Pagnozzi, Federico; Stützle, Thomas G.

Article

Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems with additional constraints

Operations Research Perspectives

Provided in Cooperation with:
Elsevier

Suggested Citation: Pagnozzi, Federico; Stützle, Thomas G. (2021) : Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems with additional constraints, Operations Research Perspectives, ISSN 2214-7160, Elsevier, Amsterdam, Vol. 8, pp. 1-17,
https://doi.org/10.1016/j.orp.2021.100180

This Version is available at:
https://hdl.handle.net/10419/246440

Operations Research Perspectives 8 (2021) 100180

Contents lists available at ScienceDirect

Operations Research Perspectives

journal homepage: www.elsevier.com/locate/orp

Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems with additional constraints

Federico Pagnozzi *, Thomas Stützle

IRIDIA, Université Libre de Bruxelles (ULB), 50, Av. F. Roosevelt, CP 194/6, Brussels B-1050, Belgium

ARTICLE INFO

Keywords:
Combinatorial optimization
Stochastic local search algorithms
Automatic algorithm design
Permutation flowshop problem
No-idle
Sequence dependent setup times

ABSTRACT

Automatic design of stochastic local search algorithms has been shown to be very effective in generating algorithms for the permutation flowshop problem for the most studied objectives including makespan, flowtime and total tardiness. The automatic design system uses a configuration tool to combine algorithmic components following a set of rules defined as a context-free grammar. In this paper we use the same system to tackle two of the most studied additional constraints for these objectives: sequence dependent setup times and no-idle constraint. Additional components have been added to adapt the system to the new problems while keeping intact the grammar structure and the experimental setup. The experiments show that the generated algorithms outperform the state of the art in each case.

1. Introduction

Automatic algorithm design (*AAD*) has shown to be able to produce state-of-the-art algorithms for the permutation flowshop problem [43]. The method proposed by Pagnozzi and Stützle [43] is based on using an automatic configuration tool to assemble algorithmic components following rules defined as a context-free grammar. The algorithmic components were implemented in the EMILI framework, a flexible framework that allows the generation of stochastic local search (SLS) algorithms. In particular, the EMILI framework allows the definition of both high specialized problem-specific components as well as general problem-agnostic components. In order to use an automatic configuration tool to design an SLS algorithm, such as irace, the grammar is converted to a set of parameters. In this paper, *AAD* is used to tackle the permutation flowshop problem with the sequence dependent setup times and the no-idle constraints.

The permutation flowshop problem (PFSP) is a very well known problem [14]. It has been shown to be $\mathcal{NP}$-hard for different objectives with the exception of the two machine case for the makespan objective [23]. The problem models a flowshop where a set of jobs have to be processed on a group of machines. Several additional constraints have been proposed in the literature to take into account different scenarios. Often, machines have to be set up before being able to process a job. For instance, a machine may need to be cleaned or calibrated before processing another job. The setup time may depend not only on the job that has to be processed, but also on the changes made to the setup of the machine before processing the previous job. The permutation flowshop problem with sequence dependent setup times, PFSP^{sdst}, has been introduced to model this scenario. This problem has been shown to be $\mathcal{NP}$-hard, considering the makespan objective, even when there is only one machine [18]. Considering the complexity hierarchies for scheduling problems, this result can be extended to the total completion time and total tardiness objectives [46]. Several SLS algorithms such as iterated local search [65], iterated greedy [51] and population-based algorithms [30,50,58] have been proposed to solve the permutation flowshop problem with such constraint.

No-idle permutation flowshop (PFSP^{ni}) is another such variant. PFSP^{ni} models a scenario where machines cannot have idle times. This constraint is necessary in some contexts such as the steel industry, ceramic production or in photolithography methods used in the production of integrated circuits [45]. PFSP^{ni} with the makespan objective is also an $\mathcal{NP}$-hard problem [1]. Following the same reasoning about complexity hierarchies for scheduling problems [46], the no-idle permutation flowshop problem can be assumed to be $\mathcal{NP}$-hard also when considering the total completion time and total tardiness objectives. Among the SLS algorithms proposed for this problem there are iterated greedy [40], memetic algorithms [55] and variable neighborhood search [61].

In this paper we extend the work carried out on automatic algorithm design for the permutation flowshop problem by considering the

* Corresponding author.
E-mail addresses: federico.pagnozzi@ulb.ac.be (F. Pagnozzi), stuetzle@ulb.ac.be (T. Stützle).

https://doi.org/10.1016/j.orp.2021.100180
Received 19 November 2020; Received in revised form 27 January 2021; Accepted 27 January 2021
Available online 30 January 2021

permutation flowshop problem with the sequence dependent setup times and the no-idle constraints. For each constraint we consider the minimization of the makespan, the total completion time and the total tardiness. For each objective and constraint, the algorithms generated by our *AAD* system are compared with the best performing algorithm from the literature. The results show that the generated algorithms outperform the state of the art.

The paper is structured as follows. In Section 2 there is a definition of PFSP, PFSP^{ni} and PFSP^{sdst}. In Section 3, there is a description of how the automatic design works and how the system was updated for these problems. The experimental setup is reported in Section 4. We report the experimental results for the sequence setup times problem and the no-idle problem, respectively, in Section 5 and Section 6. Finally, the conclusions are in Section 7.

2. Permutation flowshop with additional constraints

In its standard formulation, the PFSP models a flowshop in which a series of n jobs $\{J_1, \ldots, J_n\}$ have to be processed one at a time, in order, on a set of m machines $\{M_1, \ldots, M_m\}$. The jobs are released at time 0 and the jobs are executed in order with no preemption allowed. A solution is represented by a permutation $\pi = \{\pi(1), \ldots, \pi(k), \ldots, \pi(n)\}$ that specifies the processing order of the jobs. The processing time needed for a job j on a machine i is indicated as $p_{i,j}$. The completion time of a job $\pi(j)$ on a machine i is given by Eq. (1), where $C_{i,j-1}$ is the completion time of the last job processed on machine i while $C_{i-1,j}$ is the completion of job $\pi(j)$ on the previous machine, that is

$$C_{i,j} = \max\left(C_{i,j-1}, C_{i-1,j}\right) + p_{i,\pi(j)}. \tag{1}$$

In PFSP^{sdst}, each machine has to undergo a setup operation before being able to process the following job. A matrix S of dimension $n \times n \times m$ is defined, where $S_{i,l,k}$ is the setup time that machine i needs when passing from working on job l to job k. The setup time has to be considered when computing the completion times for each job. Consequently, Eq. (1) is modified as follows:

$$C^{sdst}_{i,j} = \max\left(C_{i,j-1}, C_{i-1,j} + S_{i,\pi(j-1),\pi(j)}\right) + p_{i,\pi(j)} \tag{2}$$

The time required to setup the machine depends on the job that has to be processed and on the last job processed.

In PFSP^{ni}, machines cannot have idle times, that is, as soon as a job is completed, the next one should be ready to start processing. The completion time is calculated as in Eq. (3)

$$C^{ni}_{i,j} = \max\left(C_{i,j-1} + a_{i,\pi(j-1),\pi(j)}, C_{i-1,j}\right) + p_{i,\pi(j)}, \tag{3}$$

where $a_{i,\pi(j-1),\pi(j)}$ is calculated as in Eq. (4) and it is used to ensure that there is no idle time between the starting of job $\pi(j)$ and the ending of the previous job $\pi(j-1)$, that is

$$a_{i,\pi(j-1),\pi(j)} = \sum_{k=2}^{k=i} \max\left(C_{k-1,j} - C_{k,j-1}, 0\right). \tag{4}$$

The most common objective considered for the flowshop problem is minimizing the makespan, $C_{\max}$, that is the time needed to complete all jobs. The makespan is defined as $C_{\max} = C_{m,n}$. Another common objective considers minimizing the sum of the completion times which minimizes the occupation time of the machines. The sum of completion times is defined as

$$TCT = \sum_{i=1}^{n} C_{m,i}.$$

This objective is also known as total completion time and it is equal to the minimization of the flowtime when the release times for all jobs are equal to zero. Finally, we consider also the minimization of the total tardiness, which tries to minimize the tardiness of all jobs. Assigning a due date to each job so that d_i is the due date of job J_i, the tardiness of J_i is defined as $\max(0, C_{m,i} - d_{\pi(i)})$ and the total tardiness is

$$TT = \sum_{i=1}^{n} \max\left(0, C_{m,i} - d_{\pi(i)}\right).$$

Summarizing, in this paper we are considering the makespan, sum of completion times and total tardiness objectives for both sequence dependent setup times constraint and the no-idle constraint. An analysis of the state of the art for these problems is given in Sections 5 and 6.

3. Automatic algorithm design

Historically, implementing an SLS algorithm for some problem has always been a manual engineering process. A designer would implement an SLS algorithm of his choice, usually the one he knows the most, as well as one or more alternative behaviors for each aspect of the algorithm [20]. Moreover, SLS algorithms have often many parameters that need to be set in order to better adapt the algorithm to the problem it has to solve. The designer would choose among the alternative behaviors and set the parameters of the algorithm in a manual trial-and-error process or, more recently, using automatic algorithm configuration (*AAC*). Given an application scenario, *AAC* tools apply different techniques in order to find the best parameter setting, known as configuration, for a target application [21,22,34]. Automatic algorithm design stems from the work carried out on automatic algorithm configuration and is based on combining *AAC* tools with configurable algorithmic frameworks [59]. A configurable algorithmic framework implements one or more SLS algorithms as templates in such a way that for every design choice of the algorithm one can choose among different alternatives. The framework exposes all these choices as parameters so that an *AAC* tool can be used to find the best configuration, that is, a new algorithm adapted to solve a specific problem.

This idea has been considered both in a top-down and bottom-up approach. The top-down approach focuses on one algorithm template and expresses all the design choices as parameters, e.g. implementing a simulated annealing algorithm where all the components are parameters. Notable examples of this approach can be found in SAT solvers [27], frameworks for ant colony optimization algorithms [32] and for multi-objective evolutionary algorithms [2–4]. Recently, this approach has been applied to generate iterated local search algorithms for the standard PFSP [6,7], the unconstrained quadratic problem [11], the test-assignment problem [12] and the simulated annealing algorithm [15].

In the bottom-up approach instead, the template structure is not fixed, that is, different types of SLS algorithms can be instantiated. Moreover, some degree of hybridization between different algorithms is allowed, enabling the system to generate new combinations. Such flexibility requires the use of an algorithmic framework to define the components and handle their integration into an algorithm. Furthermore, since the template is not fixed, defining the set of parameters that the parameter tuner has to optimize is not a trivial task. For this reason, context-free grammars have been used to specify how the algorithms should be composed [8,36,42]. Grammars have the advantage of limiting all the possible combinations of components to only those that generate a valid algorithm.

The bottom-up approach has been applied to several problems such as PFSP with the weighted tardiness objectives, unconstrained binary quadratic problem and traveling salesman problem with time windows [35]. The proposed system uses the ParadisEO framework [10] as algorithmic framework, irace as parameter tuner and the grammar is converted to a set of finite parameters following the approach proposed by Mascia et al. [37]. In a most recent publication [43], a system based on the same principles, but with a new algorithmic framework, the EMILI framework, has been used to generate new state-of-the-art algorithms for the PFSP problem with the makespan, sum of completion

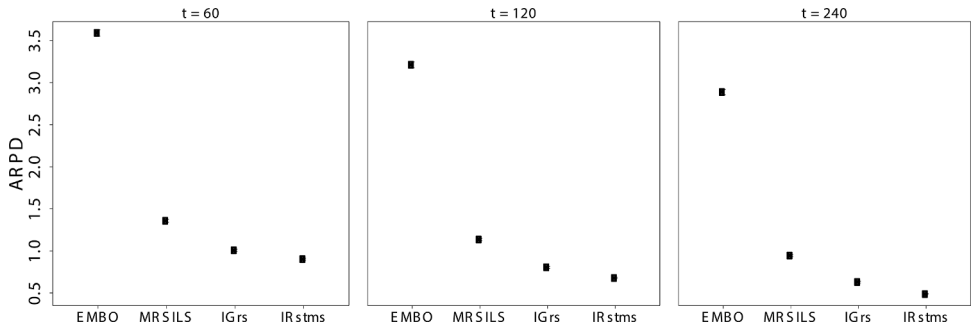

Fig. 1. Context-free grammar that contains the rules used to build algorithm templates for this study. Note that rules ILS together with LocalSearch define a recursion that can be exploited to generate hybrids combining various algorithms.

