

Article

A New One-Machine Decomposition Technique for Solving the Permutation Flow-Shop Scheduling Problem

Mehrdad Amirghasemi ^{1,*}, Stefan Voß ^{2,*}, Wolfgang Garn ³, Amir Arjomandi ¹ and Robert Ogie ⁴

¹ School of Business, University of Wollongong, Wollongong, NSW 2522, Australia

² Institute of Information Systems, University of Hamburg, Von-Melle-Park 5, 20146 Hamburg, Germany

³ Surrey Business School, University of Surrey, Guildford GU2 7XH, UK

⁴ SMART Infrastructure Facility, University of Wollongong, Wollongong, NSW 2522, Australia

* Correspondence: mehrdad@uow.edu.au (M.A.); stefan.voss@uni-hamburg.de (S.V.)

Abstract

Existing metaheuristics for the permutation flow-shop scheduling problem primarily explore the search space directly. This study presents a new decomposition technique that, unlike those methods, repeatedly solves one-machine subproblems and extends their solutions to all machines in order to minimize the makespan objective. To this end, the proposed algorithm solves the one-machine problem using a novel forward-backward procedure and extends its solutions to all machines through recurrent displacement of jobs. The improvement of the current solution proceeds until no optimal permutation for a single machine can improve the overall permutation for all machines, gradually improving current solutions and directing the search towards obtaining high-quality solutions. To further improve the results, any solution proposed by the one-machine solution strategy is refined by a local search process. An innovative triangular mechanism is also proposed for constructing initial solutions, with the aim of providing a high-quality starting point for the algorithm. The results of computational experiments not only demonstrate the efficiency of the one-machine forward-backward technique, but also indicate that the algorithm is both robust and highly effective in solving the standard benchmark instances.

Keywords: decomposition algorithms; metaheuristics; one-machine scheduling problem; flow-shop scheduling problem; permutation flow-shop scheduling problem

1. Introduction

The permutation flow-shop scheduling problem (PFSP) is a classical optimization problem that also has practical applications in manufacturing [1], logistics, and supply chain management [2]. Novel decomposition-based heuristics can directly influence production costs, resource allocation, and overall operational efficiency. Although most successful metaheuristics improve solutions by exploring the permutation flow-shop search space directly, decomposition-based approaches that repeatedly exploit exact one-machine optimization have received limited attention. To this end, this paper presents an effective decomposition method by solving one-machine subproblems and iteratively extending their solutions to all machines.

Despite the NP-hardness of the one-machine problem, optimal solutions for instances with 1000 jobs can be obtained in milliseconds, whereas some PFSPs with just 50 jobs on three machines remain unsolved by exact methods. This contrast is the key motivation for the proposed novel algorithm, the Recurrent Intelligent Job Displacer (RIJD), which



Academic Editor: Frank Werner

Received: 29 May 2026

Revised: 29 June 2026

Accepted: 6 July 2026

Published: 13 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

decomposes the PFSP into one-machine problems and solves them iteratively using the highly effective method from [3]. While this approach has previously been applied to job shop scheduling [4], the RIJD is the first to utilize it for solving the PFSP.

The RIJD starts with an initial solution, $x^{(0)}$, which is a permutation of the jobs on all machines, repeatedly solves one-machine problems, followed by a local search, and generates a sequence of solutions $x^{(1)}, x^{(2)}, \dots, x^{(z-1)}, x^{(z)}$. All pieces of information in the current solution are used to construct a new one-machine problem, which is solved to provide another one-machine problem. In particular, for computing $x^{(k)}$, it is assumed that the starting and completion times of all operations are based on the solution $x^{(k-1)}$. After providing a solution for the original problem, the RIJD solves the one-machine problem and then uses the solution of the original problem to create another one-machine problem. This recurrent feedback progressively improves the current solutions and leads to a high-quality solution overall.

To further improve the results, and to adopt the solution obtained for the one-machine problem for all machines, the algorithm also refines any solution proposed by the one-machine solution strategy through a local search process with an insertion and a 2-swap neighborhood scheme. An initial solution is needed before the application of the one-machine solution strategy, and such a solution is constructed with an innovative triangular mechanism aimed at generating a high-quality starting point. Hence, three main components synergistically assist the algorithm towards its intelligent job displacement, namely, (i) the one-machine solution technique, (ii) a local search with two neighborhood schemes, and (iii) an innovative triangular method for constructing initial starting points.

The RIJD constructs a solution with its triangular mechanism and, after applying the insertion and 2-swap local searches on such a solution, it sets the starting times of operations and improves them through the recurrent application of the solution technique for the one-machine problem. For this purpose, it solves a one-machine problem for each of the existing machines, and, after applying a local search with insertion and 2-swap neighborhood schemes to such a solution, it extends the solution to all machines. The process of constructing a new solution through the triangular mechanism and improving it with the one-machine problem technique, followed by the application of the local search, continues until a given time limit is reached. An outer loop for this purpose controls an inner loop in which the one-machine problem technique and the local search are applied to an initially constructed solution. The outer loop, which constructs the initial solution with the triangular method, is repeated until a time limit is reached. Although the RIJD has demonstrated strong performance across standard benchmarks, its current diversification mechanisms can become less effective on larger instances, motivating future work on improving scalability, e.g., through parallelization, and search diversification.

The rest of the paper is organized as follows: Section 2 presents the formulation of the permutation flow-shop scheduling problem. Section 3 provides details of related works, and Section 4 describes the RIJD as well as a complete pseudocode for the algorithm. The results of computational experiments are reported in Section 5. Concluding remarks are discussed in Section 6, which also sketches some directions for further research.

2. Problem Formulation

Advanced manufacturing is an area of research that greatly depends on algorithmic design to embed effective decision-making strategies into procedural routines in manufacturing plants. Among the problems existing in this area, the flow-shop scheduling problem (FSP) is involved with circumstances in which all jobs require the same processing order on machines. Since, in many manufacturing plants, jobs are usually carried from one machine to the other machine by conveyors, jobs should have an identical sequence on all machines.

This means that, for instance, if, in an obtained solution, job 2 precedes job 6 on machine 1, on all other machines, these jobs should have the same precedence relationship. Depending on whether jobs have identical or non-identical sequences on machines, the FSP is divided into the permutation FSP (PFSP) and the non-permutation FSP (hereafter denoted as FSP), which are both NP-hard problems. In [5], these two groups of problems are compared, showing that, in very special cases, the objective function of the PFSP is, by a factor of the number of machines, larger than that of the FSP.

The PFSP can be simply defined as a set of n jobs and m machines in which each job should be processed on the machines 1 through m , one after another. In effect, every job is processed exactly once on each machine and all jobs follow the same machine sequence. Each machine can process only one job at a time, and each job can be processed on only one machine at a time. Processing times are deterministic and known in advance and pre-emption is not allowed, meaning that, once processing of a job begins on a machine, it must continue uninterrupted until completion. The formal definition of the PFSP can be found in [6,7]. By finding a permutation $\Pi = (\pi_1, \pi_2, \dots, \pi_n)$ of jobs, the goal is to minimize the makespan, which is the completion time of the last job on the last machine.

Note that, despite the significant difference between the sizes of their solution space, $(n!)^m$ versus $n!$, work on the FSP has not produced significantly better solutions compared to the solutions obtained by the PFSP [8]. This describes why most of the attention in the literature has been focused on the PFSP rather than on the FSP.

In the permutation Π , π_k shows the number of the k^{th} job in the sequence of jobs, with the elements of the sequence being executed one after another, on machines 1 through m . The makespan of the problem can be calculated by rearranging the jobs, based on a given permutation Π , in a matrix F , which shows the processing time of each operation associated with a job. The element f_{ki} of matrix F shows the processing time of job π_k on machine i . Another matrix is matrix C , denoting the earliest completion time of job π_k on the machine i as c_{ki} , and the value of c_{nm} shows the exact amount of the makespan. Hence, the goal can be stated to be finding a permutation Π that can minimize c_{nm} .

In general, the calculation of the completion times is performed in four simple steps: First, the completion time of the first job in the sequence is saved in c_{11} . Then, the completion times of the first job on all other machines are calculated, one after another and based on the completion time on the previous machine. In the third step, the completion times of all jobs on the first machine are calculated, again one after another and based on the completion time of the previous job in the sequence. In the fourth step, any element (cell) of matrix C whose upper and left cells have been specified can be calculated. In other words, for calculating the completion time of the k^{th} job in the sequence on machine i , c_{ki} , both the completion time of its previous job on machine i and the completion time of the k^{th} job on the previous machine are required. In effect, the maximum between these two values plus the processing time of the k^{th} job in the sequence on machine i comprise the completion time of this job on machine i , c_{ki} .

The presented formulation has been enhanced by the notions provided in [9]. These notions, which were originally developed to increase the speed of an algorithm called the NEH algorithm [10], can change the above forward formulation to an effective forward-backward formulation. In the calculation of the makespan, the aforementioned formulation is based on the earliest completion times, c_{ki} , whereas, in a backward formulation, the equivalence of the earliest completion time is the tail of an operation. Hence, the only fixation needed to convert the above forward formulation to an effective forward-backward formulation is the addition of a set of variables that can facilitate the calculation of tails.

Shown in a matrix Q and represented for the k^{th} job in the sequence, job π_k , on machine i , tails, q_{ki} , are calculated in the reverse fashion in which completion times c_{ki} , shown in

matrix C , are calculated. The same goes for the four steps mentioned in the calculation of the completion times. The following four steps can easily calculate the tails: As a first step, the tail of the last job on the last machine is calculated, and, as the second step, the cells of the last row of Q are calculated. The third step calculates the cells of the last column of Q . In the final step, any cell whose lower and right cells have been specified can be calculated. Technically, the matrix Q is calculated as follows:

$$q_{nm} = f_{nm} \quad (1)$$

$$q_{ni} = q_{n,i+1} + f_{ni} \quad i = m-1, m-2, \dots, 1 \quad (2)$$

$$q_{km} = q_{k+1,m} + f_{km} \quad k = n-1, n-2, \dots, 1 \quad (3)$$

$$q_{ki} = \max\{q_{k,i+1}, q_{k+1,i}\} + f_{ki} \quad \begin{matrix} k = n-1, n-2, \dots, 1 \\ i = m-1, m-2, \dots, 1 \end{matrix} \quad (4)$$

Based on the backward formulation, the goal in the PFSP can be stated to be finding a permutation Π that can minimize q_{11} . The equality of $c_{nm} = q_{11}$ indicates that both the forward and backward formulations are aimed at minimizing the same quantity expressed in two different ways. This is similar to the primal and dual formulation of optimization problems, with both dual and primal solution strategies leading to the same value of the objective function.

In analogy with the primal-dual solution strategy, which needs both primal and dual formulations, and by having access to both the C and Q matrices, one can evaluate the effect of some changes to the permutation on the makespan comparatively faster. First performed by Taillard in [9,11], this forward-backward approach has demonstrated momentous efficiency. Additionally, Taillard's speedup was extended and generalized in [12] to apply to the PFSP with objectives other than the makespan, facilitating the adaptability of our proposed method to such PFSP variants.

The other characteristics of the problem that can be embedded in its formulation are the contributions each job makes to the critical path [13], where the critical path represents the longest sequence of dependent operations from the first to the last machine. Despite the fact that all jobs participate in the critical path, every one of them can have a different contribution in the creation of this path. In effect, the contribution of the k^{th} job in the sequence, job π_k , is denoted with w_k , and equals the portion of this job processing time lying on the critical path.

$$w_k = \sum_{i=d_{k-1}}^{d_k} f_{ki} \quad (5)$$

In this equation, d_k shows the last machine on which the k^{th} job in the sequence, job π_k , has a critical operation. This notation has a vital role in the description of the RIJD, because, by rearranging the permutation Π and relocating the jobs which have a higher contribution, w_k , the RIJD is aimed at incurring the largest decreases in the makespan.

In other words, the PFSP is seen as a min-max problem in which a minimizer component (agent) tries to find $d_1, d_2, \dots, d_{n-1}$, through a permutation Π of jobs, and a maximizer component recreates another critical path based on the new matrix F provided by the minimizer component. The interaction between the maximizer and minimizer component is the repeated decrease in the makespan. This interaction can be simply stated with the following equation:

$$\text{makespan} = \min_F \left(\max_{d_1, d_2, \dots, d_{n-1}} \sum_{k=1}^n \sum_{i=d_{k-1}}^{d_k} f_{ki} \right) \quad (6)$$

It is such a perspective that is used by the local search component of the RIJD to incur the largest possible decreases in the makespan. For this purpose, the algorithm repeatedly proceeds from one permutation to the other, and continues to do so until no improvement is possible. Guided by the above formulation, the local search component of the RIJD looks for those rearrangements of jobs which have the chance of decreasing the makespan and reduces some of the unfruitful arrangements.

