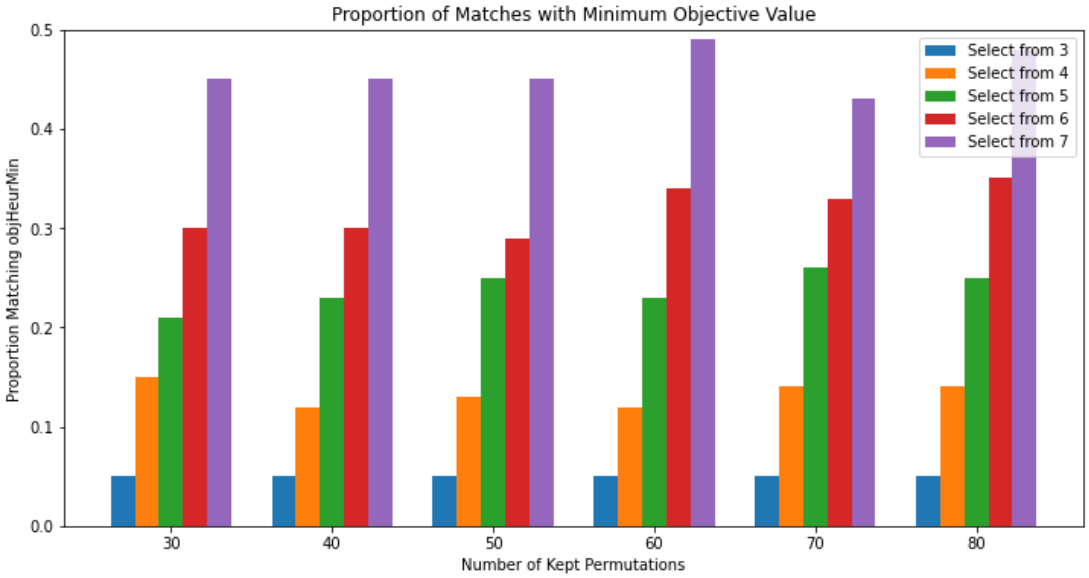

FIGURE 8. Proportion that each selection algorithm variant attains the minimum penalty out of all obtained values across 50 job instances. Selection continues to improve with larger windows, and reaches almost 0.5 with $W = 7$.

