Automatic Design of Hybrid Stochastic Local Search Algorithms for Permutation Flowshop Problems

Federico Pagnozzi*, Thomas Stützle

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

Abstract

Stochastic local search methods are at the core of many effective heuristics for tackling different permutation flowshop problems (PFSPs). Usually, such algorithms require a careful, manual algorithm engineering effort to reach high performance. An alternative to the manual algorithm engineering is the automated design of effective SLS algorithms through building flexible algorithm frameworks and using automatic algorithm configuration techniques to instantiate high-performing algorithms. In this paper, we automatically generate new high-performing algorithms for some of the most widely studied variants of the PFSP. More in detail, we (i) developed a new algorithm framework, EMILI, that implements algorithm-specific and problem-specific building blocks; (ii) define the rules of how to compose algorithms from the building blocks; and (iii) employ an automatic algorithm configuration tool to search for high performing algorithm configurations. With these ingredients, we automatically generate algorithms for the PFSP with the objectives makespan, total completion time and total tardiness, which outperform the best algorithms obtained by a manual algorithm engineering process.

Keywords: Scheduling, stochastic local search, automatic algorithm design

*Corresponding author.

Email addresses: `federico.pagnozzi@ulb.ac.be` (Federico Pagnozzi), `stuetzle@ulb.ac.be` (Thomas Stützle)

1. Introduction

Permutation flow-shop problems (PFSPs) are one of the most widely studied classes of scheduling problems (Framiñán et al., 2014), the arguably most studied variant being the one to minimize a schedule's makespan (Fernandez-Viagas et al., 2017). Other widely studied variants include those that consider minimizing the sum of completion times of the jobs (Pan and Ruiz, 2012) or the sum of the jobs' tardiness if due dates are considered. As these variants (with few exceptions such as the two machine case for makespan minimization (Johnson, 1954)) are NP-hard, much of the research on these problems has focused on heuristic and metaheuristic algorithms. In fact, over the years a large number of high-performing algorithms have been proposed for the above cited variants (Fernandez-Viagas et al., 2017; Framiñán et al., 2014; Pan and Ruiz, 2012), often obtained after a significant, manual algorithm engineering effort.

In this paper, we show that for the PFSP variants with makespan, sum completion time and total tardiness objectives we can generate automatically new state-of-the-art algorithms from a same code-base and without human intervention in the algorithm design process. The main ingredients that we use for this design process is a flexible algorithm framework from which a set of pre-programmed algorithmic components can be combined to generate algorithms, a coherent way of how to generate stochastic local search (SLS) algorithms from the framework and automatic algorithm configuration tools.

The flexible algorithm framework we use is called EMILI for Easily Modifiable Iterated Local search Implementation and it builds on the ideas proposed in our earlier research on similar topics (Marmion et al., 2013; López-Ibáñez et al., 2017). EMILI is designed with the aim to support the automated design of hybrid SLS algorithms. It implements generic algorithm components required, for example, to instantiate a number of different SLS methods (aka metaheuristics) but also basic local search methods such as iterative improvement algorithms. In addition, it allows us also to include into the framework problem-specific algorithm components that stem from known algorithms, exploiting in this way the vast knowledge on specific algorithm components available in the literature. For the automatic composition of new algorithms from the EMILI framework we exploit the approach proposed in earlier work (Marmion et al., 2013; López-Ibáñez et al., 2017), which is based on the encoding of possible combinations of algorithm components using a grammatical representation, translating this encoding into a parametric form (as explored by Mascia et al. (2014)) and the exploitation of automatic algorithm configuration techniques such as irace (López-Ibáñez et al., 2016).

With these ingredients, we generate SLS algorithms for three of the most studied PFSP objectives: the minimization of (i) the makespan, PFSP$_{MS}$ (Fernandez-Viagas et al., 2017); (ii) the sum of completion times, PFSP$_{TCT}$ (Pan and Ruiz, 2012) and (iii) the total tardiness, PFSP$_{TT}$ (Li et al., 2015; Karabulut, 2016). A comparison of our automatically generated algorithms to the state of the art for each objective shows that in all cases our algorithms are clear improvements. Thus, our results indicate a new way of how to generate high-performing algorithms for a set of scheduling problems, which so far have been tackled by extensive, manual algorithm engineering efforts.