3. Related Works

The solution approaches for the PFSP can be categorized into exact and heuristic methods. The combinatorial node explosion associated with this problem reduces the practicality of exact methods, making heuristics the most viable strategy. It is also worth noting that exact and heuristic methods have been combined and are often referred to as matheuristics in the literature [14].

Given the vast body of literature on flow-shop scheduling, we have structured this section as follows: first, we present the background, motivation, and rationale behind the proposed algorithm; next, we briefly discuss various literature surveys of the FSP and PFSP; and, finally, we detail the approaches most closely related to the RIJD.

3.1. Background and Motivation

Whereas, for the one-machine problem [3], despite its NP-hardness [15], the optimized solution of a typical problem with 1000 jobs can be obtained in a matter of milliseconds, the optimal solution of some PFSP with 50 jobs on three machines is still outside the reach of the current exact algorithms. Based on this contrast, the algorithm presented in this paper effectively uses the solution strategy of the one-machine problem to tackle the PFSP.

It is worth noting that, although six years after the development of this famous method for the one-machine problem, it was used to solve the job shop scheduling problem [4], the proposed RIJD is the first algorithm which uses this solution strategy in the context of solving the permutation flow-shop scheduling problem. In effect, despite the success of decomposition in job shop scheduling, it has not been exploited to repeatedly guide the search for the PFSP through exact one-machine optimization, leaving a gap that the proposed RIJD addresses.

3.2. Literature Surveys

A survey of some earlier approaches for flow-shop scheduling is presented in [16], highlighting that minimizing the makespan is equivalent to minimizing total delay, comprising run-in, between-jobs, and run-out delays, which can be a useful insight for heuristic design. Run-in and run-out delays reflect idle time before the first and after the last job, respectively, while between-jobs delays are more undesirable when occurring later in the sequence, as they are more likely to affect the makespan, which equals total delay plus total processing time divided by the number of machines.

Extensive literature surveys of the efficient algorithms presented for the FSP can be found in [17,18]. Other surveys and reviews include [6,19]. It is worth noting that, in [20], comparisons of different heuristic and metaheuristic algorithms are performed and it is concluded that the NEH heuristic of [10] and the iterated local search metaheuristic of [21] are the top-performing approaches.

Additional surveys regarding the comparison of heuristics and metaheuristics include [22], which features an empirical evaluation. From the metaheuristics side, comparison frameworks are proposed, too; see, e.g., ref. [23].

Finally, we should note that the FSP and the PFSP are the basis of a wealth of more general flow-shop-type problems in practice; see, e.g., ref. [2].

3.3. Key Related Approaches and Connections

Here, the key work related to the method presented in this paper is briefly surveyed. In most mentioned algorithms, the criterion is the makespan; however, many other criteria can be used for the problem. A general empirical investigation of scheduling performance criteria is discussed in [24].

The most related method is the one-machine sequencing algorithm presented in [3], and, since we have equipped this method with a forward-backward mechanism, it will be discussed along with its extended mechanism in the next section. The next related method is the method presented in [25], which examines some aspects of scatter search in the PFSP and presents several efficient neighborhood schemes for the PFSP.

In particular, neighborhood schemes have been created with INS (insertion) and API (adjacent pairwise interchange). While INS can move a job from any position of the permutation and insert it to any other position, API can only swap the positions of two adjacent jobs in the permutation. This means that, by using INS, each permutation can be obtained from another permutation with at most $n - 1$ moves, whereas API needs at most $n(n - 1)/2$ moves to convert a permutation to another. The neighborhood schemes considered are the following: (i) Full-INS, (ii) Basic INS+BLOCKS, (iii) Extended INS+BLOCKS, (iv) Large INS+BLOCKS, and (v) FULL-API.

The first scheme contains all $(n - 1)^2$ permutations obtained by the insert operation. The second scheme, however, includes only moves that relocate a job inside a critical block (a sequence of critical jobs) to the first or the last position of the block, and contains approximately $2n$ moves. On the other hand, the third scheme includes moves that relocate a job from a block to one of the positions of its immediately preceding block or immediately succeeding block. The fourth scheme contains any move that can relocate a job from one block to any other block, and the fifth scheme contains all moves that manage to swap the positions of two adjacent jobs in the permutation and consists of exactly $n - 1$ moves.

In [26], a particle swarm optimization approach is presented that works based on an efficient change of job permutations. The method seems to be an improvement over the method presented in [27]. Compared to their previous method, several new mutation operators are added to the procedure. In effect, they use a combination of ten different mutation operators, ranging from inserting up to four segments to random positions through reversing a segment to inserting up to three single jobs in random positions.

Kuo et al. [28] have developed an efficient scheduling algorithm based on a hybrid particle swarm optimization model in which a random-key encoding scheme is used for the representation of individuals. For instance, a random key like (0.2, 0.6, 0.1, 0.5) shows the permutation of (2, 4, 1, 3) for the jobs.

The use of a vector of real numbers instead of a permutation, for the encoding purpose, has its advantages as well as its disadvantages. On the one hand, the positions of particles change in an n -dimensional Cartesian space, which makes the moves advantageously smooth; on the other hand, many random keys can be mapped into a single permutation, which is an obvious disadvantage.

The hybrid consists of an individual enhancement scheme and a particle swarm optimization component. Whereas the individual enhancement scheme searches each individual's neighborhood in order to find a better solution, the other component moves every particle in an n -dimensional Cartesian space and evaluates the objective function of the particle by transforming the random-key encoding to its corresponding permutation and converting that permutation into an objective function value.

In [29], some efficient heuristics for the PFSP are presented. First, the results of some observations are provided relating to finding optimal solutions of instances with 9 jobs and 10 machines, whose processing times are generated uniformly in the range between 1 and

100. The distribution of the optimal solutions is almost symmetrical. Statistical tests have neither confirmed nor refuted whether this distribution is normal. Quite interestingly, a random solution is about 20 percent over the optimum, while only 2 out of 10,000 solutions are within 1 percent of the optimum.

The author also facilitated the speed of the NEH algorithm by calculating the earliest completion time as well as the tail of operations, with the tail of each operation indicating the duration between the starting time and the end of the partial schedule. Assuming that, in the NEH algorithm, (i) the first $k - 1$ longest jobs have been sequenced, (ii) e_{ji} denotes the earliest completion time of j , the job in the sequence on the machine i , and (iii) q_{ji} denotes its tail, the values of e_{ji} and q_{ji} are calculated as follows:

$$e_{j0} = 0 \quad \& \quad e_{0i} = 0 \quad (7)$$

$$e_{ji} = \max \{e_{j-1,i}, e_{j,i-1}\} + t_{\text{num}(j),i} \quad (j = 1, 2, \dots, k-1) \\ (i = 1, 2, \dots, m) \quad (8)$$

$$q_{ki} = 0 \quad \& \quad q_{j,m+1} = 0 \quad (9)$$

$$q_{ji} = \max \{q_{j+1,i}, q_{j,i+1}\} + t_{\text{num}(j),i} \quad (j = k-1, k-2, \dots, 1) \\ (i = m, m-1, \dots, 1) \quad (10)$$

Based on this calculation, the evaluation of inserting a new job in any position of the permutation can be performed in $O(m)$, and, since there are k possible positions for such insertion, the best position can be found in $O(km)$. This decreases the complexity of the NEH algorithm from $O(n^3m)$ to $O(n^2m)$.

As well as improving the performance of the NEH algorithm, the interesting feature of the above calculation allows all neighbors in a local search to be evaluated efficiently. In the employed tabu search, each job can leave its current position in the sequence and locate itself in the best possible place based on the above calculation. In an unconventional use of the tabu list in preventing the cycles, if the obtained objective function value exists in the tabu list, the move is prevented. In other words, if the size of the tabu list is β , then each β consecutive solutions should have different makespan values. It is worth highlighting that a new NEH heuristic has also been proposed by [30], incorporating new priority and tie-breaking rules.

These methods efficiently evaluate neighborhood moves and have become the foundation of many high-performing PFSP heuristics. However, they still explore the PFSP search space directly, whereas the proposed RIJD repeatedly optimizes decomposed one-machine subproblems before projecting the resulting permutation back to the original problem.

In [13], a fast tabu search algorithm for the PFSP is presented which reduces computation by exploiting block properties along a critical path derived from a job permutation. In effect, moves are restricted to operations at the start or end of each block, or involve shifting a job to a neighboring block, guided by a parameter, ϵ , determined by the job-to-machine ratio. To enhance efficiency, the algorithm maintains the longest path from each node to others in the south-east of the job-machine matrix, enabling faster move evaluations without relying on Taillard's mechanism.

These neighborhood structures provide effective local intensification but remain restricted to incremental modifications of the current PFSP permutation. In contrast, the RIJD generates larger structural changes by replacing the current permutation with one obtained from an optimally solved one-machine subproblem before applying the local search.

Lastly, recent deep learning approaches have also been applied to the PFSP. Ref. [31] proposed PFSPNet, a deep reinforcement learning model incorporating Long Short-Term Memory (LSTM) networks and including a refinement strategy based on the NEH heuristic. Their method, however, does not utilize the concept of a critical path and is primarily

compared against traditional heuristics, showing relatively good performance. Similarly, Li et al. [32] introduced a graph-based imitation learning framework for the PFSP, aiming to reduce the number of model parameters and enhance scalability by leveraging graph-based feature representations to tackle large-scale scheduling problems.

Apart from the effective approaches mentioned earlier, several other notable approaches have been proposed, including tabu search, simulated annealing, genetic algorithms, scatter search, particle swarm optimization, multi-objective particle swarm optimization, ant colony optimization, an evolutionary algorithm, and discrete differential evolution. In addition to these metaheuristics, an efficient branch-and-bound algorithm has also been proposed. All of these and other related work are summarized in Table 1.

Table 1. Summary of the reviewed studies and their key contribution related to the proposed algorithm.

Citation	Key Related Contribution
Carlier (1982) [3]	One-machine problem solution strategy.
Garey et al. (1976) [15]	NP-hardness of the one-machine problem.
Adams et al. (1988) [4]	Application of one-machine problem solution strategy to the job shop scheduling problem.
King and Spachis (1980) [16]	Early survey highlighting the equivalence between minimizing the makespan and total delay.
Dudek et al. (1992) [17]	Literature survey on efficient algorithms for the FSP.
Blazewicz et al. (1996) [18]	Survey of flow-shop scheduling algorithms.
Framinan et al. (2004) [6]	Survey of developments in flow-shop scheduling.
Gupta and Stafford (2006) [19]	Survey of various methods for flow-shop scheduling.
Ruiz and Maroto (2005) [20]	Comparison of heuristic and metaheuristic algorithms for the PFSP, highlighting the NEH heuristic and iterated local search.
Fernandez-Viagas et al. (2017) [22]	Empirical evaluation of heuristic and metaheuristic methods for the PFSP.
de Armas et al. (2022) [23]	Framework for comparing metaheuristics for flow-shop scheduling.
Ying et al. (2023) [2]	Application of the FSP and PFSP in general flow-shop-type problems.
De Snoo et al. (2011) [24]	Empirical investigation of scheduling performance criteria.
Nowicki and Smutnicki (2006) [25]	Scatter search approach for the PFSP with efficient neighborhood schemes (INS and API).
Lian et al. (2008) [26]	Particle swarm optimization with new mutation operators for the PFSP.
Kuo et al. (2009) [28]	Hybrid particle swarm optimization for the PFSP with random-key encoding.
Taillard (1998) [29]	Heuristics for the PFSP, including an improved NEH algorithm with improved complexity.
Liu et al. (2017) [30]	Improved NEH heuristic with new priority and tie-breaking rules.
Nowicki and Smutnicki (1996) [13]	Fast tabu search for the PFSP using critical path and block properties.