1: **Output** The best solution found π^*
2: π := Init()
3: π := SLS(π)
4: **while** ! termination criterion **do**
5: π' := Perturbation(π)
6: π' := SLS(π')
7: π := AcceptanceCriterion(π, π')
8: **end while**
9: **Return** the best solution found in the search process

Algorithm 1. ILS.

times, total tardiness objectives. This system, with some additions to the framework, is the same used in this study.

A line of research related to automatic algorithm design can be considered the one on hyperheuristics [9]. These methods propose techniques to generate heuristics that can be seen as specific components in automatic algorithm design [59]. Hyperheuristics based on genetic programming [29], where evolutionary algorithms are used to generate computer programs, have been used to generate heuristics and heuristic components for problems such as SAT problems [16,17], scheduling problems [5,19], bin packing [33,57] and traveling salesman problem [25,26]. Grammars have been also used together with genetic programming in methods called grammatical evolution [8,42]. Grammatical evolution has been applied to generate local search heuristics and ant colony optimization algorithms [8,54,63].

3.1. Grammar based AAD with the EMILI framework

An idea of how the grammar works can be given by making a small example. Let us consider a rule deriving an iterated local search (ILS) algorithm as shown in Eq. (5)

$$< \text{ILS} > \; ::= \; \text{`ils'} \; < \text{LocalSearch} > \; < \text{Termination} > \; < \text{Perturbation} > \; < \text{Acceptance} > \; . \tag{5}$$

The rule states that to instantiate an ILS algorithm one needs to instantiate first the components <LocalSearch>, <Termination>, <Perturbation> and <Acceptance>. For example, an ILS algorithm for the PFSP could be described by

$$ils_{pfsp} \; ::= \text{`}ils\text{'} \; ls_{pfsp} \; \text{`}time20\text{'} \; \text{`}random\_moveexchange2\text{'} \; \text{`}better\text{'} \; . \tag{6}$$

The algorithm described in Eq. (6) is an ILS that uses ls_{pfsp} as local search, it is executed for 20 s, performs two random steps in the exchange neighborhood as perturbation and accepts only improving solutions. Fig. 1 shows a snapshot of the grammar used for this study. The different components available for each component type are listed in Section 3.2.

When converting the grammar to parameters, a distinction has to be made between simple and complex rules. Simple rules can be directly translated to parameters. For instance, a rule that sets all possible alternatives for a neighborhood can be directly translated into a categorical parameter. Complex rules, that is recursive rules or groups of rules that can form a loop, need to be explicitly expanded. This means that the rule, or the group of rules in case of loops, generate a new set of parameters each time it is expanded. Consequently, a limit needs to be set to the total number of expansions. A more detailed explanation of how the grammar is defined and converted in parameters can be found in [37]. In this work we fix the maximum number of expansions to three.

The EMILI framework is based on a generalized version of a hybrid SLS algorithm. Hybrid SLS algorithms are defined in [20] as those that manipulate at each search step a single solution combining two or more search types. The framework supports also "simple" SLS algorithms that are identified in [20] as SLS algorithms that use only one type of search step. An outline of an iterated local search algorithm, which represents a hybrid SLS template as implemented in the framework, is shown in Algorithm 1.

The algorithm works as follows. An initial candidate solution is generated using a heuristic (Line 2). Then an SLS is applied to the candidate solution (Line 3). The algorithm executes the main loop until the termination criterion is met (Line 4). In the main loop, the candidate solution is perturbed (Line 5), the SLS is applied to the perturbed solution (Line 6), and an acceptance criterion is used to decide whether to keep the current candidate solution or to accept the perturbed solution (Line 7).

As explained in [37], this structure can describe many different SLS algorithms. For instance, simulated annealing can be instantiated by choosing an initial solution component that returns a random candidate solution, a perturbation that generates random neighboring solutions of the candidate solution, an acceptance criterion based on the Metropolis condition and not applying any SLS to the perturbed solution.

The EMILI framework classifies algorithmic components in problem dependent and problem independent components. The first are components that have to access and modify the data structures representing the problem and the solution. Typically, initial solution heuristics, neighborhoods and perturbations belong to this category. Problem independent components, instead, only need to compare solutions or access information about the search process (e.g. number of iterations). These components can be defined once and then used with any problem

Table 1
Algorithmic components implemented in the EMILI framework that were used in this work.

Type	Component	Parameters
Construction Heuristics	*NEH* [41], NEH_{tb} [13], NEH_{edd} [28], *LR* [31], *NLR*	-
	FRB5 [47], *RZ* [48], *NRZ*, NRZ_2, *SLACK*,	-
	NAG [40]	$\langle x, y \rangle$
	NEH_{rs}	-
Iterative improvements	First Improvement	$\langle In, T, N \rangle$
	Best improvement	$\langle In, T, N \rangle$
	VND	$\langle P, In, T, \{N_1, \ldots, N_k\} \rangle$
	iRZ	$\langle In \rangle$
	STH	$\langle b \rangle$
	als	$\langle l_1, l_2 \rangle$
Neighborhoods	*transpose, exchange, insert, binsert finsert, sttinsert, nitinsert, twinsert*	-
Termination criteria	*local minimum*	-
	maxsteps	$\langle maxi \rangle$
	maxstepsorlocmin	$\langle maxi \rangle$
	nstepsorlocmin	-
	non_imp_it	$\langle maxi \rangle$
Perturbation criteria	*random_move*	$\langle N, num \rangle$
	vr_move	$\langle \{d, num, (N_1, \ldots, N_k)\} \rangle$
	IG_{lsps}	$\langle d \rangle$
	IG,IG_{st},IG_{ni}	$\langle d \rangle$
	IG_{io}	$\langle d \rangle$
	MRSILSp	$\langle p \rangle$
	shake	$\langle \{P_1, \ldots, P_n\} \rangle$
Acceptance criteria	*better*	$\langle \varnothing \rangle$
	improveif	$\langle s_t, s_n \rangle$
	ft	$\langle T \rangle$
	psa	$\langle T_s, T_e, \beta, it \rangle$
	sa	$\langle T_s, T_e, \beta, \alpha, it \rangle$
	rsacc	$\langle T_p \rangle$
	karacc	$\langle T_p \rangle$
	neighchange	$\langle A \rangle$

and solution definition. Termination criteria and acceptance criteria usually can be implemented as problem independent.

In the next section, we will give a brief description of the algorithmic components used for this study.

3.2. Algorithmic components

All the components used in this work are listed in Table 1. Most of these components were implemented for the previous work on PFSP as general components and, therefore, can be used in this work without any change. The components added to the framework for this work are

1: **Output** The best solution found π^*,
2: π := Init()
3: π := ls (π)
4: k := 0
5: **while** ! termination criterion **do**
6: π' := Shake(π, k)
7: π' := ls (π')
8: π := NeighborhoodChange(π, π', k)
9: **end while**
10: **Return** the best solution found in the search process

Algorithm 2. VNS.

1: **Output** The best solution found π^*,
2: π := *NRZ*()
3: π := *FirstImprovement*(π, *local minima*, *sttinsert*)
4: π^* := π
5: **while** ! time is over **do**
6: π' := $IG_{st}(\pi)$
7: π' := *FirstImprovement*(π', *local minima*, *sttinsert*)
8: π := *psa*(π, π')
9: **if** $f(\pi') < f(\pi^*)$ **then**
10: π^* := π'
11: **end if**
12: **end while**
13: **Return** π^*

Algorithm 3. IR_{stms} .

reported in the table in bold. In the following, we give a description of the components implemented for this study and a brief presentation of the components already present in the framework. A more detailed description of these components can be found in our previous publication [43].

3.2.1. Neighborhood

A neighborhood of a solution consists of all the solutions that can be generated by applying a modification rule. The framework provides several base neighborhood definitions for PFSP: *exchange, insert, transpose, binsert, finsert* and *twinsert*. The first two are based on exchanging the position of two jobs (*exchange*) and removing one job and inserting it in another position (*insert*). The others, with the exception of *twinsert*, represent a subset of the first two. The *transpose* neighborhood exchanges only adjacent jobs. In *binsert* a removed job can be inserted only before its original position, while in *finsert* the insertion point has to be after the original position. Instead, *twinsert* considers all the permutations that can be created by removing and inserting groups of two adjacent jobs. The two jobs are reinserted in the same order in which they are removed. Additionally, Taillard's technique to speedup the exploration of the insert neighborhood has been adapted to $PFSP^{ni}$ and $PFSP^{sdst}$.

3.2.2. Construction heuristics

SLS algorithms use construction heuristics to generate the initial solution from which they start to explore the solution space. In our previous study several construction heuristics for the PFSP have been implemented in the EMILI framework as general components and, therefore, were also used in this study. Typically, construction heuristics build a solution by adding solution components in a step-by-step process until a complete solution is constructed. A construction heuristic can be defined by the way the solution component is selected, the selection rule, and how it is added to the partial solution, the construction rule. Solution components can be added in two ways by either appending the solution component at the end of the partial solution or inserting it in the position that gives the best solution value. Heuristics that use insertion as construction rule are also called insertion heuristics. In some cases, a local search may be applied to the partial solution.

Table 2
Parameter settings for IR_{stms} .

	Component	Parameter	Value
IR_{stms}			
	IG_{st}	d	6
	psa	T_s	4.2073
		T_e	0.0441
		β	0.0042
		it	206

Table 3
Average RPD results of *EMBO*, *MRSILS*, IG_{rs} and IR_{stms}. If the result of one of the algorithms is in bold face it means that it is statistically significantly better then the others according to the Wilcoxon signed-rank test with a 95% confidence using the Bonferroni correction to take into account multiple comparisons.