The paper is organized as follows. Section 2 recalls some basics about the PFSPs we tackle. In Section 3 we describe our methodology and present the experimental results in Section 4. In Section 5 we summarize the results and outline future directions for our work.

2. Scheduling Problems

The basic version of the flowshop problem can be described as follows. A set of n jobs that have to be processed on m machines. Each job J_i consists of (at most) m operations, where each operation has a non-negative processing time p_{ij} on machine M_j. All jobs are released at time zero and they must be processed on the machines $M_1, M_2, \ldots, M_m$ in the same canonical order. A common restriction is that job-passing is not allowed, that is, all jobs are processed on all machines in the same order. In that case, a solution can be represented as a permutation $\pi = (\pi(1), \pi(2), ...\pi(n))$ of the job indices, leading to the permutation flowshop problem (PFSP). The completion time of the job at position i on machine j is given by

$$C_{\pi(i),j} = max\{C_{\pi(i-1),j}, C_{\pi(i),j-1}\} + p_{\pi(i),j}, \quad i, j > 1 \tag{1}$$

where one has $C_{\pi(i),1} = \sum_{l=1}^{i} p_{\pi(l),1}$ and $C_{\pi(1),j} = \sum_{l=1}^{j} p_{\pi(1),l}$.

The goal of the PFSP is to find a permutation that optimizes some given objective function and the most common objective one to be minimized is the makespan, that is, the completion time $C_{\pi(n),m}$ of the last job on the last machine, commonly denoted as $C_{\max}$. The PFSP with makespan objective (PFSP$_{MS}$) is one of the most widely tackled scheduling problems in the literature and for a recent review of solution approaches to it we refer to Fernandez-Viagas et al. (2017).

Another, common objective is to minimize the sum of completion times

$$TCT = \sum_{i=1}^{n} C_{i,m} \tag{2}$$

of all jobs and, by doing so, trying to minimize the occupation time of the machines. This objective is also called total completion time and when the release times of all jobs are equal to zero, it is equivalent to minimizing the jobs' flowtime. We denote this problem as PFSP$_{TCT}$ (Pan and Ruiz, 2012).

In this article, we also tackle the PFSP with the objective of minimizing the total tardiness, which is defined as

$$TT = \sum_{i=1}^{n} max\{C_{i,m} - d_{\pi(i)}, 0\} \tag{3}$$

where $d_{\pi(i)}$ is the due date of job $\pi(i)$ and, hence, the tardiness $\{C_{i,m} - d_{\pi(i)}, 0\}$ of a job measures how long a job is finished after its due date. We denote this variant by PFSP$_{TT}$ in what follows (Hasija and Rajendran, 2004; Li et al., 2015; Karabulut, 2016). Some indication on the state-of-the-art methods for these three problems are given in Section 4.

3. Automated SLS algorithm design with EMILI

3.1. High-level view of EMILI

EMILI is an algorithm framework that was built to support the automatic design of (hybrid) SLS algorithms. Following Hoos and Stützle (2005), here we understand hybrid SLS algorithms as those that manipulate at each search step a single solution and combine two or more different types of search steps, that is, ways of modifying candidate solutions. For convenience, within the EMILI framework we also consider SLS algorithms that use only a single type of search step, which in Hoos and Stützle (2005) are called “simple” SLS algorithms. Types of SLS algorithms that can be generated from EMILI comprise simulated annealing, iterated local serch, variable neighborhood search, iterated greedy, GRASP, tabu search, probabilistic iterative improvement and combinations of these.

EMILI is based on (i) a decomposition of SLS algorithms into algorithmic components, (ii) an algorithm template from which many different types of SLS methods can be instantiated, (iii) a recursive definition of possible algorithm compositions that in turn allow to generate hybrid algorithms, and (iv) a strict separation between algorithm-related components and problem-related components. EMILI is a significant refinement of the initial proposal from Marmion et al. (2013); López-Ibáñez et al. (2017) in terms of ease of implementation and algorithm composition, the comprehensiveness of the implemented components, and the possibility of tackling problem classes rather than single problems.