Table 1. *Cont.*

Citation	Key Related Contribution
Taillard (1990) [11]	Tabu search approach for the PFSP.
Ogbu and Smith (1990) [33]	Simulated annealing approach for the PFSP.
Reeves (1995) [8]	Genetic algorithm for the PFSP.
Iyer and Saxena (2004) [34]	Genetic algorithm for the PFSP.
Chang et al. (2013) [35]	Genetic algorithm for the PFSP.
Nowicki and Smutnicki (2006) [25]	Scatter search for the PFSP.
Lian et al. (2008) [26]	Particle swarm optimization for the PFSP.
Rahimi-Vahed and Mirghorbani (2007) [36]	Multi-objective particle swarm optimization for the PFSP.
Yagmahan and Yenisey (2010) [37]	Ant colony optimization for the PFSP.
Amirghasemi and Zamani (2017) [38]	Evolutionary algorithm for the PFSP.
de Fátima Morais et al. (2022) [39]	Discrete differential evolution for the PFSP.
Gmys et al. (2020) [40]	New branch-and-bound algorithm for the PFSP.
Pan et al. (2023) [31]	Deep reinforcement learning approach (PFSPNet) combining LSTM networks with an NEH-based refinement strategy.
Li et al. (2024) [32]	Graph-based imitation learning model for the PFSP.

Overall, the surveyed approaches have achieved remarkable performance through increasingly sophisticated neighborhood structures, population-based search, and learning-based techniques. The RIJD complements these directions by introducing a fundamentally different decomposition-based framework that repeatedly solves one-machine subproblems to optimality and integrates their solutions through recurrent feedback and local search.

4. The Recurrent Intelligent Job Displacer (RIJD)

The distinguishing feature of the RIJD, in contrast with conventional PFSP heuristics that search the solution space directly, is the repeated decomposition of the PFSP into one-machine subproblems. It then solves them optimally and projects their solutions back to the original problem through recurrent feedback. This decomposition framework is complemented by a novel triangular construction method for generating high-quality initial solutions, as well as a forward-backward extension of Carlier's one-machine algorithm. Together, these provide the three key components of the RIJD.

In effect, the RIJD works based on the strategy of successive displacement, which is involved with recurrent feedback proposed for an incumbent solution. For this purpose, as long as the incumbent solution has a significant difference with its revised solution, the RIJD revises the incumbent solution based on the received feedback. In fact, an improved version of the one-machine method, which produces an optimal permutation for a one-machine problem, has been used to revise the incumbent solution.

Since any solution of the PFSP is represented as a permutation of jobs, the feedback received from the one-machine method replaces a permutation with another one. The non-guaranteed optimal solution for all machines is replaced with the guaranteed optimal solution presented for one machine. Assuming that the permutation of jobs on all other machines is the same as the current solution, the optimal permutation of jobs for a single machine is calculated through the use of a one-machine problem (OMP).

In addition to this key feedback component, which works based on the one-machine problem, a triangular method and a local search have been embedded in the RIJD. While

the triangular method, as a construction technique, provides initial solutions, the employed local search, as a fine-tuning progression, improves the results proposed by the one-machine solution strategy.

After an initial solution is generated, the process of solving one-machine problems followed by the local search starts, and, as long as the optimal permutation for a single machine can improve the incumbent solution, the process continues. The triangular method, the one-machine solution strategy, and the local search are the three main components of the algorithm. First, we describe the triangular method.

4.1. Triangular Construction Method

Let us introduce the notion of an arrow as moving towards either the east ($\rightarrow$) or the south ($\downarrow$) when considering a path from the cell $(1, 1)$ to the cell (n, m) as a critical path.

Since the PFSP is considered a min-max problem, the maximizer agent should be deprived of permutations in which large cells of the matrix can be connected to create a long critical path between the cell $(1, 1)$ and the cell (n, m) . Showing the number of combinations of m out of n objects by C_m^n , between these two cells there are C_{m-1}^{m+n-2} paths.

The reason is that each path consists of only $n - 1$ south and $m - 1$ east arrows, and, consequently, the $m - 1$ east arrows can be selected as any of the $(n - 1) + (m - 1)$, which is $m + n - 2$ arrows. Figure 1 shows a sample problem, and Figure 2 depicts a critical path for a particular arrangement of jobs selected randomly.

		Machines									
		1	2	3	4	5	6	7	8	9	10
Jobs	j ₁	73	44	26	70	57	52	48	91	62	64
	j ₂	73	80	32	28	80	21	55	48	75	93
	j ₃	13	12	11	97	81	24	49	84	45	80
	j ₄	16	50	83	50	44	95	82	67	71	5
	j ₅	30	31	95	74	15	75	91	22	81	67
	j ₆	75	78	33	45	76	98	24	36	29	88
	j ₇	10	92	52	26	39	12	2	76	9	62
	j ₈	34	64	36	0	67	8	99	51	9	11
	j ₉	98	15	78	15	18	94	53	81	74	78
	j ₁₀	59	58	92	65	15	78	95	0	6	43

Figure 1. The processing times of jobs on machines for a sample 10×10 instance, with the arrows showing the maximizer agent moving towards either the east or the south.

		Machines									
		1	2	3	4	5	6	7	8	9	10
Jobs	j ₂	<u>73</u>	<u>80</u>	32	28	80	21	55	48	75	93
	j ₁₀	59	<u>58</u>	<u>92</u>	65	15	78	95	0	6	43
	j ₉	98	15	<u>78</u>	<u>15</u>	18	94	53	81	74	78
	j ₃	13	12	11	<u>97</u>	<u>81</u>	24	49	84	45	80
	j ₁	73	44	26	70	<u>57</u>	<u>52</u>	48	91	62	64
	j ₅	30	31	95	74	15	<u>75</u>	91	22	81	67
	j ₄	16	50	83	50	44	<u>95</u>	<u>82</u>	67	71	5
	j ₈	34	64	36	0	67	8	<u>99</u>	<u>51</u>	9	11
	j ₇	10	92	52	26	39	12	2	<u>76</u>	<u>9</u>	<u>62</u>
	j ₆	75	78	33	45	76	98	24	36	29	<u>88</u>

Figure 2. A random permutation of jobs in the sample instance and its critical path, shown in shaded cells, with the sum of the underlined processing times equating to a makespan of 1320.

The arrows should not be in the direction of north or west, and this prevents the maximizer agent from connecting many cells to many other cells. This is the characteristic exploited by the current algorithm to initially arrange the jobs. For this purpose, the large cells can be gathered in two contrasting triangular areas that, based on south and east arrows, do not have access to one another. Figure 3 shows these two triangles in the grid.

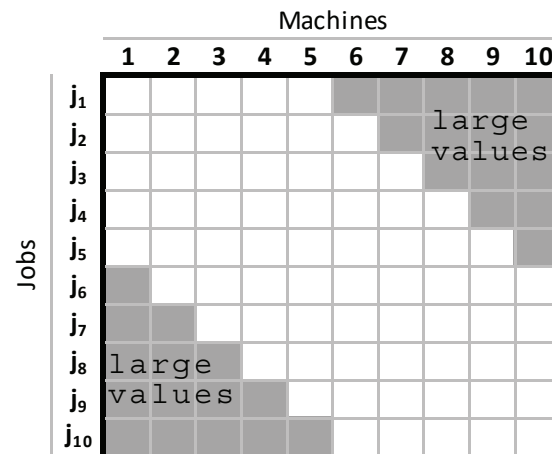


Figure 3. The triangular construction mechanism in which large values fill the shaded parts.

The other mechanism for depriving the agent of connecting large cells to one another is to put small cells in the highly related triangular areas through which the south and east arrows of the path normally run. This is shown in Figure 4.

For each of the two cases, we proceed as follows: For the first mechanism, the sum of the first k operations and the sum of the last k operations for each job, for k from 1 to $\lfloor n/2 \rfloor$, are calculated. Then, a random permutation is created representing which location should be filled first. Finally, locations are selected one after another and an unselected job is selected which, based on the sums calculated, best fits in the location.

For instance, if the random permutation generated is (4, 7, 1, 3, 2, 6, 5), it means that the job that should be located in location 4 should be identified first, then the job that should be located in position 7, etc., and, at last, the job that should be located in location 5 should be identified.

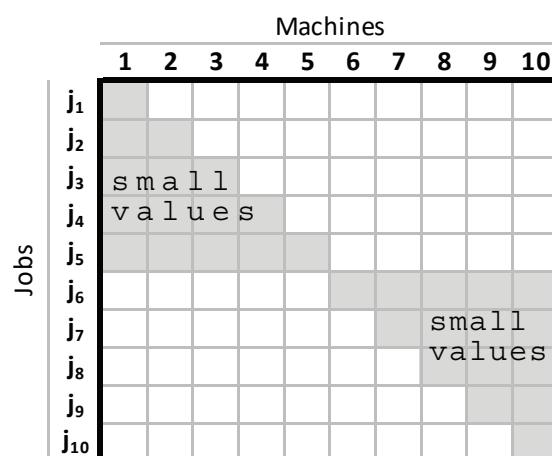


Figure 4. The triangular construction mechanism in which small values fill the shaded parts.

As can be seen in this case for the job located in location 4, there are n options, and among them the best can be selected. Then, for the job located in location 7, there are $n - 1$ options, and, for the job located in location 5, there is only one option left. This

emphasizes how the quality of the generated solution can be related to the randomly generated permutation.

The construction of the solution based on the second mechanism is the same as that of the first case. Because, in the second case, the small cells are placed in the other two triangular areas, the only point that needs consideration is that, instead of large values, small values should be chosen. The two methods can be easily integrated so that small and large values can be placed simultaneously in their corresponding triangular areas. Figures 5 and 6 show two solutions, constructed by the first and second mechanisms, respectively.

		Machines									
		1	2	3	4	5	6	7	8	9	10
Jobs	j ₅	<u>30</u>	<u>31</u>	<u>95</u>	<u>74</u>	15	75	91	22	81	67
	j ₁	73	44	26	<u>70</u>	57	52	48	91	62	64
	j ₃	13	12	11	<u>97</u>	<u>81</u>	24	49	84	45	80
	j ₂	73	80	32	28	<u>80</u>	21	55	48	75	93
	j ₇	10	92	52	26	<u>39</u>	12	2	76	9	62
	j ₉	98	15	78	15	<u>18</u>	<u>94</u>	53	81	74	78
	j ₁₀	59	58	92	65	15	<u>78</u>	95	0	6	43
	j ₄	16	50	83	50	44	<u>95</u>	<u>82</u>	67	71	5
	j ₈	34	64	36	0	67	8	<u>99</u>	<u>51</u>	9	11
	j ₆	75	78	33	45	76	98	24	<u>36</u>	<u>29</u>	<u>88</u>

Figure 5. A solution constructed with the triangular construction mechanism for the sample instance with the makespan of 1267 and its corresponding critical path. Larger values in their corresponding triangular areas are shown in bold.

The triangular construction method steps can be simply summarized as follows:

1. Calculate the cumulative processing time sums for each job.
2. Generate a random order of positions in the permutation.
3. Fill each position by assigning the unselected job that, probabilistically, best fits the triangular structure (placing either large or small values in the designated triangular regions).
4. Repeat until all positions have been assigned.

		Machines									
		1	2	3	4	5	6	7	8	9	10
Jobs	j ₈	34	64	36	0	67	8	99	51	9	11
	j ₅	30	31	<u>95</u>	<u>74</u>	15	75	91	22	81	67
	j ₁	73	44	26	<u>70</u>	57	52	48	91	62	64
	j ₃	13	12	11	<u>97</u>	<u>81</u>	<u>24</u>	49	84	45	80
	j ₉	98	15	78	15	18	<u>94</u>	<u>53</u>	<u>81</u>	<u>74</u>	<u>78</u>
	j ₇	10	92	52	26	39	12	2	76	9	62
	j ₁₀	59	58	92	65	15	78	95	0	6	43
	j ₆	75	78	33	45	76	98	24	36	29	88
	j ₂	73	80	32	28	80	21	55	48	75	93
	j ₄	16	50	83	50	44	95	82	67	71	<u>5</u>

Figure 6. A solution constructed with the triangular construction mechanism for the sample instance with the makespan of 1246 and its corresponding critical path. Smaller values in their corresponding triangular areas are shown in bold.

4.2. Forward-Backward One-Machine Solution Strategy

The forward-backward one-machine solution strategy, as an improvement over the one-machine sequencing method [3], is a relaxation of the job shop problem (JSP). This strategy had a significant role in the development of some of the highly effective algorithms developed for solving the JSP.

In the JSP with n jobs, if we assume that there are only three machines, namely, M1, M2, and M3, in which M1 and M3 are non-bottleneck machines, and the job i has to spend a_i , p_i , and q_i on each of these three machines, respectively, minimizing the makespan of such a JSP is equivalent to solving an OMP for M2. The OMP, as a pure sequencing problem, can be simply stated as follows: There is one machine and n independent jobs are to be sequenced on this machine. The job i is available at time a_i and needs the processing time of p_i . When the job i is completed, it has to stay in the system for q_i periods of time before it is released. The goal is to find a sequence of jobs that leads to the minimum release time of the last job from the system. Surprisingly, this seemingly simple problem is an NP-hard problem. However, the solution strategy presented in [3] has solved typical OMPs with between 50 and 10,000 jobs to optimality. By changing the roles of a_i and q_i , we present a parallel method that can solve some of the instances which are very hard to solve for the original solution strategy.