	$t = 60$				$t = 120$				$t = 240$			
Instances	*EMBO*	*MRSILS*	IG_{rs}	IR_{stms}	*EMBO*	*MRSILS*	IG_{rs}	IR_{stms}	*EMBO*	*MRSILS*	IG_{rs}	IR_{stms}
20 × 5	0.77	0.08	0.17	**0.06**	0.53	0.05	0.12	**0.02**	0.35	0.04	0.09	**0.01**
20 × 10	0.75	0.10	0.20	**0.09**	0.54	0.07	0.16	**0.05**	0.37	0.05	0.12	**0.03**
20 × 20	0.50	0.07	0.12	**0.05**	0.36	0.04	0.09	**0.02**	0.25	0.03	0.07	**0.01**
50 × 5	3.88	1.39	1.25	**1.04**	3.62	1.19	1.06	**0.86**	3.41	1.02	0.90	**0.74**
50 × 10	4.08	1.53	1.33	**1.01**	3.84	1.31	1.15	**0.84**	3.61	1.14	0.98	**0.71**
50 × 20	3.52	1.39	1.23	**0.88**	3.32	1.19	1.06	**0.71**	3.13	1.05	0.90	**0.60**
100 × 5	4.49	2.02	1.45	**1.31**	3.98	1.68	1.20	**1.01**	3.66	1.40	0.96	**0.75**
100 × 10	4.49	1.92	1.33	**1.23**	4.05	1.58	1.08	**0.94**	3.79	1.31	0.88	**0.68**
100 × 20	4.41	1.91	1.36	**1.20**	4.02	1.60	1.10	**0.93**	3.78	1.34	0.89	**0.66**
200 × 10	5.52	2.17	**1.30**	1.37	4.67	1.80	0.98	1.00	3.98	1.45	0.69	**0.64**
200 × 20	5.32	2.02	1.29	1.28	4.59	1.70	0.99	**0.93**	3.92	1.40	0.73	**0.59**
500 × 20	5.35	1.71	**1.05**	1.31	5.00	1.42	**0.69**	0.83	4.41	1.12	**0.34**	0.42
Average	3.59	1.36	1.00	**0.90**	3.21	1.14	0.81	**0.68**	2.89	0.95	0.63	**0.49**

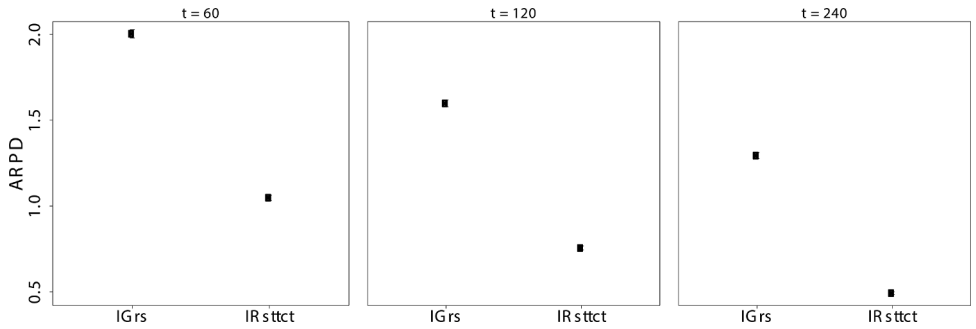

Fig. 2. Average RPD and 95% confidence intervals of *EMBO*, *MRSILS*, IG_{rs} and IR_{stms} for $t = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

The *NEH* heuristic [41] is rated as one of the most effective heuristics for PFSP. This insertion heuristic selects the jobs in descending order of the sum of processing times. *NEH* has been so influential that several improvements and variations have been proposed. The ones implemented in EMILI are NEH_{tb} [13], NEH_{edd} [28], *FRB5* [47] and NEH_{rs}. NEH_{tb} introduces a different rule to break ties when a job can be inserted in more than one position resulting in the same objective function value. NEH_{edd} selects the job in descending order of due dates and is feasible only for the total tardiness objective. *FRB5* executes a local search on the partial solution after each step of the heuristic. Finally, the NEH_{rs} heuristic modifies the initial step of the *NEH* by choosing randomly the first job of the permutation.

Considering the other heuristics implemented, the *LR* heuristic [31] uses for the selection an index function that considers the idle times and an approximation of the sum of completion times. At each step the index function is calculated and the job with the smaller value is appended at the end of the partial solution. *NLR* is an insertion heuristic that uses the index function of the *LR* heuristics to select the next job. The *RZ* heuristic appends first to the partial solution the jobs that minimize a function based on the weighted sum of processing times [48]. The generated solution is improved by means of a local search. *NRZ* and NRZ_2 are insertion heuristics that are based on the *RZ* heuristic. *NRZ* uses the solution generated by *RZ* before applying the local search as selection rule of an insertion heuristic. The generated solution is still improved as in *RZ* with a local search. In NRZ_2 the local search is not applied, generating a typically worse solution but in little time. *SLACK* is a construction heuristic for the total tardiness objective that appends jobs to the partial solution by selecting at each step the job with the minimal tardiness.

For this work, the *NAG* insertion heuristic [40] proposed for $PFSP^{ni}_{TCT}$ was implemented in EMILI. This heuristic uses the index function of the *LR* heuristic and after each insertion another index function is used to select y jobs to remove and reinsert in the partial solution. A number x of initial sequences is generated by choosing for each sequence a different first job to append to the partial solution. The parameters y and x can

1: **Output** The best solution found π^*,
2: $\pi := NEH()$
3: $\pi := IR_{sttct}\ 2(\pi)$
4: $\pi^* := \pi$
5: **while** ! time is over **do**
6: $\quad \pi' := IG(\pi)$
7: $\quad \pi' := IR_{sttct}\ 2(\pi')$
8: $\quad$ **if** $f(\pi') < f(\pi^*)$ **then**
9: $\quad\quad \pi^* := \pi'$
10: $\quad$ **end if**
11: **end while**
12: **Return** π^*

Algorithm 4. IR_{sttct}.

Output The best solution found π^*,
Input current solution π.
$\pi := IR_{sttct}\ 3(\pi)$
$\pi^* := \pi$
while *maxsteps*() **do**
  $\pi' := IG(\pi)$
  $\pi' := IR_{sttct}\ 3(\pi)$
  $\pi := sa(\pi', \pi')$
  if $f(\pi') < f(\pi^*)$ **then**
    $\pi^* := \pi'$
  end if
end while
Return π^*

Algorithm 5. IR_{sttct2}.

Output The best solution found π^*,
Input current solution π.
$\pi := VND(\pi, nstepsorlocmin, binsert, exchange, twinsert)$
$\pi^* := \pi$
while *nstepsorlocmin*() **do**
  $\pi' := IG_{lsps}(\pi, FirstImprovement(local\ minima, exchange))$
  $\pi' := VND(\pi', local\ minima, binsert, exchange, twinsert)$
  $\pi := \pi'$
  if $f(\pi') < f(\pi^*)$ **then**
    $\pi^* := \pi'$
  end if
end while
Return π^*

Algorithm 6. IR_{sttct3}.

assume values in the interval $[1, n]$.

3.2.3. Iterative improvement

Iterative improvement algorithms are local search algorithms that explore the solution space in an iterative fashion by going from one solution to an improving neighboring solution. This process typically stops when no improving neighbor can be found or, in some cases, after a certain number of steps. These algorithms take as input the starting solution, the neighborhood relation and the way to choose which candidate neighbor is selected for the next iteration, known as pivotal rule. The algorithm can consider also multiple neighborhood relations as in the variable neighborhood descent (VND).

The algorithms already implemented in the EMILI framework and that were used in this study comprehend the most widely used pivotal rules *FirstImprovement* and *BestImprovement* as well as the *iRZ* local search and VND. In *FirstImprovement* the exploration of the neighborhood of the current solution is stopped as soon as an improving solution is found. In *BestImprovement* instead, the whole neighborhood is explored and the best neighbor is returned. The *iRZ* algorithm iterates the local search phase defined in the *RZ* heuristic until it cannot find any improving solutions. The VND explores, in order, a set of neighborhoods passing from one neighborhood to the next when no improvement is found. Each time an improving solution is found the algorithm starts again from the first neighborhood. The algorithm stops when it scanned all the neighborhoods with no improvement. Additionally, two other algorithms have been implemented for this study, *STH* [58] implemented for PFSP^{sdst}_{MS} and *als* that has been implemented as a problem independent component. *STH* selects a block of jobs of size [1,3] and evaluates b insertion points, choosing the best. The process is iterated b times, where b is a parameter of the algorithm. The *als* local search takes as parameters two iterative improvements algorithms (l_1 and l_2) and, at each iteration, applies them alternatively.

3.2.4. Perturbation

The perturbation in an SLS has the role of letting the search process escape local minima by changing the current solution in a way that cannot be undone by the local search. The most simple way of implementing a perturbation is by taking a random neighbor of the current solution. *random_move* will perturb the current solution by executing *num* random steps in the *N* neighborhood. Instead, *vr_move* expands the concept of *random_move* by allowing to specify multiple neighborhoods *N*. The number of random steps to execute per neighborhood is specified by the parameter *num*. The neighborhood is changed to the next one after *it* iterations.

A widely used perturbation scheme for the PFSP is the iterated greedy (IG) perturbation. This scheme is composed of a destruction phase and a construction phase. In the destruction phase a number d of jobs are removed from the solution. In the construction phase, the jobs are inserted in the partial solution, one by one, in the position that minimizes the objective function value. This perturbation is implemented in IG, IG_{ni} and IG_{st} where the last two use Taillard's acceleration for, respectively, PFSP^{ni}_{MS} and PFSP^{sdst}_{MS} to find the best insertion point in the construction phase. In IG_{io} the jobs to be reinserted are considered in the descending order of sum of processing times. With IG_{lsps}, a local search is used to further improve the partial solution after each reinsertion. The *MRSILSp* perturbation keeps a pool of *size* solutions containing the best *size* solutions. If the pool is full, the worst solution of the pool is discarded; when the pool is not yet full, the current solution is perturbed by executing t random steps in the transpose neighborhood and returned. When the pool is full, a random solution is selected from the pool and perturbed using the IG perturbation.

3.2.5. Termination condition

Termination conditions are components that trigger the stop of an SLS algorithm when a certain condition is verified. Several termination conditions have been considered. *local minima* will stop the execution when there is no more improvement. *maxsteps* instead stops the execution when *maxi* iterations have been completed. *maxstepsorlocmin* combines the first two: the algorithm will be stopped either when there is no more improvement or after *maxi* iterations. *non_imp_it* stops the algorithm if no improvement is achieved in *maxi* iterations. Finally, *nstepsorlocmin* works in the same way as *maxstepsorlocmin*, but the *maxi* parameter is always set to the number of jobs.

3.2.6. Acceptance criterion

In an SLS algorithm, the acceptance criterion influences the balance

Table 4
Average RPD results of IG_{rs} and IR_{sttct}. If an algorithm is statistically significantly better according to the Wilcoxon signed-rank test with a 95% confidence, the result is shown in bold face.

	$t = 60$		$t = 120$		$t = 240$	
Instances	IG_{rs}	IR_{sttct}	IG_{rs}	IR_{sttct}	IG_{rs}	IR_{sttct}
20×5	0.35	**0.01**	0.30	**0.001**	0.24	**0**
20×10	0.18	**0.01**	0.15	**0.003**	0.12	**0.002**
20×20	0.13	**0.003**	0.11	**0.001**	0.09	**0.001**
50×5	2.40	**1.16**	2.08	**0.89**	1.86	**0.70**
50×10	1.94	**1.00**	1.74	**0.78**	1.55	**0.63**
50×20	1.49	**0.78**	1.31	**0.63**	1.18	**0.52**
100×5	3.89	**2.22**	3.22	**1.57**	2.68	**1.01**
100×10	2.79	**1.80**	2.33	**1.28**	1.95	**0.84**
100×20	2.12	**1.37**	1.76	**0.97**	1.45	**0.63**
200×10	3.24	**1.99**	2.59	**1.39**	1.95	**0.78**
200×20	2.27	**1.55**	1.79	**1.10**	1.34	**0.59**
500×20	3.25	**0.68**	1.79	**0.44**	1.13	**0.19**
Average	2.00	**1.05**	1.60	**0.76**	1.29	**0.49**

Table 5
Parameter settings for IR_{sttct} .