From a high-level perspective, the algorithms that may be instantiated by EMILI follow the structure of an iterated local search (ILS) (Lourenço et al., 2010) as depicted in Algorithm 1. After generating an initial solution (line 2) and possibly improving it by some SLS algorithm (line 3), the main loop (lines 4 to 8) is invoked, where first a solution may be perturbed (line 5), an SLS algorithm may be applied to improve it (line 6), and an acceptance criterion decides whether to accept a new solution or not (line 7). (Note that the outline doesn’t make explicit keeping track of the best solution found so far, but this is of course done.) As explained in Marmion et al. (2013) and in López-Ibáñez et al.

Algorithm 1 ILS

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

(2017), from this structure one can instantiate a number of SLS methods other than ILS. For example, a simulated annealing is obtained by choosing as a perturbation a random solution in some neighborhood, not applying any SLS algorithm in SLS, and accepting a new solution according to the Metropolis condition or similar acceptance criteria known from simulated annealing.

The algorithmic components that EMILI offers can be classified as follows. The first class comprises components that do not manipulate candidate solutions by themselves and, hence, can be re-used for all problems tackled with EMILI. This comprises generic components such as termination criteria or acceptance criteria. Of course, these components may need to know information such as an evaluation function value, but otherwise are independent of a specific problem. A second class comprises algorithmic components that depend on the solution representation but that are otherwise generic. This includes representation-specific ways of generating solutions, neighborhoods, or perturbations. Considering the permutation-representation we use in this paper, these include (i) components for generating initial candidate solutions, for example, by generating random permutations or generic insertion heuristics, (ii) generic neighborhoods such as transpose, exchange, and insert ones and (iii) perturbation moves that correspond to compositions of k random moves in the aforementioned neighborhoods. Finally, we have also the possibility of including problem-specific components that include problem-specific constructive heuristics or problem-specific local search and speed-up techniques. Given the problem-independent components and the representation-specific ones we have currently implemented, it is possible for a new problem to run a simple ILS or using the whole automated design machinery explained next by only defining basic problem-specific routines such as reading instances, indicating the pre-defined solution representation to be used and computing the objective or evaluation function.

3.2. Automatic design of SLS algorithms with EMILI

To automatically design hybrid SLS algorithms, EMILI uses a representation of possible algorithm compositions as grammars. A grammar uses a set of rules that define how to build sentences in a defined language. In our case, the grammar is defined so that a legal sentence represents a valid algorithm that is an algorithm that can be instantiated and run. This limits the search space that an automatic configuration has to explore to only valid algorithm configurations, as combinations of components that do not represent a valid algorithm (e.g. trying to use a perturbation as a termination criterion) cannot be produced. This property has made grammars an attractive option to implicitly define algorithm spaces (Burke et al., 2012; Mckay et al., 2010). Instead of directly instantiating grammar rules, we adopt the approach proposed by Mascia et al. (2014), which allows to convert the grammar rules into a finite set of parameters. There are two advantages by doing so. The first one is that it allows the exploitation of standard automatic algorithm configuration (AAC) tools such as SMAC (Hutter et al., 2011) or irace (López-Ibáñez et al., 2016). The second is that numerical parameters may be configured directly by the AAC tools without needing to represent them as derivations in a grammar. The conversion of the grammar to parameters is a rather straightforward step if recursive grammar rules are cut after a specific number of recursions.

While the overall approach we follow has been proposed before, the EMILI framework presents a new implementation of these ideas and simplifies the previous approach used in Marmion et al. (2013) and in López-Ibáñez et al. (2017). One simplification is to replace the dependence on the ParadisEO framework (Cahon et al., 2004) by an implementation at a level of detail that better supports composability of algorithm components, providing better support for automated algorithm design. Overall, it should be emphasized that the approach we follow here is also different from other recent efforts at automating algorithm design. In fact, related efforts are focused mainly on fixed algorithm templates where alternative algorithm compositions consist of alternative choices for specific tasks within the given template. Examples of such approaches comprise (simple) SLS algorithms for the satisfiability problem in propositional logic (KhudaBukhsh et al., 2009), frameworks for multi-objective ACO algorithms (López-Ibáñez and Stützle, 2012), or ACO algorithms for continuous optimization (Liao et al., 2014).