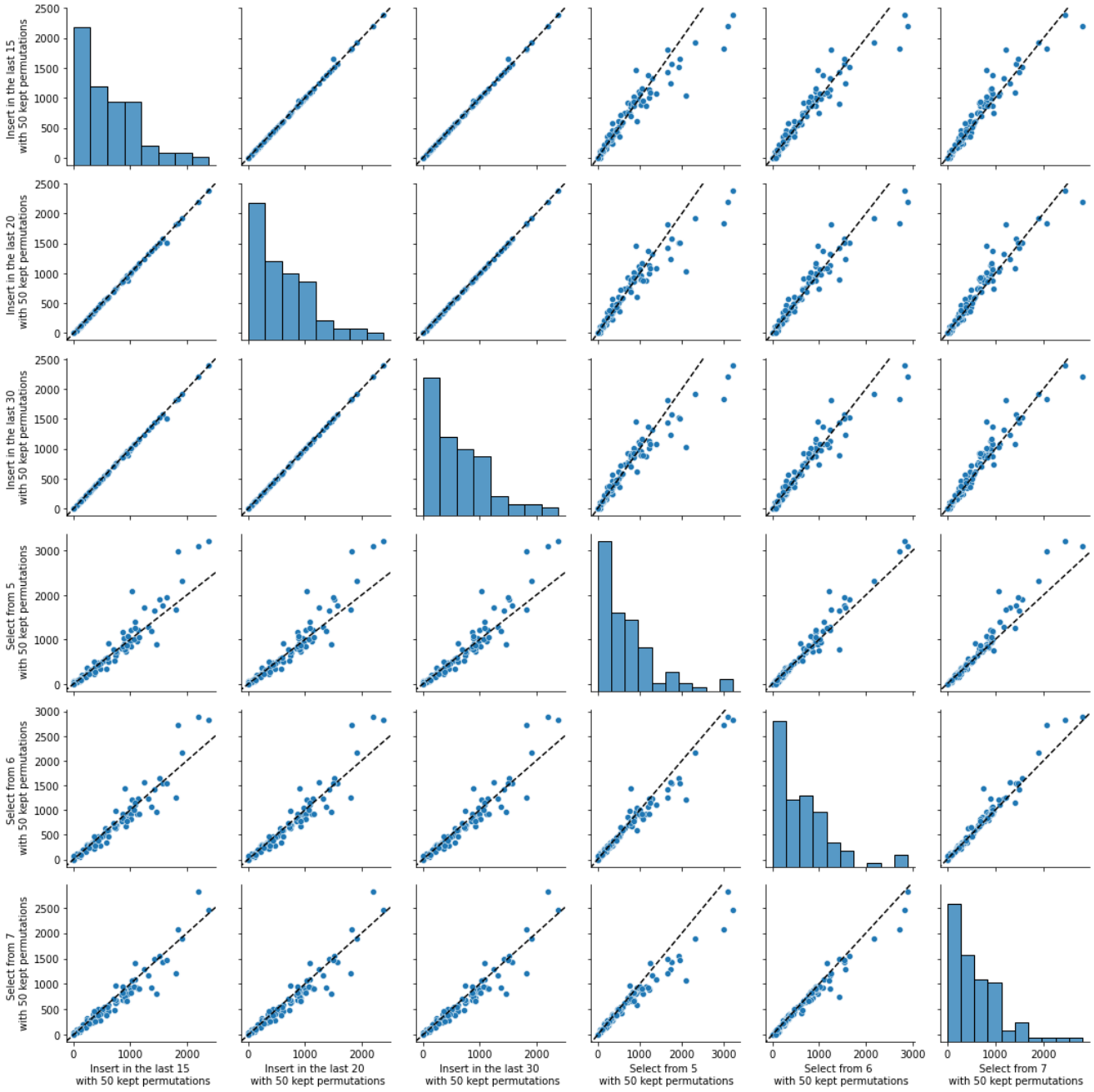

FIGURE 9. Scatter plots comparing penalties obtained from different insertion and selection algorithms. Pair plots for insertion algorithm variants with $S = 15, 20, 30$ and $P = 50$ and selection algorithm variants with $W = 5, 6, 7$ and $K = 50$.