	Component	Parameter	Value		Component	Parameter	Value
IR_{sttct}				IR_{sttct2}			
	IG	*d*	2		*maxsteps*	*maxi*	32
IR_{sttct3}					*IG*	*d*	9
	IG_{lsps}	*d*	5		*sa*	t_s	2.0027
						t_e	0.7216
						β	0.0313
						α	0.4397
						it	308

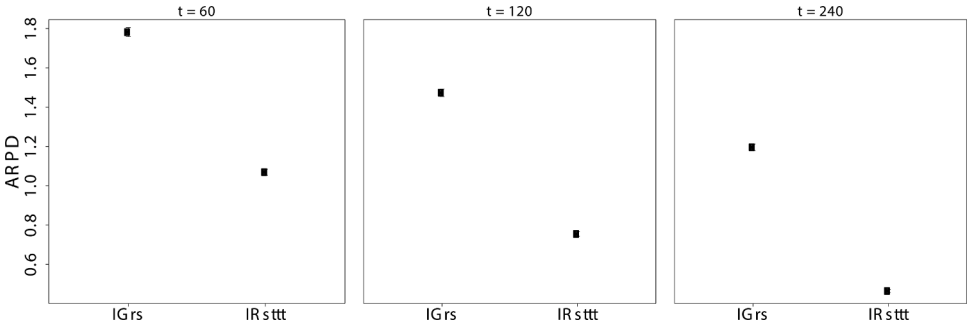

Fig. 3. Average RPD and 95% confidence intervals of IG_{rs} and IR_{sttct} for $t = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

between intensification and diversification. The most simple of such criteria, *improve*, accepts only improving solutions. With *improveif*, a certain degree of diversification is introduced by allowing the criterion to accept worsening solutions if no improving solution has been met for a certain number of iterations. The framework also provides several variants of acceptance criteria based on the Metropolis condition [38]. This probabilistic criterion, commonly used in simulated annealing, accepts incumbent solutions with a probability P_a defined as in Eq. (7)

$$P_a = \begin{cases} 1 & \text{if } f(\pi^{'}) \leq f(\pi) \\ exp\left(\frac{f(\pi) - f(\pi^{'})}{T}\right) & \text{otherwise.} \end{cases} \tag{7}$$

Commonly in simulated annealing the temperature T is updated during the algorithm execution. The temperature would start at a certain value T_s and then decrease, according to a schedule, until it reaches a final value T_e. The schedule we used updates the temperature every *it* iterations following the rule $T_{n+1} = \alpha \cdot T_n - \beta$ where α and β are real values between 0 and 1. The criteria *sa* and *psa* update the temperature using this rule, with *psa* setting α always equal to 1. Instead, *ft, rsacc* and *karacc* do not update the temperature. In the *rsacc* [52] acceptance criterion, the temperature T_{rs} is linked to the average processing time of the instance to be solved and it is calculated as

$$T_{rs} = T_p \cdot \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} p_{i,j}}{n \cdot m \cdot 10}, \tag{8}$$

where T_p is a parameter. The *karacc* [24] acceptance criterion adapts the *rsacc* criterion to the total tardiness by calculating the temperature T_{kar} as

$$T_{kar} = T_p \cdot \frac{\sum_{j=1}^{n} LB_{C_{\max}} - d_j}{n \cdot 10}, \tag{9}$$

where $LB_{C_{\max}}$ is the lower bound for the makespan calculated using the method defined by Taillard [60]. Additionally, an SLS algorithm can be set to always accept the perturbed solution regardless of its solution quality.

Output The best solution found π^*,
π := *NEH*()
π := IR_{sttt} 2(π)
π^* := π
while ! time is over **do**
  π' := *random_move*(π, *binsert*)
  π' := IR_{sttt} 2(π')
  π := *rsacc*(π, π')
  if $f(\pi') < f(\pi^*)$ **then**
    π^* := π'
  end if
end while
Return π^*

Algorithm 7. IR_{sttt}.

Output The best solution found π^*,
Input current solution π.
π := *als*(π,*FirstImprovement*(*local minima*, *finsert*), *FirstImprovement*(*maxstepsorlocmin*, *twinsert*))
π^* := π
while *maxstepsorlocmin*() **do**
  π' := *IG*(π)
  π' := *als*(π',*FirstImprovement*(*local minima*, *finsert*), *FirstImprovement*(*maxstepsorlocmin*, *twinsert*))
  π := *rsacc*(π, π')
  if $f(\pi') < f(\pi^*)$ **then**
    π^* := π'
  end if
end while
Return π^*

Algorithm 8. IR_{sttt2}.

Table 6
Average RPD results of IG_{rs} and IR_{sttt}. If an algorithm is statistically significantly better according to the Wilcoxon signed-rank test with a 95% confidence, the result is shown in bold face.

	$t = 60$		$t = 120$		$t = 240$	
Instances	IG_{rs}	IR_{sttt}	IG_{rs}	IR_{sttt}	IG_{rs}	IR_{sttt}
20 × 5	0.28	**0.03**	0.22	**0.01**	0.18	**0.01**
20 × 10	0.11	**0.01**	0.09	**0.00**	0.08	**0.00**
20 × 20	0.08	**0.00**	0.06	**0.00**	0.05	**0.00**
50 × 5	2.20	**1.17**	1.92	**0.85**	1.72	**0.59**
50 × 10	1.82	**1.02**	1.61	**0.76**	1.44	**0.53**
50 × 20	1.34	**0.75**	1.18	**0.54**	1.06	**0.38**
100 × 5	3.71	**2.27**	3.05	**1.61**	2.47	**1.00**
100 × 10	2.62	**1.67**	2.15	**1.17**	1.73	**0.75**
100 × 20	2.02	**1.33**	1.66	**0.94**	1.34	**0.60**
200 × 10	3.12	**2.11**	2.50	**1.46**	1.87	**0.78**
200 × 20	2.24	**1.60**	1.77	**1.09**	1.29	**0.58**
500 × 20	1.86	**0.88**	1.47	**0.61**	1.13	**0.35**
Average	1.78	**1.07**	1.47	**0.75**	1.20	**0.46**

3.2.7. Adding VNS to the EMILI framework

The VNS algorithm is considered as a special case of the ILS algorithm. The similarity is evident if we consider the outline of both algorithms, shown in Algorithm 2 for the VNS and in Algorithm 1 for the ILS. Both use a heuristic to generate an initial solution and use a local search for intensification. A VNS algorithm is characterized by the shake and the neighborhood change as shown in Algorithm 2. The shake works as a perturbation applying random changes to the current solution according to one neighborhood. The shake keeps a set of neighborhoods and selects the one to apply according to the parameter k. The neighborhood change component acts as an acceptance criterion and controls the parameter k. When the current solution is accepted, k is set to zero otherwise it is incremented by one.

4. Experimental settings

In this section we report the setup used for the automatic design and for the comparisons with the current state-of-the-art algorithms. In order to apply the automated design approach presented in this paper, the grammar presented in Section 3 needs to be adapted to each objective and PFSP variant. The resulting six grammars maintain the same general structure shown in Fig. 1, but have different variant-specific and objective-specific components. For example, the speedup for the insert neighborhood for $PFSP^{sdst}_{MS}$ is not present in the grammars for $PFSP^{sdst}_{TCT}$ and $PFSP^{sdst}_{T}$ and the same applies for no-idle. Considering these differences, the number of parameters to tune were 627 for the three objectives of $PFSP^{ni}$, 535 for $PFSP^{sdst}_{MS}$ and 507 for $PFSP^{sdst}_{TCT}$ and $PFSP^{sdst}_{T}$.

The configuration space generated from the grammar needs to be explored by an automatic configuration tool. For this task we chose irace, a publicly available *AAC* tool [34]. For each objective and PFSP variant, irace was run twice with a budget of 10^5 experiments per run for a total of $2{\cdot}10^5$ experiments. The best configurations at the end of the first run were given as initial configurations for the second run. The training set used for the automatic configuration is the same one used in [43]. This set is composed of 40 randomly generated instances following the procedure described in [39]. The instances are divided in groups of five with jobs size $n \in \{50, 60, 70, 80, 90, 100\}$ with 20 machines plus five instances of size 250×30 and five of size 250×50.

The automatically generated algorithms are compared with the current state of the art using the most commonly used benchmark instances for each PFSP variant and objective. The state-of-the-art algorithms have been implemented to the best of our ability following the respective papers using, for the experiments, the parameter settings reported by the authors. Regarding $PFSP^{sdst}$, the experiments for the three objectives were made using the benchmark presented in Ruiz and Maroto [49]. This benchmark is composed of four sets of instances. Each set is based on the original 120 instances of the Taillard's benchmark [60] comprising 12 groups of 10 instances with jobs $n \in \{20, 50, 100, 200, 500\}$ and machines $m \in \{5, 10, 20\}$. The four sets of instances, called *SDST10, SDST50, SDST100* and *SDST125*, have the setup times sampled uniformly in the range [1,9], [1,49], [1,99] and [1,124].

In the case of $PFSP^{ni}$, for makespan and sum of completion times we used the benchmark presented in Ruiz et al. [53] that is composed of 250 instances in groups of 5 with number of jobs $n \in \{50, 100, 150, 200, 250, 300, 350, 400, 450, 500\}$ and machines $m \in \{10, 20, 30, 40, 50\}$. The instances generated for this benchmark do not consider due dates. Hence for $PFSP^{ni}_{T}$ we used the benchmark presented in Vallada et al. [64] that was proposed for the standard PFSP. This benchmark is composed of 540 instances divided in groups of 45 with number of jobs $n \in \{50, 150, 250, 350\}$ and machines $m \in \{10,30,50\}$. In all cases, with the only exception of $PFSP^{ni}_{T}$, the performances have been evaluated computing the relative percentage variation (RPD) that can be calculated as follows:

$$RPD = \frac{R_a - R^*}{R^*}$$

where R_a is the solution reported by algorithm a and R^* is the best known solution. In the case of $PFSP^{ni}_{T}$, since the benchmark set has instances where the total tardiness of the best solution is equal to zero, the relative deviation index (RDI) was used. The RDI is calculated as

$$RDI = \frac{R_a - R^*}{R^w - R^*}$$

where R^w is the worst solution generated considering all the tested algorithms.

The tuning was executed on a Xeon 5410 CPU at 2.33 Ghz while the experiments were conducted on an Opteron 6410 CPU running at 2.1 Ghz. All machines use CentOS 6.2. All the algorithms in the comparisons have been implemented in the EMILI framework and each execution was single threaded. All the parameter settings as well as the best solutions

Table 7
Parameter settings for IR_{sttt} .

	Component	Parameter	Value		Component	Parameter	Value
IR_{sttt}				IR_{sttr2}			
	random_move	*num*	1				
	rsacc	T_p	0.2631		*IG*	*d*	3
					rsacc	T_p	3.3614

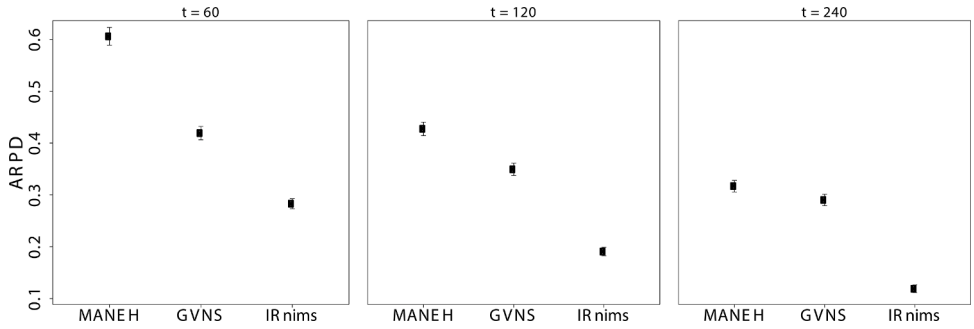

Fig. 4. Average RPD and 95% confidence intervals IG_{rs} and IR_{sttt} for $t = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

1: **Output** The best solution found π^*,
2: $\pi := NRZ_2()$
3: $\pi := IR_{nims}\ 2(\pi)$
4: $\pi^* := \pi$
5: **while** ! time is over **do**
6: $\pi' := IG_{lsps}(\pi, FirstImprovement(local\ minima, nitinsert))$
7: $\pi' := IR_{nims}\ 2(\pi')$
8: $\pi := psa(\pi, \pi')$
9: **if** $f(\pi') < f(\pi^*)$ **then**
10: $\pi^* := \pi'$
11: **end if**
12: **end while**
13: **Return** π^*

Algorithm 9. IR_{nims}.