Let us illustrate the grammar-based representation of algorithm compositions with a small example. Consider a rule for deriving an ILS algorithm

$$\text{<ILS> ::= `ils' <LocalSearch> <Termination> <Perturbation> <Acceptance>} \tag{4}$$

which says that to derive an ILS algorithm, one needs to instantiate the components LocalSearch, Termination, Perturbation, and Acceptance. For example, a specific ILS algorithm for the PFSP could be instantiated by

$$ils_{pfsp} ::= \text{`ils'}\ ls_{pfsp}\ \text{`time 30'}\ \text{`}\textit{random move exchange}\ 5\text{'}\ \text{`}\textit{better}\text{'} \tag{5}$$

This algorithm describes an *ILS* algorithm that uses a specific local search algorithm called ils_{pfsp}, stops after 30 seconds, uses a perturbation that executes 5 random moves in the exchange neighbordhood and accepts only improving solutions. Moreover, using a general representation like the one in 4, allows us to define not only an ILS but also other different types of SLS algorithms. A snapshot of the grammar used for the high-level algorithmic part in this paper is given in Figure 1.

In order to design algorithms with AAC tools, the grammar rules have to be transformed into parameters. Simple rules, like choosing among alternatives (e.g. a neighborhood) or specifying a number (e.g. setting the number of seconds before stopping an algorithm) require only one parameter, respectively categorical and numerical. Rules that are composed

<LocalSearch>	::=	<FirstImprovement> | <BestImprovement> | <TabuSearch> |
		<VND> | <ILS> | <EmptyLocalSearch>
<FirstImprovement>	::=	'first' <InitialSolution> <Termination> <Neighborhood>
<BestImprovement>	::=	'best' <InitialSolution> <Termination> <Neighborhood>
<TabuSearch>	::=	<FirstTabuSearch> | <BestTabuSearch>
<FirstTabuSearch>	::=	'tabu' 'first' <InitialSolution>
		<Termination> <Neighborhood> <TabuTenure>
<BestTabuSearch>	::=	'tabu' 'best' <InitialSolution>
		<Termination> <Neighborhood> <TabuTenure>
<EmptyLocalSearch>	::=	'nols' <InitialSolution>
<VND>	::=	'vnd' <firstVND> | <bestVND>
<firstVND>	::=	'first' <InitialSolution> <Termination> <neighborhoods>
<bestVND>	::=	'best' <InitialSolution> <Termination> <neighborhoods>
<ILS>	::=	'ils' <LocalSearch> <Termination> <Perturbation>
		<Acceptance>
<neighborhoods>	::=	<Neighborhood> <neighborhoods> | $\emptyset$

Figure 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 hybridizations of various algorithms.

of a sequence of other rules (e.g. <FirstImprovement> in Figure 1) require a parameter for each one of the composing rules. Complex rules, that is rules that can be applied more than once, require a new parameter each time they are expanded and, consequently, each parameter connected has to be duplicated. For instance, considering Figure 1, the rule <ILS> can be expanded recursively, because it can be chosen when expanding <LocalSearch>. In this case, a new parameter has to be introduced for the second <ILS> as well as new parameters for <LocalSearch>, <Termination>, <Perturbation> and <Acceptance>. Furthermore, this kind of rules could be expanded an indefinite number of times. For this reason, we limited the maximum depth for the recursive expansion of a rule to three. We refer to Mascia et al. (2014) for more details.

In the next sections the options available for basic components such as initial solutions or acceptance criteria are explained.

3.3. Initial solution

This component considers the generation of an initial solution. We consider the possibility of using either a solution generated uniformly at random or applying one of several construction heuristics. For convenience, we describe construction heuristics by three elements: (i) a greedy function h that assigns a value to each solution component; (ii) a greedy strategy that prescribes which solution component to select depending on h; and (iii) a construction rule, which describes how a solution component is added to the partial solution. When building a solution for the PFSP, the value of h is computed for each job. Then, starting from an empty solution, the solution is built in a iterative process that, at each iteration, chooses a job according to the greedy strategy and adds it to the partial solution following the construction rule.