To find an optimal solution of the OMP, the Carlier method repeatedly employs a method attributed to Schrage [41] in a branch-and-bound framework. In the Schrage method, among the available jobs, a job k is selected whose q_k is maximum. Carlier's method owes its effectiveness to the fact that solutions obtained by this simple rule have some non-negligible chance of being optimal. In effect, if no critical job exists on the longest path of the conjunctive graph created for the corresponding OMP, the solution obtained by the Schrage method is optimal.

In the case of an existing critical job c , two branches in the search tree are created to take this job out of criticality by changing a_c and q_c . A critical job c is associated with a critical block, and, by changing a_c and q_c , it is forced to appear after and before its corresponding critical block, respectively. For creating the forward-backward one-machine solution strategy, the roles of a_c and q_c are interchanged. In other words, instead of solving the original problem, its mirroring problem is solved.

Decomposing the PFSP Into One-Machine Subproblems

To see how the PFSP is converted into small one-machine subproblems, it is helpful to see the problem in a disjunctive graph representation, which is typically used for JSPs. Figure 7 illustrates a fixation of disjunctive arcs for a typical PFSP with n jobs and m machines.

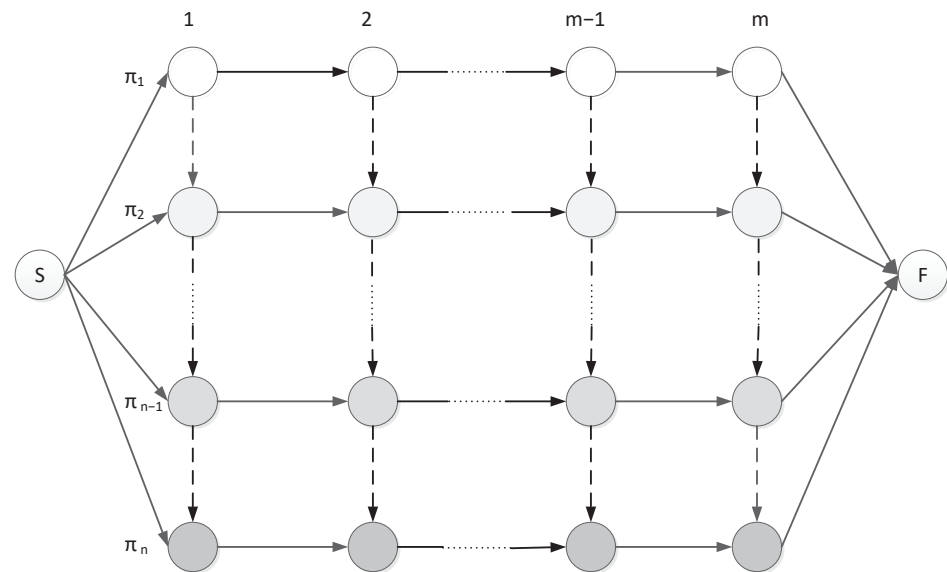


Figure 7. A fixation of the disjunctive arcs of a sample PFSP.

The OMP which is passed to the Carlier method is derived by deleting the arcs corresponding to operations of the machine for which the optimal order of jobs is to be computed. Assuming that such a machine is machine 2, the arcs associated with machine 2 are deleted, and, based on the remaining arcs, machine 2 is passed as an OMP to the forward-backward Carlier algorithm to solve. Figure 8 shows the deletion of the arcs associated with machine 2 and the preserving of the other arcs for creating an OMP, based on machine 2.

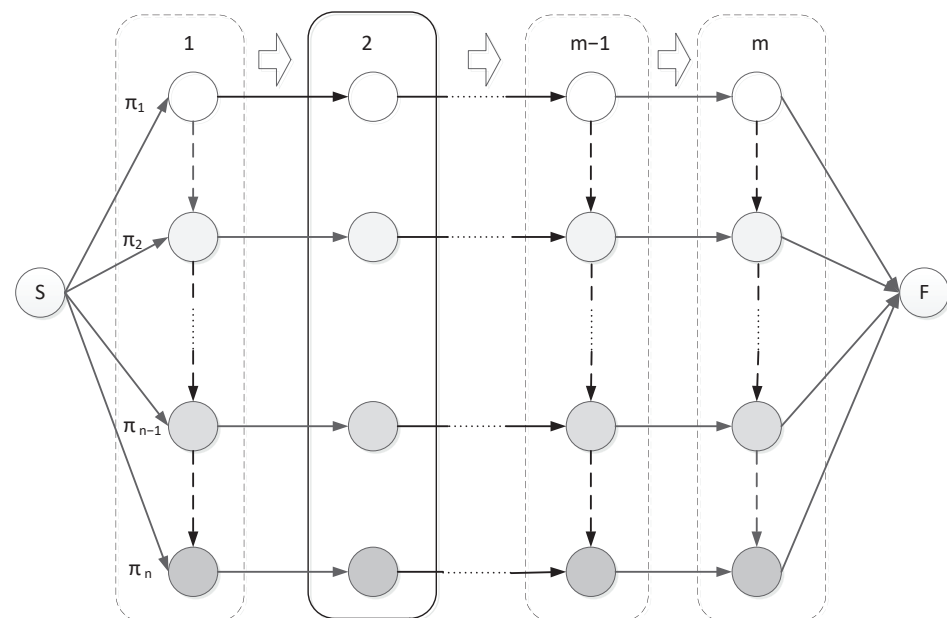


Figure 8. Removing the arcs associated with machine 2 for solving the corresponding one-machine problem.

Since any instance of the one-machine problem is defined by the three values for each job, namely, *head* (a_i), *duration* (p_i), and *tail* (q_i), based on the remaining arcs, these three values are calculated for each job. By the head, we mean the shortest time in which the corresponding job is available to start, and duration denotes the time needed for the processing of the job. The tail of each job shows the longest path from the corresponding

node to the end of the graph; in other words, the tail shows the amount of time the job has to be in the system after the completion of its process.

It is clear that the information about the duration is available and needs no calculation. The calculation of the head also needs no calculation because it is equivalent to the completion time of the previous operation of the corresponding job on the previous machine. The value of the tail needs only few calculations, because, based on Figure 8, the tail of a job can be simply obtained by the backward completion time calculated from the end to the start of the next operation of the corresponding job on the next machine. Hence, the information needed to solve the OMP is obtained with virtually no computation cost.

4.3. The RIJD Solution Strategy

Having described all individual components of the RIJD, we will now explain in this section the high-level solution strategy and detailed procedures. Figure 9 shows the pseudocode of the RIJD. The pseudocode starts with a loop, with line 5 in this loop constructing an initial solution with the triangular construction method.

```

01 RIJD ()
02 {
03   while (termination criterion has not been reached)
04   {
05     construct a solution with the triangular mechanism and call it x;
06     apply insertion local search followed by 2-swap on x;
07     improvement=true;
08     while (improvement)
09     {
10       for k = 1; k<=numberOfMachines; k++
11       {
12         improvement=false;
13         Fix the starting times of jobs based on the solution x;
14         Solve the  $k^{th}$  machine problem and assign the result,  $\pi$ , to all machines;
15         Apply insertion local search followed by 2-swap on  $\pi$  and call the result  $\Pi$ ;
16         if (the quality of solution  $\Pi$  is better than that of solution x)
17         {
18           Set x to  $\Pi$ ;
19           improvement=true;
20           break;
21         }
22       } //end for (k=1 ...
23     } //end while (improvement ...
24   } //end while termination criterion ...
25 } //end

```

Figure 9. The pseudocode of the RIJD.

For this purpose, first the cumulative sum of processing times for the 1st, 2nd, ..., and mth machine are calculated, and then they are sorted. These operations are performed once for each instance and are used for generating any construction solution for the corresponding instance. Hence, by ignoring these amortized operations, only the generation of a random permutation of positions comprises the main operations of constructing a solution. Despite such a speed in generating a solution, the solutions generated by this construction method can be of a significantly higher quality than random solutions.

Further, to enhance diversification in the triangular solution construction procedure, two modifications have been made. First, the order in which jobs are considered for allocation is guided by a uniformly random permutation of jobs. Second, with a low probability of 1%, in each iteration, a uniformly random permutation is generated as the initial solution instead of using the triangular mechanisms.

The employed local searches, in lines 6 and 15, use the critical path of the current solution as a guide to focus on the promising regions of the neighborhood, helping to ignore many 2-swap and insertion moves which are unfruitful.

The main part of the RIJD starts in line 8, aiming to divide the PFSP into small one-machine subproblems and solving each subproblem to optimality through calling the

forward-backward Carlier algorithm, which operates, with equal probability (50%), either as the Carlier method or as its mirror.

Line 14 of the pseudocode applies the output of the forward-backward Carlier method, which is an optimal solution for the corresponding OMP, to the PFSP. In other words, the sequence proposed by the forward-backward Carlier algorithm for machine k is extended to all machines and the new makespan for the PFSP is calculated. It is this solution that, in line 15, undergoes a round of possible improvements by the 2-swap and insertion local searches and becomes a new possible starting point for the loop started in line 8.

If the improved solution cannot improve the current solution, the loop started at line 8 is terminated and the loop started at line 3 continues by generating a new solution through the triangular mechanism for a new starting point. The overall solution of the problem is the best solution that is obtained during the entire search process.

As mentioned, for solving the OMP, the forward-backward Carlier method is used, in which, alternatively, either the original problem or its mirror is solved. The pseudocode presented in Figure 10 shows how this mirroring is performed. As the pseudocode shows, the *mirror problem* is simply derived by swapping the head and tail values of each job. In other words, the mirror problem is obtained by interchanging the head (a_i) and tail (q_i) values of every job before invoking the standard Carlier algorithm. Once the mirrored problem is solved, the resulting start times are transformed back to the original problem by reflecting them about the makespan.

```

01 mirrorCarlierMethod()
02 {
03   // create the mirror problem by the following loop
04   for (j=1;j<=numberOfJobs;j++)
05   {
06     temp=head[j];
07     head[j]=tail[j];
08     tail[j]=temp;
09   }
10   Call Carlier's method to solve the mirror problem;
11   Set the makespan to the solution provided by the Carlier's method;
12   //modify the starting times returned by the the Carlier's method by the following loop
13   for (j=1;j<=numberOfJobs;j++)
14     start[j]=makespan-start[j]-dur[j];
15   return the starting times;
16 } //end

```

Figure 10. The pseudocode of the mirror Carlier method.

We have observed that, depending on the instance structure, the mirror problem of some OMPs is easier to solve than the original. Consequently, the RIJD method randomly applies either the original Carlier or the mirror Carlier method at each step. Solutions from the mirror Carlier method need small adjustments to represent a valid solution to the original problem, achieved by mirroring job start times relative to the makespan, as detailed in lines 13 and 14 of the pseudocode.

It is worth highlighting that the RIJD can be viewed as a general algorithm applicable to other variants of the PFSP or other scheduling problems. The key requirement for its applicability is the development of an effective adaptation for initial solution construction when triangular construction is not readily applicable. Additionally, an efficient decomposition method is essential for dividing the original problem into subproblems and solving those problems iteratively.

Having completed the description of the RIJD and before starting to present the results of the computational experiments, we will briefly discuss the memory requirements of the RIJD and emphasize the effect of the fitness landscape on the performance of local search algorithms in general and on that of the RIJD in particular.

The memory requirements for the RIJD main module are estimated in Table 2. It is worth noting that this estimate covers only the main module, as the one-machine module

uses dynamic memory allocation, which is difficult to estimate due to its dependence on the specific instance at hand. It is worth noting that the total memory footprint for solving the largest instance in our experiments (500×20) rarely exceeded 30 MB, as observed on the machine used to run the experiments, demonstrating the method's scalability and suitability for large instances.

Table 2. Memory usage breakdown estimated in terms of number of jobs, n , and machines, m .

Variables Category	Total Memory (Bytes)
Job processing times on machines	$4nm$
Triangular construction	$16nm$
Job permutation arrays used in various components	$20n$
Completion time matrices	$12nm$
Critical path variables	$12n + 4nm$
Boolean flags indicating whether an operation is critical	nm
Total	$37nm + 32n$

The fitness landscape is induced by the neighborhood structure, and its big valley property determines the efficiency of the search. Figure 11 illustrates a fitness landscape for the instance ta21 [42] near its optimal solution, shown at point (0, 0), and Figure 12 does the same for a random solution. In both figures, the x-axis represents the swaps performed in the first half of the permutation and the y-axis shows the same for the second half.