1: **Input** current solution π.
2: **Output** The best solution found π^*,
3: $\pi := FirstImprovement(\pi, local\ minima, nitinsert)$
4: $\pi^* := \pi$
5: **while** $maxsteps()$ **do**
6: $\pi' := IG_{ni}(\pi)$
7: $\pi' := FirstImprovement(\pi', local\ minima, nitinsert)$
8: $\pi := psa(\pi, \pi')$
9: **if** $f(\pi') < f(\pi^*)$ **then**
10: $\pi^* := \pi'$
11: **end if**
12: **end while**
13: **Return** π^*

Algorithm 10. IR_{nims2}.

found for each benchmark are reported in the supplementary pages [44]. In the following, we report the algorithms generated by our *AAD* system, as well as the results of the comparison with state-of-the-art algorithms for each of the constraints and objectives tackled. Finally, all the algorithms tested are executed with a maximum running time that is calculated as $T_{\max} = n \cdot (m/2) \cdot t$ ms, where n is the number of jobs, m is the number of machines and t is a parameter. In our tests we used for the parameter t the values $\{60, 120, 240\}$.

5. Results for sequence dependent setup times PFSP

5.1. Makespan

Many different metaheuristics have been proposed for this problem like GRASP, genetic and memetic algorithms [49] before the introduction of the IG_{rs} algorithm [51]. The IG_{rs} algorithm is a very simple and powerful metaheuristic based on the *NEH* heuristic, a *FirstImprovement* local search that explores the *iRZ* neighborhood, the *IG* perturbation and a fixed temperature Metropolis like acceptance criterion. This algorithm, similarly to when it was proposed for the minimization of the makespan in the standard PFSP [43], has remained the best performing algorithm for quite sometime before new algorithms were proposed.

In 2014, $MRSILS_{st}$ [65] showed to outperform IG_{rs}. $MRSILS_{st}$ is an ILS algorithm that uses the *NEH* heuristic to generate the initial solution, an insertion based local search, a strictly improve acceptance criterion and the *MRSILSp* perturbation presented in Section 3. Recently, a migrating birds optimization, *EMBO* was proposed as new state of the art [58]. The *EMBO* algorithm divides the population in a leader solution and two groups of followers. At each iteration, the leader solution is updated by applying the *STH* algorithm. Afterwards, k solutions are selected among the swap and insert neighborhood of the leader solution. Each follower is considered going from the closest to the leader to the furthest. The solution produced applying the *STH* algorithm is compared with $k - x$ best neighbors from the previous follower solution and x neighbors selected among the swap and insert neighborhood of the current follower. Additionally, a tabu list is used to improve the neighbors selection. All the solutions are characterized by an age variable that is incremented at each iteration if the solution is not updated. If the age variable reaches $\max_{age}$ the solution is substituted by a random one.

The automatically generated algorithm, IR_{stms}, is shown in Algorithm 3 with the parameters in Table 2. This algorithm is a rather simple *IG* algorithm. It uses the *NRZ* heuristic to generate the initial solution and a *FirstImprovement* local search exploring the insert neighborhood. The perturbation is the same as IG_{rs} with a stronger destruction phase while the acceptance is closer to a classical SA acceptance.

The IR_{stms} was compared with *EMBO*, $MRSILS_{st}$ as well as IG_{rs} over the benchmark presented in [49]. The results are presented in Table 3 and in Fig. 2. IR_{stms} outperforms all the other tested algorithms, being

Table 9
Parameter settings for IR_{nims} .

	Component	Parameter	Value		Component	Parameter	Value
IR_{nims}				IR_{nims2}			
	IG_{lsps}	d	5				
	psa	T_s	3.5859		*maxsteps*	*maxi*	150
		T_e	0.0764		*IG*	d	8
		β	0.076		*psa*	T_s	3.8648
		it	336			T_e	0.6922
						β	0.014
						it	137

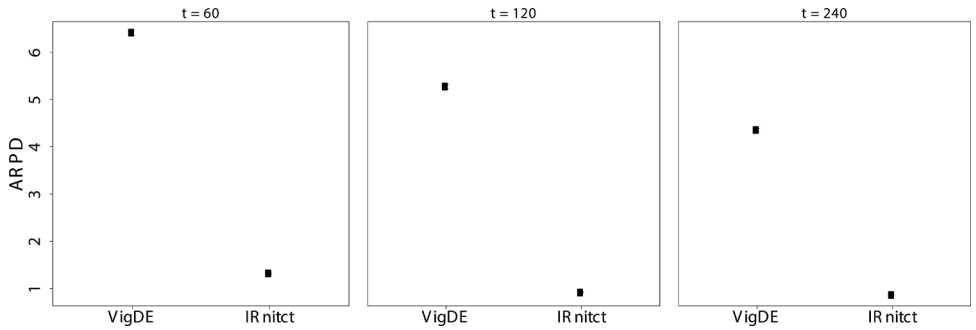

Fig. 5. Average RDI and 95% confidence intervals of *MANEH*, *GVNS* and IR_{nims} for $t = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

statistically significantly better for almost all the instance sizes with IG_{rs} being the second best followed by *MRSILS* and *EMBO*. Considering the smallest running time, IG_{rs} shows better results than IR_{stms} when considering instances with 200 or more jobs. Although the difference between the two algorithms gets smaller, this result does not change for instances with 500 jobs even when considering longer running times. In this case, IR_{stms} may be less efficient due to the lack of instances with this size in the training set. Another interesting finding about IG_{rs} performance is that, differently from the results presented in the papers introducing *EMBO* and $MRSILS_{st}$ [58,65], IG_{rs} is always able to outperform *MRSILS* and *EMBO*. One possible explanation is that this result is due to our implementation. This can be excluded in the case of *EMBO*, where a comparison with the best solutions found by the original implementation, as reported by the authors, shows that our implementation has better results. A similar comparison cannot be done with *MRSILS*, but we are confident that the algorithm was implemented as described by the authors. Since we could not find in the papers any reference about the use of Taillard's acceleration, another possible explanation is that *EMBO* and *MRSILS* were compared with a IG_{rs} algorithm that was not using this acceleration greatly reducing its performance.

Finally, the performance of IG_{rs} in our experiments and the similarity of IR_{stms} to this algorithm further confirms that the *IG* algorithm is quite effective when solving the PFSP with the makespan objective even when we take into account the sequence dependent setup times constraint.

5.2. Total completion time

Although the makespan objective for $PFSP^{sdst}$ has been extensively studied, the total completion time and total tardiness for the $PFSP^{sdst}$ have not received the same attention by the research community. To the best of our efforts, we were unable to find algorithms proposed for $PFSP^{sdst}_{TCT}$ and $PFSP^{sdst}_{T}$. The algorithms generated for these problems will be compared with the IG_{rs} algorithm presented in [51] for $PFSP^{sdst}$ to solve the makespan and total tardiness objectives. Furthermore, this algorithm has shown to have generally good performances when tackling PFSP in general.

The algorithm generated for total completion time, IR_{sttct}, is quite different from IR_{stms} as it is composed of three nested ILS. The algorithm outline is shown in Algorithms 4, 5, and 6, while the parameters are listed in Table 5. The innermost ILS, IR_{sttct3}, uses a *VND* as local search and stops either when it cannot improve anymore or after n iterations; additionally, it uses the IG_{lsps} perturbation while always accepting the perturbed solution. The second level ILS instead, use the simple *IG* perturbation with a Metropolis like acceptance criterion. Finally the outer layer ILS also use the *IG* perturbation but with a lower value for the d parameter. Considering the acceptance criteria of the different layers, such complicated structure may be explained as a way to vary the strength and type of perturbation during the execution. The comparison with IG_{rs} is shown in Table 4 and Fig. 3. Overall, IR_{sttct} outperforms IG_{rs} with results that are always statistically significant.

5.3. Total tardiness

The algorithm generated for this problem, IR_{sttt} shown in Algorithms 7 and 8, is a two layers ILS that uses the *NEH* heuristic to generate the initial solution. The most interesting feature of this algorithm is that the inner layer employs, alternatively, two local searches. One explores the *finsert* neighborhood and the other the *twinsert* neighborhood. The inner ILS also uses the *IG* perturbation as well as the *rsacc* acceptance criterion that makes it very similar to IG_{rs}. The outer layer uses one random step in the *binsert* neighborhood as a perturbation and, similarly to the inner ILS, the *rsacc* acceptance. Comparing the temperature of the acceptance criteria, the parameter settings of IR_{sttt} are shown in Table 7. It seems that the two ILS have two well defined roles. In fact, the inner ILS has a higher probability of accepting non improving solutions while the outer layer, with a low temperature, is more focused on intensification. The results of the comparison with IG_{rs} are shown in Table 6 and in Fig. 4. Similarly to the results obtained for $PFSP^{sdst}_{TCT}$, IR_{sttt} clearly outperforms IG_{rs} and it is always statistically significantly better.

Table 8
Average RPD results of *MANEH*, *GVNS* and IR_{nims}. If an algorithm is statistically significantly better according to the Wilcoxon signed-rank test with a 95% confidence with Bonferroni correction, the result is shown in bold face.