Let $\pi_p = (\pi(1), \pi(2), \pi(3), \ldots, \pi(k))$, $k < n$ be a partial solution. We consider two types of constructive heuristics, where a next job may either be appended to π_p or where each of the $k+1$ possible positions for inserting a job into π_p is tested and the best insertion position is picked. The former type is akin to dispatching heuristics while the latter is generally referred to as insertion heuristics. Sometimes, an iterative improvement algorithm is used to refine the solutions produced by a construction heuristic. The local search used in the heuristics implemented in our framework uses a first improvement pivoting rule and the insert neighborhood for n steps or until a local optimum is reached (see also next section).

In Table 1 we report the heuristics implemented for the generation of the initial solution. For each heuristic, the table reports the greedy function, the greedy strategy and the construction rule. In the local search column, we report if the local search post-processing is used and, if yes, the termination criterion. The last column indicates which PFSP objectives the heuristics are able to handle.

We can divide the heuristics in three main groups: the ones that use the append rule, insertion heuristics and a hybridizations between the two. This last group is obtained by using the solution found by a heuristic that uses the append rule as the starting sequence for an insertion heuristic. The *NEH* insertion heuristic (Nawaz et al., 1983), which orders the jobs depending on the sum of processing times, is one of the most effective heuristics for the PFSP$_{MS}$. Among the many improvements to this heuristics proposed in the literature, we implemented *NEH*$_{tb}$ (Fernandez-Viagas and Framiñán, 2014) and *FRB5* (Rad et al.,

2009) for PFSP$_{MS}$ and NEH_{edd} (Kim, 1993) for PFSP$_{TT}$. The first introduces a new rule to break ties if more insertion positions result in the same objective function value. The second performs a local search in the insert neighborhood on the partial solution after each job insertion. The third evaluates the jobs using the due dates and therefore is limited to the PFSP$_{TT}$. The *RZ* (Rajendran and Ziegler, 1997) heuristic was created to solve PFSP problems where each job is characterized by a priority or weight. This value is represented by g_i in the index function of the heuristic in Table 1. We adapted this insertion heuristic to solve general PFSP by assuming a weight of one when the problem does not define job priorities. The solution created by the initial sequence is improved using a local search.

NRZ and NRZ_2 are hybridizations of *RZ* with *NEH*. In the first, the solution generated by *RZ* is used to define the order in which jobs are considered for insertion. The second, NRZ_2, is an insertion heuristic that uses the same greedy function of *RZ*. The *SLACK* heuristic builds the solution by inserting at each step the job with the minimum tardiness, which also limits its use to PFSP$_{TT}$. The *LIT* (Wang et al., 1997) heuristic builds the solution choosing always the job that has the minimum idle time. The *LR* heuristic uses an index function that takes into account the idle times, IT_{ik} and AT_{ik}, which is the sum of completion times derived by the insertion of job k. A number of initial sequences is considered by using as starting job the j jobs with the minimum value of the index function, with j being a parameter of the heuristic. At the end of the execution the sequence with the minimum objective function value is returned. The *NLR* heuristic uses the job order defined by the *LR* heuristic (Liu and Reeves, 2001) to build a solution using the insert construction rule.

3.4. Iterative Improvement

Iterative improvement algorithms are local search algorithms that at each step only accept improving neighboring candidate solutions to replace the current one. They take as input an initial candidate solution, a specific neighborhood relation and a rule of how the neighborhood is searched and which neighboring candidate solution replaces the current