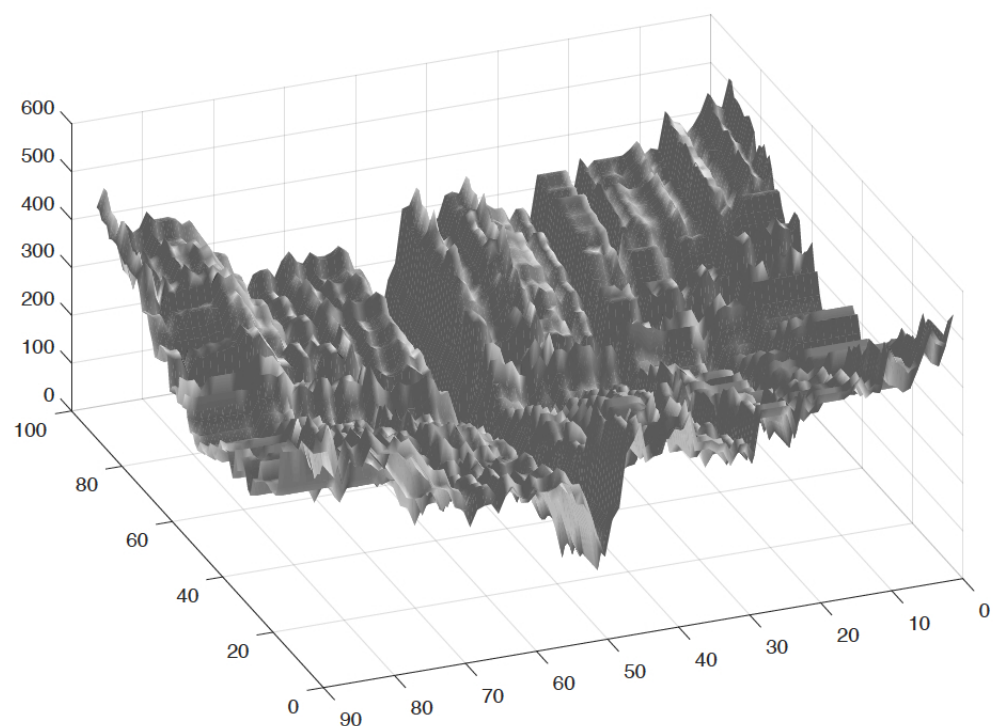


Figure 11. The schematic landscape near the optimal solution for the instance ta021 with the optimal solutions located at the point (0, 0).

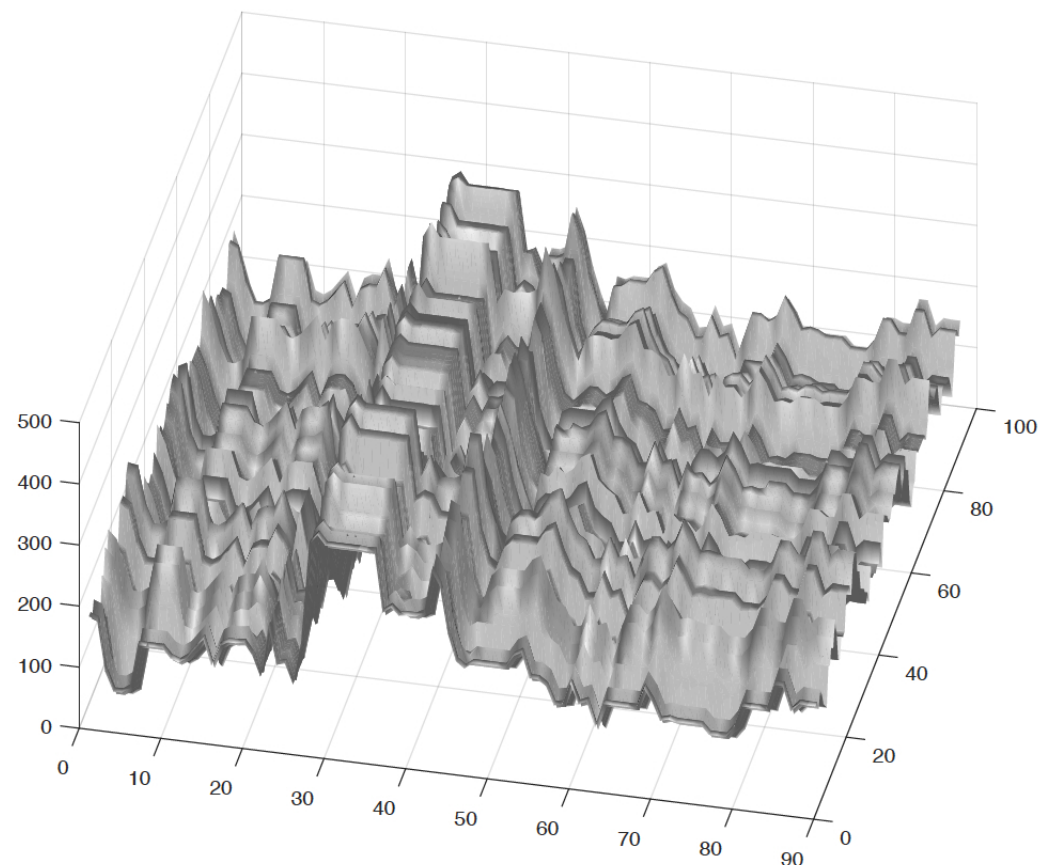


Figure 12. A schematic landscape near an initial random solution for the instance ta021 with the initial random solution located at point (0, 0).

Each half of the corresponding permutation includes 10 items; hence, there are 90, i.e., $10 \times (10 - 1)$, points on each axis. In other words, to draw each of these figures, we evaluated 8100, i.e., 90×90 , different permutations for the corresponding PFSP. Because the PFSP is a minimization problem, the z-value of each point (x, y) is shown with $\text{Max}(Z) - f(x, y)$.

This causes the smaller makespan values to be shown in higher points. These figures provide a schematic view of the fitness landscape, where each point, as a permutation, differs from its four neighboring points by the positions of only two items [43]. It should be noted that these figures are intended mainly as a qualitative visualization of the fitness landscape to illustrate the big valley property and the behavior of local search rather than as a complete representation of the PFSP search space.

5. Computational Experiments

The RIJD was implemented in C++ and compiled via a GNU GCC compiler on an Intel Core i7 PC with 2.2 Ghz speed and 8.0 GB RAM using a single thread/core. The algorithm was run on all 120 Taillard benchmark instances [42] extracted from Taillard's repository website. The number of jobs and machines in these instances ranged from 20 to 200 and from 5 to 20, respectively.

Before presenting the results of the experiments in detail, we will first show how the backward Carlier method, which was used in the RIJD, can complement the original Carlier method.

To test the efficiency of the forward-backward algorithm, a number of one-machine problem instances were generated, and, during their generation, their optimal solutions were fixed. Despite the fact that the Carlier method is able to solve some large-size

problems, the size of the generated problems varied in a small range of 25 to 100 jobs. It was seen that, in solving these particular instances, the (forward) Carlier method encountered some serious difficulty and the backward version could circumvent this difficulty. For the mirrored instances, the situation was reversed, and, while the backward version encounters some difficulty, the original method solved them quickly. In this way, we show that the forward and backward versions can complement one another, especially when they are used as two separate threads of the same process.

To define a complete one-machine problem, three values should be generated for each job i . These values are a_i , p_i , and q_i . The value of a_i shows the time at which job i becomes available and hence is ready to be processed, p_i denotes the duration or processing time needed to process the job i on the machine, and q_i is the amount of time that job i has to remain in the system. As mentioned, a_i and q_i are also known as the head and tail of job i . The value p_i is simply set to a uniform random number within 1 and 7.

To generate the instances with known optimal solutions, we define c_i as the time at which the processing of job i is completed based on their current order, i.e., $c_1 = d_1$, $c_i = \max(a_i, c_{i-1}) + p_i$ for $i = 2, 3, \dots, n$. The value of a_1 is considered to be 0, for even indices of i , a_i is set to c_{i-1} , and, for odd indices, it is set to $a_{i-1} + u$, in which u is a uniform random number between 1 and 5. Finally, for odd indices, q_i is set to $1500 - (a_i + p_i)$, and for even indices it is set to a uniform random number between 1 and 1000.

It should be noted that these synthetic instances were intentionally designed to exaggerate the asymmetry between the forward and backward search directions. They are therefore not intended to represent typical PFSP-derived one-machine problems, but to isolate and demonstrate the practical benefit of incorporating both directions within the proposed forward-backward Carlier strategy. For instance, by mirroring the problems, they become hard for the (backward) Carlier method and easy for the (forward) Carlier method. As can be seen, by forcing q_i to $1500 - (a_i + p_i)$ for the even indices of i , all these instances are forced to have the optimal solution of 1500.

Based on the setting mentioned, 16 problems were generated, and the generated instances are named inst01–inst16. The number of jobs for the first instance is 25 and each instance has 5 more jobs compared to its previous instance, with the 16th instance having 100 jobs. If the forward or backward algorithm is unable to find an optimal solution after 60 s, the algorithm is halted. It is worth noting that the forward-backward Carlier method enhances the robustness of one-machine optimization and can, therefore, serve as a practical building block for a broad range of decomposition-based scheduling algorithms.

Table 3 shows the best makespan returned as well as the time required to reach an optimal solution. As can be seen, while the backward Carlier method solved all problems in less than 20 milliseconds, the (forward) Carlier method needed dramatically longer running times and was unable to solve the instance in one minute.

To present the results of the RIJD, we will first briefly discuss the computational time complexity and run settings. For each RIJD run, in line with other methods in the literature [44], a time limit of $n.m/10$ seconds was considered, in which n and m show the number of jobs and machines, respectively. In effect, in order to provide a basic estimate of the time complexity of the RIJD, its maximum allowable run time, with respect to various problem sizes, shown as $n \times m$, is also reported in Table 4. Regarding the computational time complexity of the one-machine problem, interested readers can refer to [3].

Tables 5 and 6 show the results of running the RIJD on all 120 Taillard benchmark instances [42]. To remove the effect of the initial random seed on the performance of the algorithm, for each instance, the RIJD was restarted 10 times, with each run using a different random seed. The best and the average makespan as well as the percentage deviation from

the best-known solution (DEV%) are reported. In order to find the deviation percentage, UB was subtracted from the obtained solution and the result was divided by UB.

Table 3. Comparing solution times for the (forward) Carlier method and backward Carlier method.

Instance	No. of jobs	Best _{backward}	T _{backward} (s)	Best _{forward}	T _{forward} (s)
inst01	25	1500	0.000	1500	0.015
inst02	30	1500	0.000	1500	0.016
inst03	35	1500	0.000	1500	0.031
inst04	40	1500	0.000	1500	0.140
inst05	45	1500	0.000	1500	0.219
inst06	50	1500	0.000	1500	0.359
inst07	55	1500	0.000	1500	29.733
inst08	60	1500	0.000	1500	0.344
inst09	65	1500	0.000	1500	1.217
inst10	70	1500	0.000	1500	1.031
inst11	75	1500	0.000	1501	>60
inst12	80	1500	0.000	1501	>60
inst13	85	1500	0.000	1501	>60
inst14	90	1500	0.016	1501	>60
inst15	95	1500	0.000	1501	>60
inst16	100	1500	0.000	1501	>60

Table 4. Maximum allowed running times with respect to problem size.

<i>n</i>	<i>m</i>	Prob. Size (<i>n.m</i>)	Max. Time (s)
20	5	100	10
20	10	200	20
20	20	400	40
50	10	500	50
100	10	1000	100
100	20	2000	200
200	20	4000	400

Table 5. Comparison with the NEGA_{VNS} algorithm [44] and the NEH heuristic—Part I. Bold values indicate cases in which the RIJD has obtained an equal or better result.