	$t = 60$			$t = 120$			$t = 240$				$t = 60$			$t = 120$			$t = 240$		
Instances	*MANEH*	*GVNS*	IR_{nims}	*MANEH*	*GVNS*	IR_{nims}	*MANEH*	*GVNS*	IR_{nims}		*MANEH*	*GVNS*	IR_{nims}	*MANEH*	*GVNS*	IR_{nims}			
50 × 10	0.173	0.232	**0.071**	0.137	0.177	**0.026**	0.102	0.123	**-0.0004**	300 × 20	0.285	0.115	**0.058**	0.114	0.085	**0.031**	0.059	0.062	**0.019**
50 × 20	0.289	0.318	**0.177**	0.262	0.262	**0.126**	0.231	0.218	**0.093**	300 × 30	0.293	0.165	0.147	0.171	0.124	**0.100**	0.112	0.082	**0.065**
50 × 30	0.531	0.519	**0.244**	0.446	0.467	**0.156**	0.388	0.374	**0.090**	300 × 40	0.516	0.493	**0.407**	0.358	0.410	**0.281**	0.257	0.336	**0.156**
50 × 40	1.004	0.937	**0.443**	0.897	0.847	**0.261**	0.784	0.730	**0.138**	300 × 50	0.722	0.742	0.666	0.533	0.607	**0.412**	0.408	0.496	**0.239**
50 × 50	2.089	2.103	**1.263**	1.914	1.954	**1.109**	1.788	1.835	**0.973**	350 × 10	0.220	0.013	0.013	0.209	0.010	**0.009**	0.157	0.008	**0.008**
100 × 10	0.126	0.121	**0.073**	0.093	0.096	**0.053**	0.071	0.069	**0.032**	350 × 20	0.377	0.054	0.062	0.184	0.038	0.041	0.113	0.030	**0.025**
100 × 20	0.196	0.239	**0.097**	0.142	0.178	**0.050**	0.101	0.115	**0.024**	350 × 30	0.542	0.268	**0.163**	0.295	0.198	**0.116**	0.187	0.160	**0.079**
100 × 30	0.613	0.676	**0.331**	0.545	0.573	**0.197**	0.444	0.468	**0.084**	350 × 40	0.629	0.439	**0.300**	0.401	0.349	**0.205**	0.275	0.264	**0.136**
100 × 40	1.187	1.317	**0.668**	1.004	1.152	**0.391**	0.885	1.051	**0.157**	350 × 50	0.664	0.657	**0.423**	0.481	0.503	**0.234**	0.334	0.367	**0.095**
100 × 50	1.057	0.954	**0.536**	0.928	0.824	**0.352**	0.819	0.735	**0.196**	400 × 10	0.178	0.004	**0.001**	0.138	0.003	**0.000**	0.076	0.002	**0.000**
150 × 10	0.170	0.014	**0.010**	0.119	0.013	**0.009**	0.090	0.013	**0.007**	400 × 20	0.580	0.166	**0.095**	0.396	0.134	**0.062**	0.232	0.104	**0.041**
150 × 20	0.317	0.331	**0.149**	0.271	0.266	**0.083**	0.210	0.220	**0.037**	400 × 30	0.892	0.237	0.254	0.469	0.194	0.188	0.227	0.153	**0.125**
150 × 30	0.350	0.404	**0.219**	0.278	0.326	**0.151**	0.219	0.247	**0.091**	400 × 40	0.742	0.303	**0.243**	0.328	0.233	**0.172**	0.194	0.185	**0.113**
150 × 40	0.831	0.886	**0.475**	0.663	0.722	**0.297**	0.560	0.598	**0.165**	400 × 50	0.837	0.657	**0.462**	0.487	0.546	**0.285**	0.365	0.431	**0.167**
150 × 50	0.904	0.806	**0.613**	0.761	0.699	**0.394**	0.663	0.588	**0.203**	450 × 10	0.293	0.007	**0.002**	0.239	0.004	**0.001**	0.179	0.002	**0.000**
200 × 10	0.251	0.004	**0.000**	0.099	0.003	**0.000**	0.034	0.002	**0.000**	450 × 20	0.951	0.098	**0.069**	0.609	0.076	**0.045**	0.256	0.056	**0.027**
200 × 20	0.385	0.205	**0.106**	0.257	0.175	**0.076**	0.154	0.146	**0.059**	450 × 30	0.760	0.348	**0.268**	0.400	0.275	**0.190**	0.263	0.212	**0.128**
200 × 30	0.446	0.397	**0.281**	0.324	0.324	**0.170**	0.246	0.261	**0.112**	450 × 40	0.803	0.373	**0.283**	0.393	0.281	**0.206**	0.196	0.211	**0.118**
200 × 40	0.767	0.879	**0.626**	0.645	0.726	**0.406**	0.532	0.618	**0.211**	450 × 50	0.971	0.559	**0.506**	0.492	0.441	**0.311**	0.324	0.343	**0.170**
200 × 50	0.798	0.777	**0.591**	0.617	0.657	**0.407**	0.493	0.557	**0.280**	500 × 10	0.232	0.011	**0.002**	0.198	0.008	**0.000**	0.164	0.006	**0.000**
250 × 10	0.212	0.009	**0.004**	0.165	0.007	**0.002**	0.125	0.005	**0.001**	500 × 20	0.538	0.071	**0.023**	0.369	0.054	**0.017**	0.166	0.045	**0.011**
250 × 20	0.568	0.204	**0.157**	0.267	0.170	**0.113**	0.135	0.145	**0.078**	500 × 30	0.700	0.158	0.151	0.505	0.129	0.104	0.284	0.109	**0.070**
250 × 30	0.602	0.353	**0.293**	0.329	0.284	**0.223**	0.249	0.228	**0.162**	500 × 40	1.041	0.299	0.370	0.708	0.222	0.254	0.362	0.169	0.160
250 × 40	0.518	0.563	**0.390**	0.378	0.460	**0.282**	0.284	0.370	**0.166**	500 × 50	0.797	0.324	0.338	0.321	0.250	**0.203**	0.191	0.187	**0.116**
250 × 50	1.161	1.149	**1.025**	0.865	0.939	**0.737**	0.711	0.775	**0.521**										
300 × 10	0.195	0.003	0.003	0.171	0.002	0.003	0.142	0.001	0.002	Average	0.606	0.419	**0.283**	0.428	0.350	**0.191**	0.317	0.290	**0.119**

```
Output The best solution found π*,
k := 1
π := NAG()
π := VND(π, local minima, exchange, transpose, transpose)
π* := π
while ! time is over do
    π' := shake(π, k, vr_move(finsert, transpose), vr_move(binsert, finsert))
    π' := VND(π', local minima, exchange, transpose, transpose)
    π := neighchange(karacc(π, π'), k)
    if f(π') < f(π*) then
        π* := π'
    end if
end while
Return π*
```

Algorithm 11. IR_{nitct}.

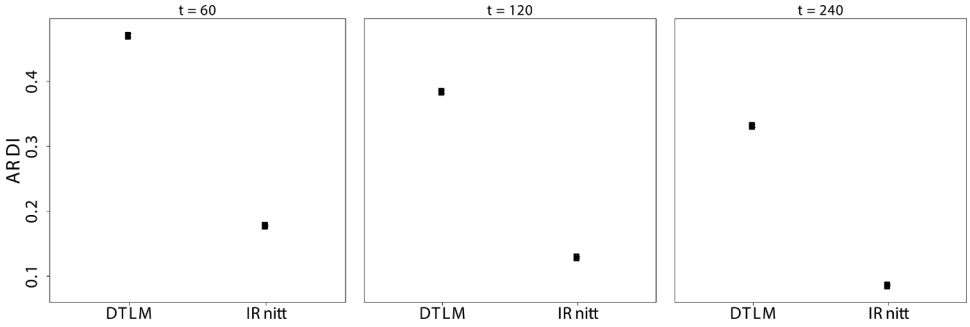

Fig. 6. Average RPD and 95% confidence intervals of *VigDE* and IR_{nitct} for $t = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

6. Results for no-idle PFSP

6.1. Makespan

Among the different algorithms proposed for this problem, GVNS [61] has been the state of the art for a long time. Recently, a memetic algorithm, MANEH [55] has shown to outperform GVNS. Since MANEH uses a modified version of GVNS to improve the best individual of the population, both algorithms are used in the comparison with IR_{nims}, the generated algorithm.

The GVNS algorithm [61] is a variable neighborhood search where the neighborhood structures used in the VND are an IG algorithm and an ILS. Both SLS algorithms use as local search the *iRZ* algorithm. The initial solution is generated using the *NEH* heuristic and the algorithm uses a random move in the insert and in the exchange neighborhood. Finally a solution is accepted only if it improves on the current solution.

MANEH is a memetic algorithm in which the population is initialized using the NEHRS heuristic [55]. At each iteration, the parents are selected using tournament selection. The random sample crossover (RSC) is used to generate two new solutions that are first mutated using a random insert move and then improved using the *iRZ* local search. If this results in a better solution, the child replaces the worst individual in the population. After a number of steps equal to a quarter of the population size, the best individual is improved using a modified version of the GVNS algorithm, SAGVNS. In SAGVNS, the strictly improve acceptance criterion of GVNS is replaced by a fixed temperature Metropolis acceptance criterion similar to the one used in IG_{rs}.

Algorithms 9 and 10 show the outline of IR_{nims}, the automatically generated algorithm for $PFSP^{ni}_{MS}$. The parameter settings are shown in Table 9. Contrary to IR_{stms}, this algorithm is composed of two ILS. As is often the case for the makespan objective, the local search used by the innermost ILS is a *FirstImprovement* local search exploring the accelerated *insert* neighborhood. Regarding the perturbation, the innermost uses the *IG* perturbation while the outermost uses the IG_{lsps} with the same local search used in the innermost ILS. The comparison of IR_{nims} with *MANEH* and *GVNS* is shown in Fig. 5 while in Table 8 the results are grouped by instance size. IR_{nims} clearly stands out as the best algorithm, outperforming both *MANEH* and *GVNS* with *GVNS* being better than *MANEH*. Overall IR_{nims} is statistically significantly better than *MANEH* and *GVNS*. However, the results grouped by instance size *GVNS* shows better results when $t = 60$ for instances of size $\{300 \times 20\}$, $\{400 \times 30\}$, $\{500 \times 40\}$, $\{500 \times 50\}$. The results are not statistically significant and, for longer running times, IR_{nims} has better performances.

6.2. Total completion time

The current state-of-the-art algorithm for $PFSP^{ni}_{TCT}$ is VigDE [62] that implements a differential evolution algorithm combined with an IG algorithm. Each individual of the population in the DE algorithm has two chromosomes. One represents the solution and the other represents two parameters of the IG algorithm, that is, the probability p of applying the local search and d, the number of jobs destroyed in the destruction-construction perturbation. The algorithm uses the NEH_{rs} heuristic to generate the initial population. At each iteration, all individuals undergo a mutation and each mutated individual is refined using the IG algorithm with probability p. The mutated individual replace the parent, if it has a better objective function value. In the experiments, we used the version of *VigDE* presented in [40], where the algorithm has been improved by using the *NAG* heuristic to generate the first individual of the population.

The best algorithm selected by irace, IR_{nitct}, is a VNS that uses the *NAG* heuristic to build the initial solution and a VND to improve the current solution. The algorithm is shown in Algorithm 11 and the parameters are shown in Table 11. Interestingly, the VND uses two times the *transpose* neighborhood. The shake employs two *vr_move* perturbations, the first executes seven random steps in the *finsert* neighborhood

Table 10
Average RPD results of *VigDE* and IR_{nitct}. If an algorithm is statistically significantly better according to the Wilcoxon signed-rank test with a 95% confidence, the result is shown in bold face.

	$t = 60$		$t = 120$		$t = 240$							
Instances	*VigDE*	IR_{nitct}	*VigDE*	IR_{nitct}	*VigDE*	IR_{nitct}						
50 × 10	3.78	**1.36**	3.29	**1.04**	2.78	**0.76**	300 × 20	6.88	**1.54**	4.86	**1.16**	4.39	**0.87**
50 × 20	3.57	**1.24**	2.96	**0.94**	2.36	**0.69**	300 × 30	6.09	**1.61**	4.21	**1.18**	3.75	**0.92**
50 × 30	3.75	**1.74**	3.11	**1.42**	2.47	**1.09**	300 × 40	8.26	**1.86**	5.11	**1.35**	4.54	**0.95**
50 × 40	3.51	**1.54**	2.74	**1.31**	2.14	**1.03**	300 × 50	9.85	**2.20**	5.50	**1.56**	4.99	**1.01**
50 × 50	3.51	**1.78**	2.79	**1.52**	2.05	**1.23**	350 × 10	6.45	**0.95**	4.52	**0.73**	4.03	**0.51**
100 × 10	4.44	**1.55**	4.03	**1.12**	3.70	**0.75**	350 × 20	7.75	**1.23**	4.44	**0.97**	3.97	**0.70**
100 × 20	4.29	**1.42**	3.83	**0.98**	3.45	**0.64**	350 × 30	8.40	**1.48**	4.54	**1.12**	3.93	**0.78**
100 × 30	5.73	**2.04**	5.16	**1.51**	4.70	**1.10**	350 × 40	9.30	**1.75**	5.25	**1.33**	4.54	**0.89**
100 × 40	6.61	**2.54**	6.08	**2.02**	5.47	**1.52**	350 × 50	9.85	**2.04**	5.43	**1.51**	4.46	**1.02**
100 × 50	5.71	**2.00**	5.21	**1.56**	4.71	**1.20**	400 × 10	6.94	**0.85**	5.60	**0.65**	4.26	**0.41**
150 × 10	4.04	**1.21**	3.74	**0.95**	3.47	**0.73**	400 × 20	8.22	**1.31**	6.02	**0.99**	4.41	**0.68**
150 × 20	5.21	**1.89**	4.83	**1.41**	4.41	**1.00**	400 × 30	8.75	**1.38**	5.57	**0.98**	4.39	**0.63**
150 × 30	5.14	**1.92**	4.77	**1.50**	4.42	**1.10**	400 × 40	8.24	**1.58**	6.92	**1.13**	4.09	**0.78**
150 × 40	6.15	**1.96**	5.63	**1.49**	5.08	**1.02**	400 × 50	9.89	**1.74**	8.77	**1.22**	4.70	**0.81**
150 × 50	6.39	**2.50**	5.93	**1.90**	5.49	**1.42**	450 × 10	7.20	**0.85**	6.89	**0.70**	4.85	**0.54**
200 × 10	4.45	**1.18**	4.10	**0.96**	3.85	**0.72**	450 × 20	8.71	**1.35**	8.33	**1.06**	4.77	**0.75**
200 × 20	5.18	**1.43**	4.74	**1.10**	4.42	**0.81**	450 × 30	10.07	**1.78**	9.56	**1.17**	5.21	**0.78**
200 × 30	5.27	**1.64**	4.82	**1.28**	4.48	**0.86**	450 × 40	9.44	**1.77**	9.17	**1.27**	5.03	**0.84**
200 × 40	6.04	**2.18**	5.45	**1.71**	5.08	**1.23**	450 × 50	9.70	**1.97**	9.34	**1.39**	4.93	**0.95**
200 × 50	5.68	**2.02**	5.24	**1.53**	4.88	**0.98**	500 × 10	7.51	**0.77**	7.19	**0.62**	5.00	**0.45**
250 × 10	4.93	**0.86**	4.44	**0.66**	4.16	**0.49**	500 × 20	7.87	**1.05**	7.51	**0.76**	4.54	**0.49**
250 × 20	5.11	**1.38**	4.48	**1.03**	4.08	**0.78**	500 × 30	8.34	**1.55**	7.97	**1.07**	4.82	**0.73**
250 × 30	5.25	**1.72**	4.62	**1.31**	4.23	**0.97**	500 × 40	9.27	**1.90**	8.83	**1.38**	5.57	**0.90**
250 × 40	6.24	**1.89**	5.59	**1.39**	5.15	**1.02**	500 × 50	9.90	**1.85**	9.37	**1.31**	5.61	**0.84**
250 × 50	7.11	**2.32**	6.38	**1.53**	5.76	**1.06**							
300 × 10	6.66	**1.19**	4.53	**0.90**	4.21	**0.65**	Average	6.73	**1.62**	5.59	**1.21**	4.36	**0.86**