Heuristic	$h(J_i)$	Greedy Strategy	Construction	Local Search∗	Objectives
NEH (Nawaz et al., 1983)	$\sum_1^m p_{ij}$	$max(h(J_i))$	Insert	-	All
NEH_{tb} (Fernandez-Viagas and Framiñán, 2014)	$\sum_1^m p_{ij}$	$max(h(J_i))$	Insert	-	PFSP$_{MS}$
NEH_{edd} (Kim, 1993)	dd_i	$max(h(J_i))$	Insert	-	PFSP$_{TT}$
FRB5 (Rad et al., 2009)	$\sum_1^m p_{ij}$	$max(h(J_i))$	Insert	local minimum	PFSP$_{MS}$
RZ (Rajendran and Ziegler, 1997)	$\frac{1}{g_i}\sum_{j=k}^m (m-j+1)\cdot p_{ij}$	$min(h(J_i))$	Append	n steps	All
NRZ	$\frac{1}{g_i}\sum_{j=k}^m (m-j+1)\cdot p_{ij}$	$min(h(J_i))$	Insert	n steps	All
NRZ_2	$\frac{1}{g_i}\sum_{j=k}^m (m-j+1)\cdot p_{ij}$	$min(h(J_i))$	Insert	-	All
SLACK	$dd_i - C_{i,m}$	$min(h(J_i))$	Append	-	PFSP$_{TT}$
LIT (Wang et al., 1997)	$\sum_{j=2}^m max\{C_{i,j-1} - C_{k,j}, 0\}$	$min(h(J_i))$	Append	-	All
LR (Liu and Reeves, 2001)	$(n-k-2)IT_{ik} + AT_{ik}$ $IT_{ik} = \sum_{j=2}^m w_{jk} max\{C_{i,j-1} - C_{k,j}, 0\}$	$min(h(J_i))$	Append	-	All
NLR	$(n-k-2)IT_{ik} + AT_{ik}$ $IT_{ik} = \sum_{j=2}^m w_{jk} max\{C_{i,j-1} - C_{k,j}, 0\}$	$min(h(J_i))$	Insert	-	All

∗The local search, if present, uses the first improvement pivoting rule and the insert neighborhood

Table 1: Heuristics implemented for the generation of the initial candidate solution.

one, which is also called *pivoting rule*. The neighborhood search process is iterated in the simplest case until no improving neighbor can be found, that is, in a local optimum. However, the iteration process may also be stopped prematurely, e.g., after a given number of iterations. Additionally, iterative improvement algorithms may use more than one neighborhood relation, leading to variable neighborhood descent (VND) algorithms (Hansen and Mladenović, 2001). We implemented the most widely used pivoting rules, that is, first improvement and best improvement. First improvement, scans the neighborhood in some specific order and returns the first improving neighbor. Best improvement, instead, checks the complete neighborhood and returns the most improving neighbor. In case of ties on the most improving neighbor, in our implementation the first one found is returned.

In addition, we implemented two algorithms specifically designed for PFSP$_{TCT}$ and PFSP$_{TT}$, respectively, namely, *iRZ* (Pan and Ruiz, 2012) and CH6 (Li et al., 2015). *iRZ* is an iterative improvement algorithm that uses the local search used in the *RZ* heuristic (see Section 3.3). Instead of being applied just once, *iRZ* is applied iteratively until reaching a local minimum. CH6 is a more complex algorithm, which in its main loop executes two best improvement local searches in series, each one exploring a different neighborhood. The two local searches stop when reaching a local minimum.

Table 2 summarizes the iterative improvement variants we implemented. They take as parameters the initial solution from which iterative improvement starts, *In*, the termination

Iterative Improvement	pivoting rule	Neighborhood	Problems	Parameters
First Improvement	first improvement	single	any	$\langle In, T, N \rangle$
Best Improvement	best improvement	single	any	$\langle In, T, N \rangle$
VND	any	multiple	any	$\langle P, In, T, \{N_1, ..., N_k\} \rangle$
iRZ	best improvement	single	PFSP$_{TCT}$	$\langle \emptyset \rangle$
CH6	best improvement	multiple	any	$\langle \emptyset \rangle$

Table 2: Iterative Improvement algorithms implemented

criterion T (see Section 3.6 for a definition of termination criteria), and the neighborhood N. The VND takes as additional parameter the pivotal rule, indicated by P.