Instance	n	m	LB	UB	RIJD			NEH		NEGA _{VNS}	
					%DEV _{best}	%DEV _{avg}	T _{best}	%DEV	%DEV _{best}	%DEV _{avg}	T _{best}
ta001	20	5	1278	1278	0.00	0.00	0.03	0.63	0.00	0.00	1
ta002	20	5	1359	1359	0.00	0.01	0.44	0.44	0.00	0.00	2
ta003	20	5	1081	1081	0.00	0.00	0.04	7.22	0.00	0.00	2
ta004	20	5	1293	1293	0.00	0.00	0.26	2.47	0.00	0.00	2
ta005	20	5	1235	1235	0.00	0.01	0.29	5.67	0.00	0.00	2
ta006	20	5	1195	1195	0.00	0.00	0.01	2.76	0.00	0.00	3
ta007	20	5	1239	1239	0.00	0.00	0.00	3.15	0.00	0.00	3
ta008	20	5	1206	1206	0.00	0.00	0.02	1.41	0.00	0.00	2
ta009	20	5	1230	1230	0.00	0.00	0.07	4.96	0.00	0.00	1
ta010	20	5	1108	1108	0.00	0.00	0.00	3.88	0.00	0.00	4
ta011	20	10	1582	1582	0.00	0.03	9.74	6.19	0.00	0.00	10
ta012	20	10	1659	1659	0.00	0.07	0.33	4.22	0.00	0.02	9
ta013	20	10	1496	1496	0.00	0.12	0.41	4.08	0.00	0.00	12
ta014	20	10	1377	1377	0.00	0.06	0.57	4.50	0.00	0.05	17
ta015	20	10	1419	1419	0.00	0.13	4.11	5.85	0.00	0.00	11
ta016	20	10	1397	1397	0.00	0.01	1.16	4.01	0.00	0.00	15

Table 5. Cont.

Instance	n	m	LB	UB	RIJD		NEH		NEGA _{VNS}		
					%DEV _{best}	%DEV _{avg}	T _{best}	%DEV	%DEV _{best}	%DEV _{avg}	T _{best}
ta017	20	10	1484	1484	0.00	0.00	0.58	5.26	0.00	0.00	11
ta018	20	10	1538	1538	0.39	0.47	3.68	4.62	0.00	0.00	10
ta019	20	10	1593	1593	0.00	0.09	0.03	3.39	0.00	0.00	13
ta020	20	10	1591	1591	0.00	0.23	5.72	3.90	0.00	0.00	14
ta021	20	20	2297	2297	0.04	0.16	16.69	4.92	0.00	0.00	26
ta022	20	20	2099	2099	0.00	0.03	3.82	2.43	0.00	0.01	33
ta023	20	20	2326	2326	0.13	0.20	8.27	3.65	0.00	0.02	32
ta024	20	20	2223	2223	0.00	0.04	4.35	1.75	0.00	0.00	22
ta025	20	20	2291	2291	0.13	0.19	0.33	4.63	0.00	0.04	21
ta026	20	20	2226	2226	0.00	0.18	31.58	5.53	0.00	0.03	35
ta027	20	20	2273	2273	0.00	0.15	13.71	3.92	0.00	0.07	36
ta028	20	20	2200	2200	0.00	0.15	25.69	2.23	0.00	0.00	25
ta029	20	20	2237	2237	0.00	0.09	5.95	3.71	0.00	0.00	23
ta030	20	20	2178	2178	0.05	0.14	1.73	4.55	0.00	0.02	39
ta031	50	5	2724	2724	0.00	0.00	0.04	0.33	0.00	0.00	6
ta032	50	5	2834	2834	0.07	0.13	11.49	0.32	0.00	0.00	5
ta033	50	5	2621	2621	0.00	0.00	0.01	0.72	0.00	0.00	1
ta034	50	5	2751	2751	0.00	0.00	0.11	1.13	0.00	0.00	11
ta035	50	5	2863	2863	0.00	0.00	0.08	0.17	0.00	0.00	8
ta036	50	5	2829	2829	0.00	0.00	0.03	0.74	0.00	0.00	5
ta037	50	5	2725	2725	0.00	0.00	0.05	1.21	0.00	0.00	7
ta038	50	5	2683	2683	0.00	0.00	0.00	1.42	0.00	0.00	6
ta039	50	5	2552	2552	0.00	0.03	1.44	0.94	0.00	0.00	12
ta040	50	5	2782	2782	0.00	0.00	0.00	0.29	0.00	0.00	21
ta041	50	10	2991	2991	1.17	1.44	25.92	4.81	1.00	1.03	38
ta042	50	10	2867	2867	1.53	1.64	3.86	5.76	1.22	1.28	41
ta043	50	10	2839	2839	1.13	1.22	0.92	5.18	1.13	1.14	36
ta044	50	10	3063	3063	0.00	0.11	35.64	4.41	0.23	0.28	29
ta045	50	10	2976	2976	1.08	1.33	11.72	6.18	0.74	0.81	25
ta046	50	10	3006	3006	0.43	0.88	5.99	5.72	0.60	0.68	32
ta047	50	10	3093	3093	1.10	1.32	12.18	5.95	0.94	0.98	19
ta048	50	10	3037	3037	0.07	0.25	41.99	2.83	0.86	0.93	29
ta049	50	10	2897	2897	0.45	0.80	9.10	3.62	0.59	0.65	39
ta050	50	10	3065	3065	1.11	1.27	8.46	6.26	0.36	0.44	35
ta051	50	20	3771	3850	1.61	1.86	10.95	6.03	0.62	0.77	22
ta052	50	20	3668	3704	1.40	1.90	71.53	5.86	0.81	1.02	87
ta053	50	20	3591	3640	1.95	2.45	42.84	7.88	1.32	1.39	56
ta054	50	20	3635	3720	1.64	1.92	75.66	6.69	1.05	1.14	39
ta055	50	20	3553	3610	1.69	2.06	34.28	6.23	0.94	1.03	48
ta056	50	20	3667	3681	1.58	1.90	47.16	6.33	0.98	1.07	74
ta057	50	20	3672	3704	1.86	2.16	63.07	6.70	0.65	0.79	42
ta058	50	20	3627	3691	2.25	2.41	41.75	6.69	1.06	1.18	28
ta059	50	20	3645	3743	1.71	1.93	97.77	5.58	0.96	1.10	90
ta060	50	20	3696	3756	1.89	2.25	68.23	8.60	1.20	1.35	64

In the table, the tightest available lower and upper bounds of each instance in the literature are provided in the third and fourth columns, respectively. For the instances where the UB (upper bound) is strictly greater than the LB (lower bound), the best available solution in the literature has no guarantee of optimality.

In addition to the results produced by the RIJD, those produced by the NEGA_{VNS} algorithm [44] are presented. Moreover, the results produced by the famous NEH algorithm [10]

are also presented. The NEH, a deterministic method, was run on an old computer, but the NEGA_{VNS} was coded in C++ and run on a Pentium with 2.4 GHz 1.0 GB of RAM [44].

Table 6. Comparison with the NEGA_{VNS} algorithm [44] and the NEH heuristic—Part II. Bold values indicate cases in which the RIJD has obtained an equal or better result.

Instance	n	m	LB	UB	RIJD			NEH		NEGA _{VNS}	
					%DEV _{best}	%DEV _{avg}	T _{best}	%DEV	%DEV _{best}	%DEV _{avg}	T _{best}
ta061	100	5	5493	5493	0.00	0.00	0.00	0.47	0.00	0.00	34
ta062	100	5	5268	5268	0.00	0.00	0.19	1.52	0.00	0.00	26
ta063	100	5	5175	5175	0.00	0.00	0.37	0.85	0.00	0.00	36
ta064	100	5	5014	5014	0.06	0.07	0.83	0.18	0.00	0.00	33
ta065	100	5	5250	5250	0.00	0.01	0.20	0.30	0.00	0.00	12
ta066	100	5	5135	5135	0.00	0.00	0.02	0.08	0.00	0.00	42
ta067	100	5	5246	5246	0.00	0.00	0.06	0.25	0.00	0.00	50
ta068	100	5	5094	5094	0.00	0.00	0.41	0.51	0.00	0.00	31
ta069	100	5	5448	5448	0.00	0.00	0.70	0.75	0.00	0.00	25
ta070	100	5	5322	5322	0.04	0.08	11.83	0.36	0.00	0.00	19
ta071	100	10	5770	5770	0.19	0.24	67.66	1.32	0.00	0.04	49
ta072	100	10	5349	5349	0.22	0.27	17.38	1.94	0.17	0.23	78
ta073	100	10	5676	5676	0.02	0.05	41.62	2.61	0.00	0.09	65
ta074	100	10	5781	5781	0.42	0.58	26.86	2.56	0.19	0.23	22
ta075	100	10	5467	5467	0.57	0.69	44.04	3.88	0.00	0.06	81
ta076	100	10	5303	5303	0.09	0.09	0.45	1.36	0.15	0.20	72
ta077	100	10	5595	5595	0.09	0.15	31.26	1.95	0.18	0.22	54
ta078	100	10	5617	5617	0.53	0.70	28.56	2.55	0.00	0.05	64
ta079	100	10	5871	5871	0.39	0.58	68.05	2.74	0.10	0.19	29
ta080	100	10	5845	5845	0.26	0.55	11.59	1.25	0.00	0.09	73
ta081	100	20	6106	6202	2.63	2.92	150.56	5.47	1.63	1.69	85
ta082	100	20	6183	6183	1.86	2.22	14.59	5.50	1.34	1.45	75
ta083	100	20	6252	6271	1.90	2.10	78.48	5.87	1.28	1.32	145
ta084	100	20	6254	6269	1.63	1.99	99.54	4.59	1.45	1.49	129
ta085	100	20	6262	6314	2.34	2.63	13.57	6.03	1.49	1.57	163
ta086	100	20	6302	6364	1.93	2.26	74.69	4.71	1.40	1.50	108
ta087	100	20	6184	6268	1.96	2.45	154.74	5.81	1.02	1.10	94
ta088	100	20	6315	6401	2.67	2.89	177.72	5.28	1.27	1.49	112
ta089	100	20	6204	6275	2.34	2.66	119.84	6.41	1.08	1.15	169
ta090	100	20	6404	6434	1.68	2.06	96.41	3.78	1.12	1.26	147
ta091	200	10	10,862	10,862	0.21	0.21	5.82	0.74	0.21	0.24	89
ta092	200	10	10,480	10,480	0.28	0.43	169.12	2.25	0.14	0.19	125
ta093	200	10	10,922	10,922	0.39	0.54	49.18	0.94	0.17	0.21	169
ta094	200	10	10,889	10,889	0.04	0.11	26.32	1.54	0.00	0.04	158
ta095	200	10	10,524	10,524	0.07	0.11	108.72	1.15	0.00	0.03	192
ta096	200	10	10,329	10,329	0.14	0.25	141.80	1.25	0.16	0.21	91
ta097	200	10	10,854	10,854	0.29	0.43	138.95	1.24	0.11	0.17	124
ta098	200	10	10,730	10,730	0.21	0.39	125.33	0.92	0.10	0.15	112
ta099	200	10	10,438	10,438	0.13	0.24	173.22	1.30	0.12	0.19	138
ta100	200	10	10,675	10,675	0.09	0.40	41.14	1.24	0.08	0.14	147
ta101	200	20	11,152	11,195	1.77	1.96	169.42	3.56	1.50	1.52	222
ta102	200	20	11,143	11,203	2.45	2.74	318.97	4.21	1.32	1.47	268
ta103	200	20	11,281	11,281	2.36	2.50	329.46	5.06	1.41	1.45	385
ta104	200	20	11,275	11,275	1.70	2.06	38.72	4.68	1.44	1.49	154
ta105	200	20	11,259	11,259	1.59	1.79	174.11	3.78	1.05	1.06	300
ta106	200	20	11,176	11,176	2.20	2.39	357.15	4.05	0.92	1.01	254
ta107	200	20	11,337	11,360	1.53	2.03	372.50	4.16	0.99	1.11	269
ta108	200	20	11,301	11,334	1.97	2.22	382.17	5.11	0.95	1.03	311
ta109	200	20	11,145	11,192	2.21	2.49	367.94	4.30	1.10	1.22	326
ta110	200	20	11,284	11,288	2.04	2.26	103.87	5.15	1.00	1.14	228
ta111	500	20	26,040	26,059	1.23	1.43	749.27	2.34	0.65	0.73	311
ta112	500	20	26,500	26,520	1.41	1.54	246.95	2.68	0.63	0.77	552
ta113	500	20	26,371	26,371	1.13	1.36	169.82	1.81	0.57	0.71	448
ta114	500	20	26,456	26,456	0.85	1.08	388.00	2.26	0.49	0.54	269
ta115	500	20	26,334	26,334	0.84	1.08	602.80	1.49	0.79	0.82	396
ta116	500	20	26,469	26,477	0.85	1.07	1000.00	1.95	0.40	0.49	682
ta117	500	20	26,389	26,389	0.78	0.89	1000.00	1.55	1.03	1.12	559
ta118	500	20	26,560	26,560	1.08	1.32	733.69	2.18	0.57	0.61	814
ta119	500	20	26,005	26,005	1.02	1.34	244.26	2.41	0.55	0.61	592
ta120	500	20	26,457	26,457	0.91	1.05	835.95	1.99	0.58	0.67	611

Tables 5 and 6 show that, in 89 out of 120 instances, the average time taken to find the best solution by the RIJD was less than or equal to that taken by NEGA_{VNS}. In Table 2, wherever the RIJD has obtained an equal or better result compared to NEGA_{VNS}, the corresponding values are written in bold. This means the best obtained results are shown in bold for 51 of the 120 instances. Moreover, in 31 out of 120 instances, the average percentage deviation from the best available solution in the literature is smaller than or equal to that of NEGA_{VNS}.