and, after five iterations, it passes it to the *transpose* neighborhood. The second *vr_move* keeps the same number of iterations but reduces the random steps to one and uses a different set of neighborhoods, *binsert* and *finsert*. As neighborhood change, the *karacc* acceptance criterion has been chosen.

The results of the comparison between *VigDE* and IR_{nitct} are shown in Fig. 6 and Table 10. IR_{nitct} clearly outperforms *VigDE* in all instances sizes and the results are always statistically significant.

6.3. Total tardiness

The current state-of-the-art algorithm for $PFSP_T^{sdst}$ is DTLM [56]. DTLM is a discrete teaching learning algorithm. This population based algorithm divides the population in three groups. The best solution is considered the teacher while the best $\lambda \times PS$ individuals are elite learners, where PS is the population size and λ is a parameter of the algorithm. At each iteration, each individual undergoes two updating phases in which the produced solution substitutes the individual if it was better. In the first phase a new solution is built by applying the PMX crossover to each solution with a consensus permutation built from the elite learners. In the second phase the three groups are updated differently. An ILS algorithm is applied to the first elite learner while the other elite learners undergo a crossover with the first. A path-relinking procedure is applied to each non elite individual in which all the solutions from the individual to a random elite individual are considered and the best one is selected. If no new solutions are produced, a destruct-construct perturbation is applied.

The algorithm automatically generated for the total tardiness objective, IR_{nitt}, is shown in Algorithm 12 and 13 with the parameters shown in Table 13. The algorithm is again a two layers ILS that uses the

Table 11
Parameter settings for IR_{nitct} .

	Component	Parameter	Value
IR_{nitct}			
	NAG	x	10
		y	20
	vr_move	*num*	7
		it	5
	vr_move	*num*	1
		it	5
	psa	T_s	3.5859
		T_e	0.0764
		β	0.076
		it	336

SLACK heuristic to generate the initial solution. The innermost ILS uses a *FirstImprovement* local search that explores the exchange neighborhood. The perturbation consists of random steps in the *binsert* and *finsert* neighborhoods while the acceptance criterion is based on a fixed temperature Metropolis condition. The external ILS instead, uses a *IG* perturbation and an acceptance criterion that accepts only improving solutions. The results of the experiments are shown in Table 12 and in Fig. 7. IR_{nitt} greatly outperforms *DTLM*.

7. Discussion and conclusions

In this paper, a *AAD* system has been applied to generate SLS algorithms to solve $PFSP^{ni}$ and $PFSP^{sdst}$ with the objectives of minimizing makespan, sum of completion time and total tardiness. The generated algorithms outperformed the state-of-the-art algorithms. The fact that these problems were less studied compared with standard PFSP may explain the huge performance difference registered between the algorithms generated trough our system and the competing human generated ones. In any case, the observed results together with the ones

1: **Output** The best solution found π^*,
2: π := *SLACK*()
3: π := IR_{nitt} 2(π)
4: π^* := π
5: **while** ! time is over **do**
6: π' := *IG*(π)
7: π' := IR_{nitt} 2(π')
8: π := *improve*(π, π')
9: **if** $f(\pi') < f(\pi^*)$ **then**
10: π^* := π'
11: **end if**
12: **end while**
13: **Return** π^*

Algorithm 12. IR_{nitt}.

1: **Input** current solution π.
2: **Output** The best solution found π^*,
3: π := *FirstImprovement*(π, *local minima*, *exchange*)
4: π^* := π
5: **while** *maxsteps*() **do**
6: π' := *vr_move*(π, *binsert*, *finsert*))
7: π' := *FirstImprovement*(π', *local minima*, *exchange*)
8: π := *karacc*(π, π')
9: **if** $f(\pi') < f(\pi^*)$ **then**
10: π^* := π'
11: **end if**
12: **end while**
13: **Return** π^*

Algorithm 13. IR_{nitt2}.

Table 13
Parameter settings for IR_{nitt} .

	Component	Parameter	Value		Component	Parameter	Value
IR_{nitt}				IR_{nitt2}			
	IG	*d*	10				
					maxsteps	*maxi*	134
					vr_move	*num*	1
						it	9
					karacc	T_p	3.3172

Table 12
Average RDI results of *DTLM* and IR_{nitt}. If an algorithm is statistically significantly better according to the Wilcoxon signed-rank test with a 95% confidence, the result is shown in bold face.

	$t = 60$		$t = 120$		$t = 240$	
Instances	*DTLM*	IR_{nitt}	*DTLM*	IR_{nitt}	*DTLM*	IR_{nitt}
50 × 10	0.42	**0.15**	0.37	**0.11**	0.32	**0.07**
50 × 30	0.39	**0.18**	0.36	**0.13**	0.31	**0.09**
50 × 50	0.40	**0.16**	0.34	**0.12**	0.29	**0.08**
150 × 10	0.36	**0.15**	0.32	**0.11**	0.27	**0.07**
150 × 30	0.43	**0.18**	0.40	**0.13**	0.35	**0.08**
150 × 50	0.41	**0.19**	0.39	**0.14**	0.33	**0.09**
250 × 10	0.47	**0.18**	0.40	**0.13**	0.34	**0.09**
250 × 30	0.45	**0.16**	0.37	**0.12**	0.32	**0.08**
250 × 50	0.41	**0.19**	0.39	**0.13**	0.33	**0.08**
350 × 10	0.43	**0.20**	0.40	**0.15**	0.35	**0.10**
350 × 30	0.48	**0.19**	0.42	**0.14**	0.36	**0.09**
350 × 50	0.46	**0.19**	0.40	**0.14**	0.35	**0.10**
Average	0.42	**0.18**	0.38	**0.13**	0.33	**0.08**

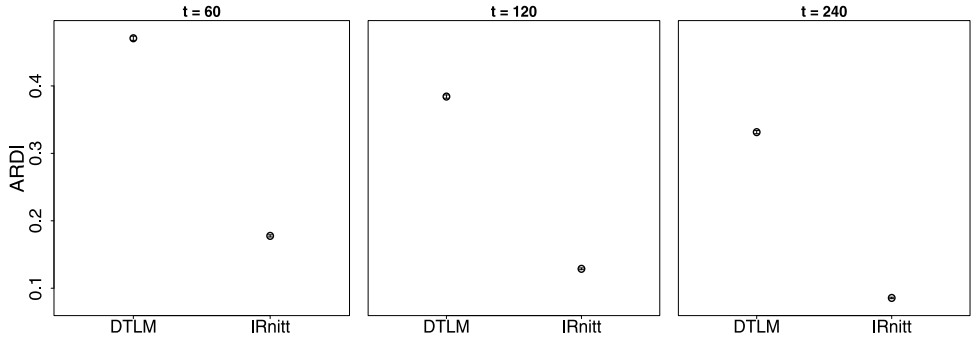

Fig. 7. Average RDI and 95% confidence intervals of *DTLM* and IR_{nitt} for $t = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

obtained for the standard PFSP [43] give further confirmation to *AAD* and this method in particular. Looking at the generated algorithms, as already observed with standard PFSP, it is interesting to note that a two levels structure is preferred in the majority of cases, being present in IR_{sttt}, IR_{sttct}, IR_{nims} and IR_{nitt}. This result is in accordance with our previous study on standard PFSP where two out of three algorithms possessed the same structure.

Interestingly, the result of *DTLM* in PFSP^{ni}_{T}, the result of *MANEH* in PFSP^{ni}_{MS} and *EMBO* in PFSP^{sdst}_{MS} show that population based algorithms seem not to be well suited to tackle the PFSP. Further confirmation of this trend can be found by considering the state-of-the-art algorithms for the standard PFSP [43,52].

Several directions can be taken to further progress this research. First, it would be of great interest to investigate whether the level of structural complexity found in generated algorithms, like IR_{sttct}, is really needed. A second direction is to use the system to generate algorithms for other problems as well as to further expand EMILI to better support other types of SLS methods such as population based algorithms. Finally, the components implemented in the EMILI framework are the result of the advance knowledge present in the literature regarding the specific problems tackled. A fourth direction to investigate would be to add to this system the ability of automatically generate new components. In particular, construction heuristics may be generated following a *AAD* process where *AAC* tools are used to generate new heuristics by combining different components.

CRediT authorship contribution statement

Federico Pagnozzi: Methodology, Investigation, Formal analysis, Software, Writing - original draft. **Thomas Stützle:** Supervision, Conceptualization, Methodology, Validation, Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and

innovation programme (grant agreement No. 681872). Thomas Stützle acknowledges support from the Belgian F.R.S.-FNRS, of which he is a Research Director.