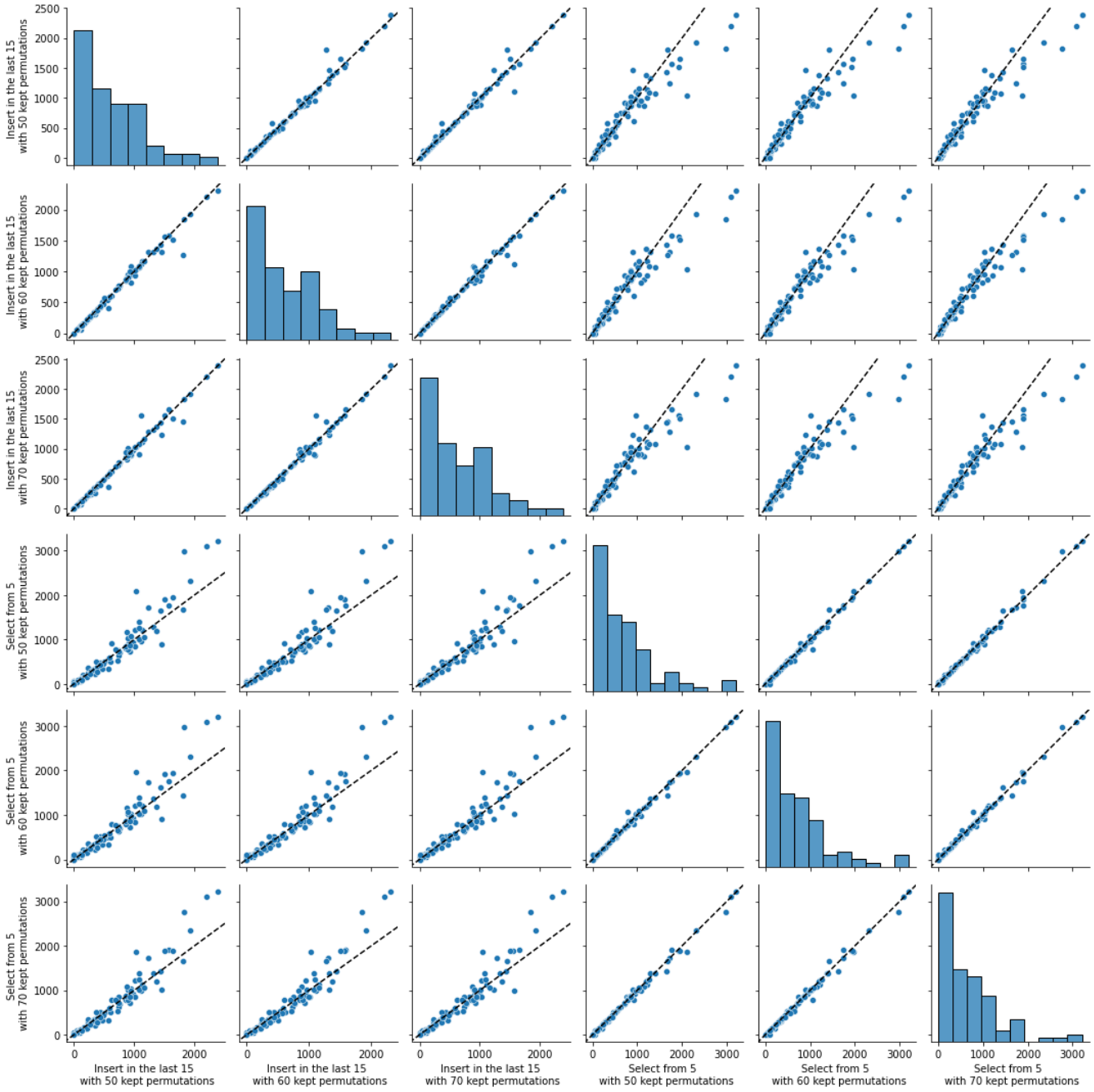

FIGURE 10. Scatter plots comparing penalties obtained from different insertion and selection algorithms. Pair plots for insertion algorithm variants with $S = 15$ and $P = 50, 60, 70$ and selection algorithm variants with $W = 5$ and $K = 50, 60, 70$.