The RIJD performance was compared with that of several methods, including the more recent Refining Decomposition-Based Integrated Search (RDIS) of [7], the Re-Blocking Adjustable Memetic Procedure (RAMP) of [38], NEGA_{VNS} of [44], the particle swarm optimization algorithm PSO_{VNS} of [45], the simulated annealing algorithm SAOP of [46], two ant colony optimization algorithms PACO and M-MMAS of [47], the iterated local search (ILS) of [21], and the hybrid genetic algorithms HGA_RMA of [48], as shown in Table 7. As can be seen, on several instance groups, the RIJD outperformed several of the other methods from the literature.

The key limitation of the RIJD can be considered to be the fact that it achieves a higher %DEV_{avg} on certain instance groups, particularly larger-sized instances. This can be attributed to the limited diversification of components such as the triangular construction method and the one-machine problem. Some potential remedies for these issues are discussed in Section 6 as future research directions. Furthermore, some instances may require longer waiting times for the one-machine problem to be solved, introducing computational overhead. While this may be acceptable in asynchronous and parallel processing environments and methods [49], it remains a challenge in sequential computing settings.

Table 7. Comparison of the average percent deviation from the best-known solution with that of other methods from the literature.

Instances	Size		SAOP	ILS	M-MMAS	PACO	HGA_RMA	PSO _{VNS}	NEGA _{VNS}	RAMP	RDIS	RIJD
	n	m										
ta001–010	20	5	1.05	0.33	0.04	0.18	0.04	0.03	0.00	0.00	0.00	0.00
ta011–020	20	10	2.60	0.52	0.07	0.24	0.02	0.02	0.01	0.03	0.03	0.12
ta021–030	20	20	2.06	0.28	0.06	0.18	0.05	0.05	0.02	0.04	0.04	0.13
ta031–040	50	5	0.34	0.18	0.02	0.05	0.00	0.00	0.00	0.00	0.01	0.02
ta041–050	50	10	3.50	1.45	1.08	0.81	0.72	0.57	0.82	0.37	0.95	1.03
ta051–060	50	20	4.66	2.05	1.93	1.41	0.99	1.36	1.08	0.61	1.88	2.08
ta061–070	100	5	0.30	0.16	0.02	0.02	0.01	0.00	0.00	0.00	0.01	0.02
ta071–080	100	10	1.34	0.64	0.39	0.29	0.16	0.18	0.14	0.06	0.38	0.39
ta081–090	100	20	4.49	2.42	2.42	1.93	1.30	1.45	1.40	1.76	2.35	2.42
ta091–100	200	10	0.94	0.50	0.30	0.23	0.14	0.18	0.16	0.15	0.32	0.31
ta101–110	200	20	3.67	2.07	2.15	1.82	1.26	1.35	1.25	2.00	2.30	2.24
ta111–120	500	20	2.20	1.20	1.02	0.85	0.69	–	0.71	1.20	1.32	1.22

It is worth outlining and contrasting the performance of recent deep learning-based approaches, namely, those of [31,32]. In [31], the proposed PFSPNet_NEH is evaluated on Taillard instances of up to 200 jobs and 20 machines, reporting a mean average percent deviation of 0.2808 with a variance of 0.2432. Results for 500 × 20 instances are not provided, likely due to limitations in handling larger problem instances. In [32], the gap from the NEH algorithm results (rather than the best-known solution) is used as a metric, and it ranges from 9.4% to 12.8%, slightly outperforming [31] on larger instances. It should be noted that both methods utilize both CPU and advanced GPU hardware and involve a training stage—taking 200 h in the case of PFSPNet—which is not included in the reported computation time. For these reasons, the reported results and running times are not directly comparable, as the RIJD requires no training stage, whereas deep learning approaches incur an additional offline training cost that is excluded from the online solution times reported here.

Finally, for a more in-depth analysis and a fair comparison, different variants of the RIJD were implemented and evaluated in the exact same running environment. These variants were run on the same CPU, each utilized a single CPU core, and each had an identical maximum allowed running time. They were designed by deactivating certain components of the RIJD procedure. Essentially, in RIJD_{TRI}, the local search is disabled, in RIJD_{LS}, the triangular construction is deactivated and replaced by a random permutation

generator, and, in RIJD_{OMP}, both the triangular construction and the local search are disabled, leaving only the one-machine problem improvement module active.

The results, presented in Table 8, for instances of different sizes, show that the RIJD outperformed the other three variants in terms of the average percent deviation from the best-known solution (%DEV_{avg}), with RIJD_{LS} ranking second. This highlights not only the high contribution and effectiveness of the local search and one-machine problem component, but also the synergistic effect of all different modules within the RIJD.

Table 8. Comparing the performance of the four different variants of the RIJD.

Instance	Size		RIJD _{TRI}		RIJD _{LS}		RIJD _{OMP}		RIJD	
	n	m	%DEV _{best}	%DEV _{avg}	%DEV _{best}	%DEV _{avg}	%DEV _{best}	%DEV _{avg}	%DEV _{best}	%DEV _{avg}
ta001	20	5	3.44	4.84	0.00	0.00	4.77	6.06	0.00	0.00
ta011	20	10	6.83	11.37	0.00	0.04	12.52	14.10	0.00	0.05
ta021	20	20	7.27	8.31	0.04	0.28	9.40	10.21	0.09	0.27
ta031	50	5	2.09	2.77	0.00	0.00	2.94	4.42	0.00	0.00
ta041	50	10	11.70	12.47	1.14	1.58	15.01	16.41	1.14	1.46
ta051	50	20	11.66	12.55	1.69	2.20	16.00	16.76	1.74	2.08
ta061	100	5	1.38	1.80	0.00	0.00	3.19	3.72	0.00	0.00
ta071	100	10	9.74	10.05	0.05	0.28	11.04	11.49	0.23	0.28
ta081	100	20	13.71	14.64	2.97	3.23	17.77	18.15	2.31	2.93
ta091	200	10	5.84	7.22	0.21	0.30	6.51	7.03	0.16	0.23
ta101	200	20	14.00	14.75	2.09	2.32	14.41	15.14	1.48	1.94
ta111	500	20	11.54	12.81	1.39	1.74	11.69	12.04	1.30	1.53
Average		8.27	9.46	0.80	1.00	10.44	11.30	0.70	0.90	

Further, a Friedman test on relative ranks with respect to the %DEV_{avg} of the RIJD variants, presented in Table 9, was performed. In the case of tied %DEV_{avg} values, the average of the ranks was selected. The Friedman test showed a statistically significant difference of the ranks, with $\chi^2(3) = 32.379$ and $p < 0.001$ ($p = 4.353 \times 10^{-7}$). Interested readers can refer to [50] for various statistical tests used to compare metaheuristic algorithms.

Table 9. The ranks of RIJD_{TRI}, RIJD_{LS}, RIJD_{OMP}, and the RIJD based on average deviation from the best-known solution.

Instance	RIJD _{TRI}	RIJD _{LS}	RIJD _{OMP}	RIJD
ta001	3	1.5	4	1.5
ta011	3	1	4	2
ta021	3	2	4	1
ta031	3	1.5	4	1.5
ta041	3	2	4	1
ta051	3	2	4	1
ta061	3	1.5	4	1.5
ta071	3	1.5	4	1.5
ta081	3	2	4	1
ta091	4	2	3	1
ta101	3	2	4	1
ta111	4	2	3	1

6. Concluding Remarks

To achieve further advances in metaheuristics, the systematic understanding of metaheuristics seems to be of paramount importance [23]. Despite the fact that research in metaheuristics is mostly experimental, the main difficulty in the study of metaheuristics is the complexity of measuring the effect of the different components of a search on its efficiency. This stems from the fact that the effect of a component can be determined only in the context in which it is used and not independently of the other components used in the same context.

The results indicating that our newly developed Recurrent Intelligent Job Displacer can obtain solutions with an accuracy within less than one percent (0.83% across all benchmark instances) of the best solution in the literature in a matter of seconds do not stem from one factor but three integrated components. Additionally, deeper analysis of the RIJD modules, reported in Table 8, and the RIJD outperforming other variants (with average $\%DEV_{best} = 0.70$ and $\%DEV_{avg} = 0.90$) given the same maximum run time and computational environment could integrate the synergistic combination of its components. These components include (i) the construction method, (ii) the successive displacement mechanism, and (iii) the local search.

In the RIJD, the probabilistic ingredient of the construction method does not focus indiscriminately on selecting elements for augmentation of the solution, but uses a biased random sampling which favors high-quality permutations. In this way, the construction method, without concentrating on a specific range of the search space, can create high-quality solutions (for the notion of reference sets to fill this with additional context see, e.g., ref. [51]).

A vital ingredient of any local search is its neighborhood scheme, and the employed local search mainly gains its effectiveness through such a scheme. Exploiting the basic structure of the PFSP through an effective neighborhood and using an effective evaluation function for searching such a neighborhood are two features which have contributed to the efficiency of the employed local search.

Four major directions are envisaged for further enhancing the overall performance of the RIJD, particularly its effectiveness on increasingly large PFSP instances: First, it can be modified for a multi-processing environment. For this purpose, and since the construction method can produce different initial starting points, each starting point can be assigned to a separate processor to improve. Interested readers are referred to [49] for a discussion of various approaches to applying parallelism for the PFSP.

Second, the probabilistic ingredient of the construction method can be made more sophisticated in order to create further diversification. Currently, this probabilistic ingredient only works based on a random permutation and fixes the positions of this permutation based on the sorted values of processing times integration. In a more sophisticated version, the reversal or perturbation of various parts of the original permutation, for instance, could create another starting permutation instead of a random permutation.

Third, instead of replacing the entire current permutation, the feedback received from the one-machine method can replace parts of the current permutation. In this case, the non-guaranteed optimal solution for all machines is not completely replaced with the guaranteed optimal solution presented for one machine and is perturbed just slightly. In other words, through this simple random perturbation step, the job permutation across all machines, representing the solution to the PFSP, is only partially influenced by the feedback from the one-machine problem.

Fourth, the employed local search can be further improved to escape local optimality through employing a biased randomization favoring neighbors with higher quality. Currently, each solution is transformed to one of its neighbors until the neighbor to which a solution is transformed no longer has better quality. In an improved version, whenever no better neighbor is found, the algorithm could use a biased randomization favoring neighbors with higher quality, and, instead of being terminated, it could continue with such a neighbor. In this way, better solutions may be obtained, at the cost of additional computation time.