References

[1] Baptiste P, Hguny L. A branch and bound algorithm for the f/no_idle/c_{max}. Proceedings of the international conference on industrial engineering and production management. IEPM'97, Lyon. 1997. p. 429–38.
[2] Bezerra LCT, López-Ibáñez M, Stützle T. Automatic component-wise design of multi-objective evolutionary algorithms. IEEE Trans Evol Comput 2016;20(3):403–17.
[3] Bezerra LCT, López-Ibáñez M, Stützle T. A large-scale experimental evaluation of high-performing multi- and many-objective evolutionary algorithms. Evol Comput 2018;26(4):621–56.
[4] Bezerra LCT, López-Ibáñez M, Stützle T. Automatic configuration of multi-objective optimizers and multi-objective configuration. High-performance simulation-based optimization. Cham, Switzerland: Springer International Publishing; 2020. p. 69–92.
[5] Branke J, Nguyen S, Pickardt CW, Zhang M. Automated design of production scheduling heuristics: a review. IEEE Trans Evol Comput 2016;20(1):110–24.
[6] Brum A, Ritt M. Automatic design of heuristics for minimizing the makespan in permutation flow shops. 2018 IEEE congress on evolutionary computation (CEC). Piscataway, NJ: IEEE Press; 2018. p. 1–8.
[7] Brum A, Ritt M, López-Ibáñez M. Automatic algorithm configuration for the permutation flow shop scheduling problem minimizing total completion time. In: Liefooghe A, editor. Proceedings of EvoCOP 2018 –European conference on evolutionary computation in combinatorial optimization, vol. 10782 of lecture notes in computer science. Heidelberg, Germany: Springer; 2018. p. 85–100.
[8] Burke EK, Hyde MR, Kendall G. Grammatical evolution of local search heuristics. IEEE Trans Evol Comput 2012;16(7):406–17.
[9] Burke EK, Hyde MR, Kendall G, Ochoa G, Özcan E, Woodward JR. A classification of hyper-heuristic approaches: Revisited. In: Gendreau M, Potvin JY, editors. Handbook of metaheuristics, vol. 272 of international series in operations research & management science. Springer; 2019. p. 453–77.Ch. 14
[10] Cahon S, Melab N, Talbi EG. ParadisEO: a framework for the reusable design of parallel and distributed metaheuristics. J Heuristics 2004;10(3):357–80.
[11] De Souza M, Ritt M. Automatic grammar-based design of heuristic algorithms for unconstrained binary quadratic programming. In: Liefooghe A, López-Ibáñez M, editors. Proceedings of evoCOP 2018 – 18th European conference on evolutionary computation in combinatorial optimization, vol. 10782 of lecture notes in computer science. Heidelberg, Germany: Springer; 2018. p. 67–84.
[12] De Souza M, Ritt M. An automatically designed recombination heuristic for the test-assignment problem. 2018 IEEE congress on evolutionary computation (CEC). Piscataway, NJ: IEEE Press; 2018. p. 1–8.
[13] Fernandez-Viagas V, Framiñán JM. On insertion tie-breaking rules in heuristics for the permutation flowshop scheduling problem. Comput Oper Res 2014;45:60–7.
[14] Framiñán JM, Leisten R, Ruiz R. Manufacturing scheduling systems: an integrated view on models, methods, and tools. New York, NY: Springer; 2014.
[15] Franzin A, Stützle T. Revisiting simulated annealing: a component-based analysis. Comput Oper Res 2019;104:191–206.
[16] Fukunaga AS. Evolving local search heuristics for SAT using genetic programming. In: Deb K, editor. Proceedings of the genetic and evolutionary computation conference, GECCO 2004, Part II, vol. 3103 of lecture notes in computer science. Heidelberg, Germany: Springer; 2004. p. 483–94.
[17] Fukunaga AS. Automated discovery of local search heuristics for satisfiability testing. Evol Comput 2008;16(1):31–61.
[18] Gupta JND. Flowshop schedules with sequence dependent setup times. J Oper Res SocJpn 1986;29:206–19.
[19] Hart E, Sim K. A hyper-heuristic ensemble method for static job-shop scheduling. Evol Comput 2016;24(4):609–35.
[20] Hoos HH, Stützle T. Stochastic local search—foundations and applications. San Francisco, CA: Morgan Kaufmann Publishers; 2005.
[21] Hutter F, Hoos HH, Leyton-Brown K. Sequential model-based optimization for general algorithm configuration. In: Coello Coello CA, editor. Learning and intelligent optimization, 5th international conference, LION 5, vol. 6683 of lecture notes in computer science. Heidelberg, Germany: Springer; 2011. p. 507–23.
[22] Hutter F, Hoos HH, Stützle T. Automatic algorithm configuration based on local search. In: Holte RC, Howe A, editors. Proc. of the twenty-second conference on artifical intelligence (AAAI '07), AAAI press/MIT press, Menlo Park, CA; 2007. p. 1152–7.
[23] Johnson DS. Optimal two- and three-stage production scheduling with setup times included. Nav Res Logist Q 1954;1:61–8.
[24] Karabulut K. A hybrid iterated greedy algorithm for total tardiness minimization in permutation flowshops. Comput Ind Eng 2016;98(Supplement C):300–7.
[25] Keller RE, Poli R. Linear genetic programming of parsimonious metaheuristics. 2007 IEEE Congress on evolutionary computation. 2007. p. 4508–15.
[26] Keller RE, Poli R. Cost-benefit investigation of a genetic-programming hyperheuristic. In: Monmarché N, Talbi E-G, Collet P, Schoenauer M, Lutton E, editors. Artificial evolution. Berlin/Heidelberg, Berlin, Heidelberg: Springer; 2008. p. 13–24.
[27] KhudaBukhsh AR, Xu L, Hoos HH, Leyton-Brown K. SATenstein: automatically building local search SAT solvers from components. Artif Intell 2016;232:20–42.
[28] Kim YD. Heuristics for flowshop scheduling problems minimizing mean tardiness. J Oper Res Soc 1993;44(1):19–28.
[29] Koza J. Genetic programming: on the programming of computers by the means of natural selection. Cambridge, MA: MIT Press; 1992.
[30] Li X, Zhang Y. Adaptive hybrid algorithms for the sequence-dependent setup time permutation flow shop scheduling problem. IEEE Trans Autom SciEng 2012;9(3):578–95.
[31] Liu J, Reeves CR. Constructive and composite heuristic solutions to the P//ΣCi scheduling problem. Eur J Oper Res 2001;132(2):439–52.
[32] López-Ibáñez M, Stützle T. The automatic design of multi-objective ant colony optimization algorithms. IEEE Trans Evol Comput 2012;16(6):861–75.
[33] López-Camacho E, Terashima-Marin H, Ross P, Ochoa G. A unified hyper-heuristic framework for solving bin packing problems. Expert Syst Appl 2014;41(15):6876–89.
[34] López-Ibáñez M, Dubois-Lacoste J, Pérez Cáceres L, Stützle T, Birattari M. The irace package: iterated racing for automatic algorithm configuration. Oper Res Perspect 2016;3:43–58.
[35] López-Ibáñez, M., Kessaci, M.-E., & Stützle, T.. Automatic design of hybrid metaheuristics from algorithmic components, submitted.
[36] Marmion M-E, Mascia F, López-Ibáñez M, Stützle T. Automatic design of hybrid stochastic local search algorithms. In: Blesa MJ, Blum C, Festa P, Roli A, Sampels M, editors. Hybrid metaheuristics, vol. 7919 of lecture notes in computer science. Heidelberg, Germany: Springer; 2013. p. 144–58.
[37] Mascia F, López-Ibáñez M, Dubois-Lacoste J, Stützle T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Comput Oper Res 2014;51:190–9.
[38] Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller A, Teller E. Equation of state calculations by fast computing machines. J Chem Phys 1953;21:1087–92.
[39] Minella G, Ruiz R, Ciavotta M. A review and evaluation of multiobjective algorithms for the flowshop scheduling problem. INFORMS J Comput 2008;20(3):451–71.
[40] Nagano MS, Rossi FL, Martarelli NJ. High-performing heuristics to minimize flowtime in no-idle permutation flowshop. Eng Optim 2019;51(2):185–98.
[41] Nawaz M, Enscore Jr E, Ham I. A heuristic algorithm for the *m*-machine, *n*-job flow-shop sequencing problem. Omega 1983;11(1):91–5.
[42] O'Neill M, Ryan C. Grammatical evolution. IEEE Trans Evol Comput 2001;5(4):349–58.
[43] Pagnozzi F, Stützle T. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. Eur J Oper Res 2019;276:409–21.
[44] Pagnozzi, F., & Stützle, T. (2019b). Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems with additional constraints: Supplementary material. http://iridia.ulb.ac.be/supp/IridiaSupp2019-007/.
[45] Pan Q-K, Ruiz R. An effective iterated greedy algorithm for the mixed no-idle permutation flowshop scheduling problem. Omega 2014;44:41–50.
[46] Pinedo ML. Scheduling: theory, algorithms, and systems, 4th edition. New York, NY: Springer; 2012.
[47] Rad SF, Ruiz R, Boroojerdian N. New high performing heuristics for minimizing makespan in permutation flowshops. Omega 2009;37(2):331–45.
[48] Rajendran C, Ziegler H. An efficient heuristic for scheduling in a flowshop to minimize total weighted flowtime of jobs. Eur J Oper Res 1997;103(1):129–38.
[49] Ruiz R, Maroto C. A comprehensive review and evaluation of permutation flowshop heuristics. Eur J Oper Res 2005;165(2):479–94.
[50] Ruiz R, Maroto C. A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility. Eur J Oper Res 2006;169(3):781–800.
[51] Ruiz R, Stützle T. An Iterated Greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. Eur J Oper Res 2008;187(3):1143–59.
[52] Ruiz R, Stützle T. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. Eur J Oper Res 2007;177(3):2033–49.
[53] Ruiz R, Vallada E, Fernández-Martínez C. Scheduling in flowshops with no-idle machines. Computational intelligence in flow shop and job shop scheduling. Springer; 2009. p. 21–51.
[54] Sabar NR, Ayob M, Kendall G, Qu R. Grammatical evolution hyper-heuristic for combinatorial optimization problems. IEEE Trans Evol Comput 2013;17(6):840–61.
[55] Shao W, Pi D, Shao Z. Memetic algorithm with node and edge histogram for no-idle flow shop scheduling problem to minimize the makespan criterion. Appl Soft Comput 2017;54:164–82.
[56] Shao W, Pi D, Shao Z. A hybrid discrete teaching-learning based meta-heuristic for solving no-idle flow shop scheduling problem with total tardiness criterion. Comput Oper Res 2018;94:89–105.
[57] Sim K, Hart E, Paechter B. A lifelong learning hyper-heuristic method for bin packing. Evol Comput 2015;23(1):37–67.

[58] Sioud A, Gagné C. Enhanced migrating birds optimization algorithm for the permutation flow shop problem with sequence dependent setup times. Eur J Oper Res 2018;264(1):66–73.
[59] Stützle T, López-Ibáñez M. Automated design of metaheuristic algorithms. In: Gendreau M, Potvin JY, editors. Handbook of metaheuristics, Vol. 272 of international series in operations research & management science. Springer; 2019. p. 541–79.
[60] Taillard ED. Benchmarks for basic scheduling problems. Eur J Oper Res 1993;64 (2):278–85.
[61] Tasgetiren MF, Buyukdagli O, Pan Q-K, Suganthan PN. A general variable neighborhood search algorithm for the no-idle permutation flowshop scheduling problem. In: Panigrahi BK, Suganthan PN, Das S, Dash SS, editors. Swarm, evolutionary, and memetic computing, vol. 8298 of theoretical computer science and general issues, springer international publishing; 2013. p. 24–34.
[62] Tasgetiren MF, Pan Q-K, Suganthan PN, Buyukdagli O. A variable iterated greedy algorithm with differential evolution for the no-idle permutation flowshop scheduling problem. Comput Oper Res 2013;40(7):1729–43.
[63] Tavares J, Pereira FB. Automatic design of ant algorithms with grammatical evolution. In: Moraglio A, Silva S, Krawiec K, Machado P, Cotta C, editors. Proceedings of the 15th European conference on genetic programming, euroGP 2012, vol. 7244 of lecture notes in computer science. Heidelberg, Germany: Springer; 2012. p. 206–17.
[64] Vallada E, Ruiz R, Minella G. Minimising total tardiness in the m-machine flowshop problem: a review and evaluation of heuristics and metaheuristics. Comput Oper Res 2008;35(4):1350–73.
[65] Wang Y, Dong X, Chen P, Lin Y. Iterated local search algorithms for the sequence-dependent setup times flow shop scheduling problem minimizing makespan. Foundations of intelligent systems. Springer; 2014. p. 329–38.