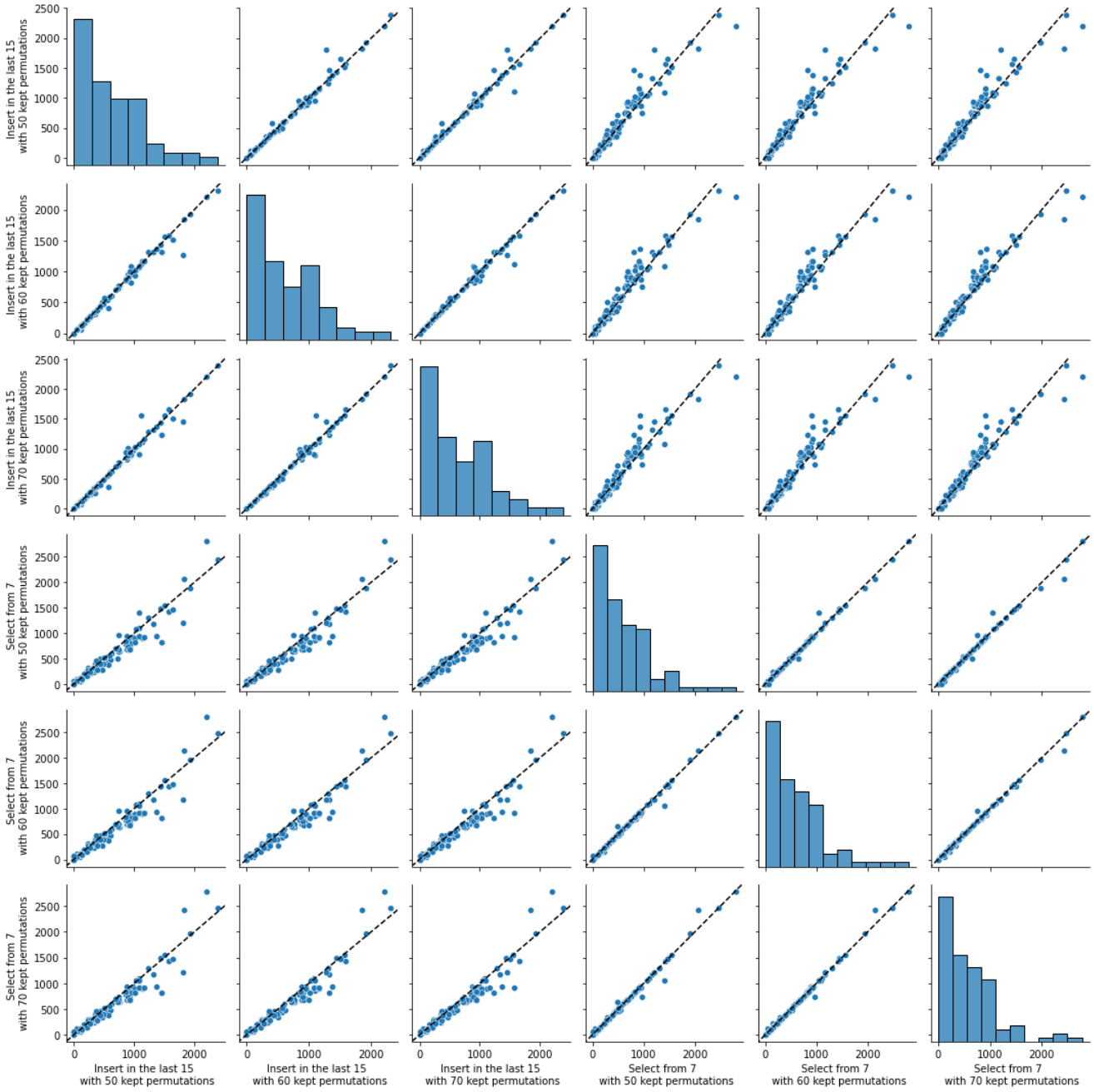

FIGURE 11. Scatter plots comparing penalties obtained from different insertion and selection algorithms. Pair plots for insertion algorithm variants with $S = 15$ and $P = 50, 60, 70$ and selection algorithm variants with $W = 7$ and $K = 50, 60, 70$.