Author Contributions: Conceptualization, M.A. and S.V.; methodology, M.A.; software, M.A.; validation, M.A. and W.G.; investigation, M.A. and S.V.; writing—original draft preparation, M.A.; writing—review and editing, M.A., S.V., W.G., A.A., and R.O.; visualization, M.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Márquez, C.R.H.; Ribeiro, C.C. Shop scheduling in manufacturing environments: A review. *Int. Trans. Oper. Res.* **2022**, *29*, 3237–3293. <https://doi.org/10.1111/itor.13108>.
2. Ying, K.C.; Pourhejazy, P.; Cheng, C.Y.; Syu, R.S. Supply chain-oriented permutation flowshop scheduling considering flexible assembly and setup times. *Int. J. Prod. Res.* **2023**, *61*, 258–281. <https://doi.org/10.1080/00207543.2020.1842938>.
3. Carlier, J. The one-machine sequencing problem. *Eur. J. Oper. Res.* **1982**, *11*, 42–47. [https://doi.org/10.1016/S0377-2217\(82\)80007-6](https://doi.org/10.1016/S0377-2217(82)80007-6).
4. Adams, J.; Balas, E.; Zawack, D. The shifting bottleneck procedure for job shop scheduling. *Manag. Sci.* **1988**, *34*, 391–401. <https://doi.org/10.1287/mnsc.34.3.391>.
5. Potts, C.N.; Shmoys, D.B.; Williamson, D.P. Permutation vs. non-permutation flow shop schedules. *Oper. Res. Lett.* **1991**, *10*, 281–284. [https://doi.org/10.1016/0167-6377\(91\)90014-G](https://doi.org/10.1016/0167-6377(91)90014-G).
6. Framinan, J.M.; Gupta, J.N.D.; Leisten, R. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *J. Oper. Res. Soc.* **2004**, *55*, 1243–1255. <https://doi.org/10.1057/palgrave.jors.2601784>.
7. Amirghasemi, M. An effective decomposition-based stochastic algorithm for solving the permutation flow-shop scheduling problem. *Algorithms* **2021**, *14*, 112. <https://doi.org/10.3390/a14040112>.
8. Reeves, C. A genetic algorithm for flowshop sequencing. *Comput. Oper. Res.* **1995**, *22*, 5–13. [https://doi.org/10.1016/0305-0548\(93\)E0014-K](https://doi.org/10.1016/0305-0548(93)E0014-K).
9. Taillard, E. Robust taboo search for the quadratic assignment problem. *Parallel Comput.* **1991**, *17*, 443–455. [https://doi.org/10.1016/S0167-8191\(05\)80147-4](https://doi.org/10.1016/S0167-8191(05)80147-4).
10. Nawaz, M.; Enscore, E.; Ham, I. A heuristic algorithm for the m-machine n-job flow-shop sequencing problem. *Omega* **1983**, *11*, 91–95. [https://doi.org/10.1016/0305-0483\(83\)90088-9](https://doi.org/10.1016/0305-0483(83)90088-9).
11. Taillard, E. Some efficient heuristic methods for the flow shop sequencing problem. *Eur. J. Oper. Res.* **1990**, *47*, 65–74. [https://doi.org/10.1016/0377-2217\(90\)90090-X](https://doi.org/10.1016/0377-2217(90)90090-X).
12. Fernandez-Viagas, V.; Molina-Pariente, J.M.; Framinan, J.M. Generalised accelerations for insertion-based heuristics in permutation flowshop scheduling. *Eur. J. Oper. Res.* **2020**, *282*, 858–872. <https://doi.org/10.1016/j.ejor.2019.10.017>.
13. Nowicki, E.; Smutnicki, C. A fast tabu search algorithm for the permutation flow-shop problem. *Eur. J. Oper. Res.* **1996**, *91*, 160–175. [https://doi.org/10.1016/0377-2217\(95\)00037-2](https://doi.org/10.1016/0377-2217(95)00037-2).
14. Caserta, M.; Voß, S. Metaheuristics: Intelligent Problem Solving. In *Matheuristics: Hybridizing Metaheuristics and Mathematical Programming*; Maniezzo, V., Stützle, T., Voß, S., Eds.; Springer US: Boston, MA, USA, 2010; pp. 1–38. https://doi.org/10.1007/978-1-4419-1306-7_1.
15. Garey, M.; Johnson, D.; Sethi, R. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.* **1976**, *1*, 117–129.
16. King, J.R.; Spachis, A.S. Heuristics for flow-shop scheduling. *Int. J. Prod. Res.* **1980**, *18*, 345–357. <https://doi.org/10.1080/00207548008919673>.
17. Dudek, R.A.; Panwalkar, S.S.; Smith, M.L. The lessons of flowshop scheduling research. *Oper. Res.* **1992**, *40*, 7–13. <https://doi.org/10.1287/opre.40.1.7>.
18. Blazewicz, J.; Domschke, W.; Pesch, E. The job shop scheduling problem: Conventional and new solution techniques. *Eur. J. Oper. Res.* **1996**, *93*, 1–33. [https://doi.org/10.1016/0377-2217\(95\)00362-2](https://doi.org/10.1016/0377-2217(95)00362-2).
19. Gupta, J.N.; Stafford, E.F. Flowshop scheduling research after five decades. *Eur. J. Oper. Res.* **2006**, *169*, 699–711. <https://doi.org/10.1016/j.ejor.2005.02.001>.
20. Ruiz, R.; Maroto, C. A comprehensive review and evaluation of permutation flowshop heuristics. *Eur. J. Oper. Res.* **2005**, *165*, 479–494. <https://doi.org/10.1016/j.ejor.2004.04.017>.
21. Stützle, T. *Applying Iterated Local Search to the Permutation Flow Shop Problem*; Technical Report, AIDA-98-04; FG Intellektik, TU Darmstadt: Darmstadt, Germany, **1998**.

22. Fernandez-Viagas, V.; Ruiz, R.; Framinan, J.M. A new vision of approximate methods for the permutation flowshop to minimise makespan: State-of-the-art and computational evaluation. *Eur. J. Oper. Res.* **2017**, *257*, 707–721. <https://doi.org/10.1016/j.ejor.2016.09.055>.
23. de Armas, J.; Lalla-Ruiz, E.; Tilahun, S.L.; Voß, S. Similarity in metaheuristics: A gentle step towards a comparison methodology. *Nat. Comput.* **2022**, *21*, 265–287. <https://doi.org/10.1007/s11047-020-09837-9>.
24. De Snoo, C.; Van Wezel, W.; Jorna, R.J. An empirical investigation of scheduling performance criteria. *J. Oper. Manag.* **2011**, *29*, 181–193. <https://doi.org/10.1016/j.jom.2010.12.006>.
25. Nowicki, E.; Smutnicki, C. Some aspects of scatter search in the flow-shop problem. *Eur. J. Oper. Res.* **2006**, *169*, 654–666. <https://doi.org/10.1016/j.ejor.2004.08.021>.
26. Lian, Z.; Gu, X.; Jiao, B. A novel particle swarm optimization algorithm for permutation flow-shop scheduling to minimize makespan. *Chaos, Solitons & Fractals* **2008**, *35*, 851–861. <https://doi.org/10.1016/j.chaos.2006.05.082>.
27. Lian, Z.; Gu, X.; Jiao, B. A similar particle swarm optimization algorithm for permutation flowshop scheduling to minimize makespan. *Appl. Math. Comput.* **2006**, *175*, 773–785. <https://doi.org/10.1016/j.amc.2005.07.042>.
28. Kuo, I.H.; Horng, S.J.; Kao, T.W.; Lin, T.L.; Lee, C.L.; Terano, T.; Pan, Y. An efficient flow-shop scheduling algorithm based on a hybrid particle swarm optimization model. *Expert Syst. Appl.* **2009**, *36*, 7027–7032. <https://doi.org/10.1016/j.eswa.2008.08.054>.
29. Taillard, E.D. *FANT: Fast Ant System*; Technical Report; Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale: Viganello, Switzerland, **1998**.
30. Liu, W.; Jin, Y.; Price, M. A new improved NEH heuristic for permutation flowshop scheduling problems. *Int. J. Prod. Econ.* **2017**, *193*, 21–30. <https://doi.org/https://doi.org/10.1016/j.ijpe.2017.06.026>.
31. Pan, Z.; Wang, L.; Wang, J.; Lu, J. Deep reinforcement learning based optimization algorithm for permutation flow-shop scheduling. *IEEE Trans. Emerg. Top. Comput. Intell.* **2023**, *7*, 983–994. <https://doi.org/10.1109/TETCI.2021.3098354>.
32. Li, L.; Liang, S.; Zhu, Z.; Ding, C.; Zha, H.; Wu, B. Learning to optimize permutation flow shop scheduling via graph-based imitation learning. *Proc. Aaai Conf. Artif. Intell.* **2024**, *38*, 20185–20193. <https://doi.org/10.1609/aaai.v38i18.29998>.
33. Ogbu, F.; Smith, D. The application of the simulated annealing algorithm to the solution of the n/m/Cmax flowshop problem. *Comput. Oper. Res.* **1990**, *17*, 243–253. [https://doi.org/10.1016/0305-0548\(90\)90001-N](https://doi.org/10.1016/0305-0548(90)90001-N).
34. Iyer, S.; Saxena, B. Improved genetic algorithm for the permutation flowshop scheduling problem. *Comput. Oper. Res.* **2004**, *31*, 593–606. [https://doi.org/10.1016/S0305-0548\(03\)00016-9](https://doi.org/10.1016/S0305-0548(03)00016-9).
35. Chang, P.C.; Huang, W.H.; Wu, J.L.; Cheng, T. A block mining and re-combination enhanced genetic algorithm for the permutation flowshop scheduling problem. *Int. J. Prod. Econ.* **2013**, *141*, 45–55. <https://doi.org/https://doi.org/10.1016/j.ijpe.2012.06.007>.
36. Rahimi-Vahed, A.R.; Mirghorbani, S.M. A multi-objective particle swarm for a flow shop scheduling problem. *J. Comb. Optim.* **2007**, *13*, 79–102. <https://doi.org/10.1007/s10878-006-9015-7>.
37. Yagmahan, B.; Yenisey, M.M. A multi-objective ant colony system algorithm for flow shop scheduling problem. *Expert Syst. Appl.* **2010**, *37*, 1361–1368. <https://doi.org/10.1016/j.eswa.2009.06.105>.
38. Amirghasemi, M.; Zamani, R. An effective evolutionary hybrid for solving the permutation flowshop scheduling problem. *Evol. Comput.* **2017**, *25*, 87–111. https://doi.org/10.1162/evco_a_00162.
39. de Fátima Moraes, M.; Ribeiro, M.H.D.M.; da Silva, R.G.; Mariani, V.C.; dos Santos Coelho, L. Discrete differential evolution metaheuristics for permutation flow shop scheduling problems. *Comput. Ind. Eng.* **2022**, *166*, 107956. <https://doi.org/10.1016/j.cie.2022.107956>.
40. Gmys, J.; Mezma, M.; Melab, N.; Tuytens, D. A computationally efficient branch-and-bound algorithm for the permutation flow-shop scheduling problem. *Eur. J. Oper. Res.* **2020**, *284*, 814–833. <https://doi.org/10.1016/j.ejor.2020.01.039>.
41. Schrage, L. Solving resource-constrained network problems by implicit enumeration-preemptive case. *Oper. Res.* **1972**, *20*, 668–677. <https://doi.org/10.1287/opre.20.3.668>.
42. Taillard, E. Benchmarks for basic scheduling problems. *Eur. J. Oper. Res.* **1993**, *64*, 278–285. [https://doi.org/10.1016/0377-2217\(93\)90182-M](https://doi.org/10.1016/0377-2217(93)90182-M).
43. Cartwright, H.M.; Mott, G.F. Looking around: Using clues from the data space to guide genetic algorithm searches. In Proceedings of the Fourth International Conference on Genetic Algorithms, San Diego, CA, USA, 13–16 July 1991; pp. 108–114.
44. Zobolas, G.; Tarantilis, C.; Ioannou, G. Minimizing makespan in permutation flow shop scheduling problems using a hybrid metaheuristic algorithm. *Comput. Oper. Res.* **2009**, *36*, 1249–1267. <https://doi.org/10.1016/j.cor.2008.01.007>.
45. Tasgetiren, M.F.; Liang, Y.C.; Sevcli, M.; Gencyilmaz, G. A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem. *Eur. J. Oper. Res.* **2007**, *177*, 1930–1947. <https://doi.org/10.1016/j.ejor.2005.12.024>.
46. Osman, I.H.; Potts, C.N. Simulated annealing for permutation flow-shop scheduling. *Omega* **1989**, *17*, 551–557. [https://doi.org/10.1016/0305-0483\(89\)90059-5](https://doi.org/10.1016/0305-0483(89)90059-5).
47. Rajendran, C.; Ziegler, H. Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs. *Eur. J. Oper. Res.* **2004**, *155*, 426–438. [https://doi.org/10.1016/S0377-2217\(02\)00908-6](https://doi.org/10.1016/S0377-2217(02)00908-6).

48. Ruiz, R.; Maroto, C.; Alcaraz, J. Two new robust genetic algorithms for the flowshop scheduling problem. *Omega* **2006**, *34*, 461–476. <https://doi.org/10.1016/j.omega.2004.12.006>.
49. Amirghasemi, M. An effective parallel evolutionary metaheuristic with its application to three optimization problems. *Appl. Intell.* **2022**, *53*, 5887–5909. <https://doi.org/10.1007/s10489-022-03599-w>.
50. Derrac, J.; García, S.; Molina, D.; Herrera, F. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm Evol. Comput.* **2011**, *1*, 3–18. <https://doi.org/10.1016/j.swevo.2011.02.002>.
51. Palk, M.; Voß, S.; Glover, F. The next stage of evolutionary computation in the era of agentic generative AI. In *Proceedings of the Applications of Evolutionary Computation*; García-Sánchez, P., Díaz Álvarez, J., Murphy, A., Eds.; Springer: Cham, Switzerland, **2026**; pp. 306–326. https://doi.org/10.1007/978-3-032-23607-4_19.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.