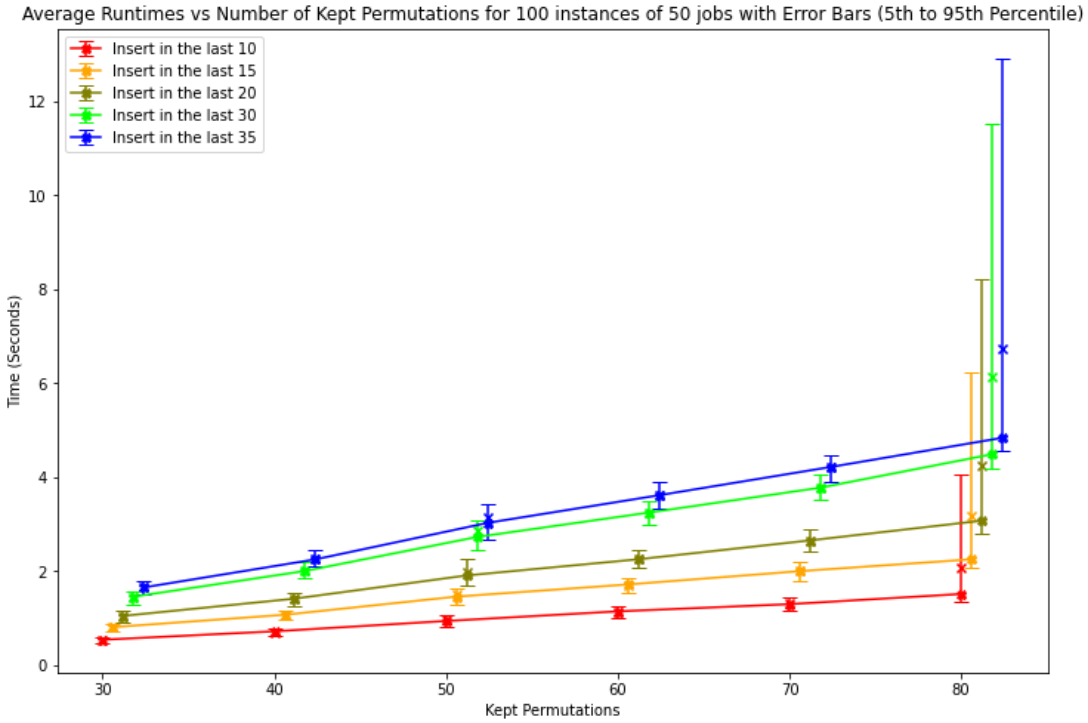

FIGURE 12. Runtime comparisons for insertion algorithm variants with kept permutations $P = 30, \ldots 80$ and $S = 10, \ldots 35$. The median runtimes for each variant are plotted with circles, means are plotted with $\times$'s. and error bars indicate the 5th and 95th percentiles. The runtime is linear in P for fixed S, with slope that increases as S increases.

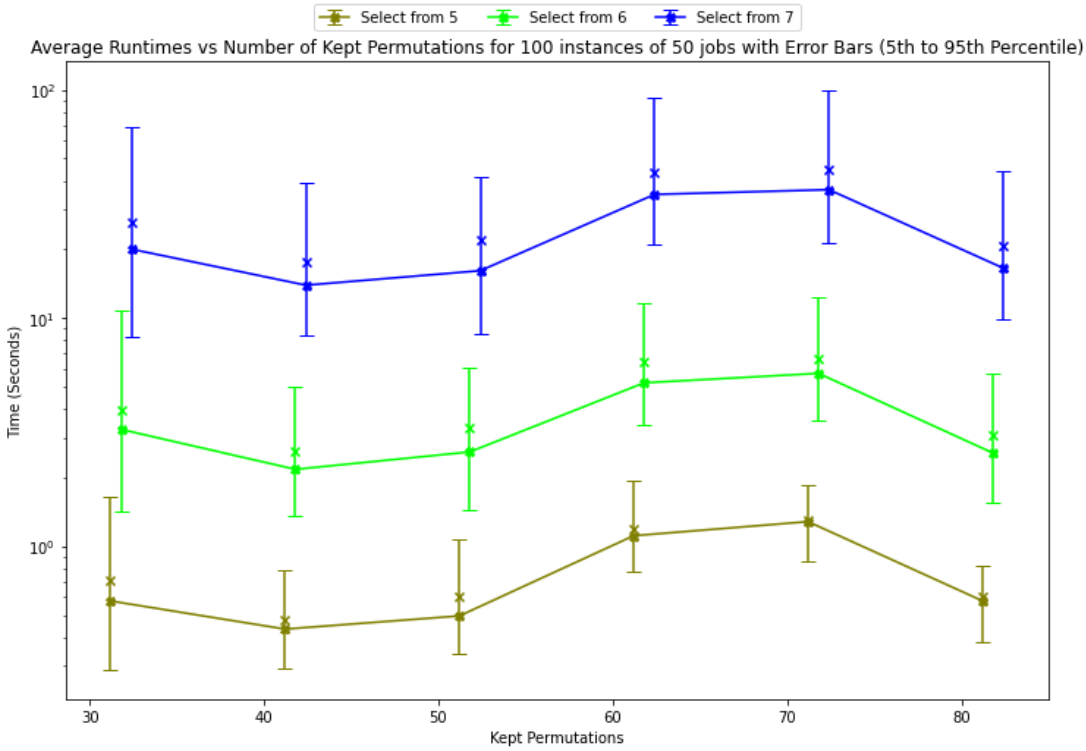

FIGURE 13. Runtime comparisons for selection algorithm variants with kept permutations $K = 30, \ldots 80$ and permutation window size $W = 5, 6, 7$. The median runtimes for each variant are plotted with circles, means are plotted with $\times$'s, and error bars indicate the 5th and 95th percentiles. The runtime is to be mostly unaffected by the number of kept permutations, but increases exponentially with the window size.

References

[1] Hafewa Bargaoui, Olfa Belkahla Driss, and Khaled Ghédira. Minimizing makespan in multi-factory flow shop problem using a chemical reaction metaheuristic. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 2919–2926. IEEE, 2016.

[2] Amar Oukil and Ahmed El-Bouri. Ranking dispatching rules in multi-objective dynamic flow shop scheduling: a multi-faceted perspective. *International Journal of Production Research*, 59(2):388–411, 2021.

[3] Harwin Kurniawan, Tanika D Sofianti, Aditya Tirta Pratama, and Prianggada Indra Tanaya. Optimizing production scheduling using genetic algorithm in textile factory. *Journal of System and Management Sciences*, 4(4):27–44, 2014.

[4] MK Marichelvam and M Geetha. Solving tri-objective multistage hybrid flow shop scheduling problems using a discrete firefly algorithm. *International Journal of Intelligent Engineering Informatics*, 2(4):284–303, 2014.

[5] TS Benthem. Solving a multi-objective hybrid flow shop scheduling problem with practical constraints from the food industry. Master's thesis, University of Twente, 2021.

[6] Shuai Chen, Quan-Ke Pan, Liang Gao, and Hong-yan Sang. A population-based iterated greedy algorithm to minimize total flowtime for the distributed blocking flowshop scheduling problem. *Engineering Applications of Artificial Intelligence*, 104:104375, 2021.

[7] Heiner Ackermann, Sandy Heydrich, and Christian Weiß. Analyzing and optimizing the throughput of a pharmaceutical production process. In *Operations Research Proceedings 2019*, pages 591–597. Springer, 2020.

[8] Tanzila Azad and Asif Ahmed Sarja. A comparative analysis of heuristic metaheuristic and exact approach to minimize make span of permutation flow shop scheduling. *American Journal of Industrial Engineering*, 8(1):1–8, 2021.

[9] Eva Vallada, Rubén Ruiz, and Gerardo Minella. Minimising total tardiness in the m-machine flowshop problem: A review and evaluation of heuristics and metaheuristics. *Computers & Operations Research*, 35(4):1350–1373, 2008.

[10] Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

[11] Eva Vallada, Rubén Ruiz, and Jose M Framinan. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3):666–677, 2015.

[12] Michael Pinedo and Khosrow Hadavi. Scheduling: Theory, algorithms and systems development. In Wolfgang Gaul, Achim Bachem, Wilhelm Habenicht, Wolfgang Runge, and Werner W. Stahl, editors, *Operations Research Proceedings 1991*, volume 1991, pages 35–42. Springer, Berlin, Heidelberg, 1992.

[13] Steven Nahmias and Tava Olsen. *Production and Operations Analysis*, chapter 9, page 490. Waveland Press, 7th edition, 2015.

[14] Cecilia E Nugraheni, Luciana Abednego, and Maria Widyarini. A combination of palmer algorithm and gupta algorithm for scheduling problem in apparel industry. *International Journal of Fuzzy Logic Systems*, 11(1):1–12, 2021.

[15] Christophe Sauvey and Nathalie Sauer. Two neh heuristic improvements for flowshop scheduling problem with makespan criterion. *Algorithms*, 13(5):112, 2020.

[16] Yeong-Dae Kim. Heuristics for flowshop scheduling problems minimizing mean tardiness. *Journal of the Operational Research Society*, 44(1):19–28, 1993.

[17] Victor Fernandez-Viagas and Jose M Framinan. NEH-based heuristics for the permutation flowshop scheduling problem to minimise total tardiness. *Computers & Operations Research*, 60:27–36, 2015.

Matthew Gradwohl
Department of Science and Mathematics, Texas A&M University-Central Texas, Killeen TX USA

Guidio Sewa
Institut de Mathématiques et de Sciences Physiques, Porto Novo, Benin Republic.

Oke Blessing Oghojafor
Department of Statistics, University of Lagos. Nigeria
Email address: oghojaforokerogheneblessing@gmail.com

Richard Wilouwou
University of South Brittany, Lorient France.

Muminu Adamu
Department of Statistics, University of Lagos. Nigeria
Email address: madamu@unilag.edu.ng

Christopher Thron*
Department of Science and Mathematics, Texas A&M University-Central Texas, Killeen TX USA
Email address: thron@tamuct.edu