3.5. Neighborhood

The neighborhood $N(\pi)$ of a solution π can be defined as comprising the set of all solutions that can be generated by applying a specific operator that modifies π. For the PFSP, relevant are a few neighborhoods that change the order or the position of jobs in a permutation. We implemented the following operators. The **transpose** operator changes the position of two contiguous jobs. This neighborhood is of size $n - 1$ and can, thus, be very quickly explored. The **exchange** operator exchanges the positions of two jobs; the exchange neighborhood is of size of $n(n - 1)/2$ and contains the transpose neighborhood as a subset. The **insert** operator removes a job at a position i and inserts it in a position $k \neq i$; The insert neighborhood is of size $n(n-1)$ and contains the transpose neighborhood as a subset. The insert neighborhood may be further split into left or right inserts.

Table 3 shows the neighborhoods, which are included in EMILI as general implementations, as well as optimized versions of the insert neighborhood for the PFSP objectives tackled in this study. The insert neighborhood implemented for PFSP$_{MS}$, *tainsert*, uses the well known Taillard's accelerations (Taillard, 1990) while *attinsert* and *atctinsert*, the neighborhoods implemented specifically for PFSP$_{TT}$ and PFSP$_{TCT}$, use the speed-up technique presented in Pagnozzi and Stützle (2017). Finally, *karneigh* is a neighborhood relation proposed by Karabulut (2016) for the PFSP$_{TT}$ that at each step generates randomly either an insert or an exchange neighbor.

Neighborhood	Objectives	Neighborhood	Objectives
transpose	any	*tainsert*	PFSP$_{MS}$
exchange	any	*attinsert*	PFSP$_{TT}$
insert	any	*atctinsert*	PFSP$_{TCT}$
karneigh	PFSP$_{TT}$		

Table 3: Neighborhood implementations

Criterion	Stopping condition	Problem	Parameters
local minimum	*no improvement*	any	$\langle\emptyset\rangle$
maxsteps	$currenti > maxi$	any	$\langle maxi\rangle$
maxstepsorlocmin	*local minimum* $\wedge$ *maxsteps*	any	$\langle maxi\rangle$
non_imp_it	$currenti > maxi$	any	$\langle maxi\rangle$

Table 4: Termination Criteria used to generate algorithms in this study

3.6. Termination criterion

Generally, metaheuristics stop their execution when for some number of steps they cannot find an improving solution or after a specific CPU time. In EMILI, CPU time as a termination criterion is handled as an overall stopping criterion but not as an algorithm component. The other termination criteria described here can be used to either terminate local searches such as those described in Section 3.4 as well as the generic metaheuristic components. The termination criteria implemented in EMILI are shown in Table 4.

The termination condition *local minimum* triggers the termination of the corresponding procedure as soon as no improved candidate solution can be found by its search process. *maxsteps* triggers the termination of the algorithm when it has executed a number of iterations specified by parameter *maxi*. *maxstepsorlocmin* stops a procedure where it is used either if the condition *maxsteps* or the condition *local minimum* is satisfied. Finally, *non_imp_it* terminates a procedure if for a given number of iterations no improvement over the incumbent was found. If an improved solution is found, the counter is reset to 0.

3.7. Perturbation

Perturbations introduce a change to the current solution that is typically larger than the ones done in the local search. Thus, perturbations are crucial to allow the search process to escape local optima and explore different regions of the search space. The perturbations implemented in the framework and used in this study are shown in Table 5.

The *random move* perturbation does a number of *num* random walk steps in a specific neighborhood; at each random walk step it generates uniformly at random a neighboring solution and accepts it. It is advisable that the neighborhood chosen for the perturbation is different from the one used for the intensification phase of the metaheuristic in order to minimize the probability of going back to the starting current solution (Lourenço et al., 2010). Parameters of *random move* are the neighborhood and *num*.

Another effective perturbation technique is used in *IG* algorithms (Ruiz and Stützle, 2007); it is composed of two phases, destruction and construction. In the destruction phase, a number of d randomly chosen jobs is removed from the solution. In the construction phase, the removed jobs are reinserted in the solution one at a time, following the same procedure used by *NEH*. The IG_{tb} perturbation adds the tie breaking mechanism used in the NEH_{tb} heuristic. In IG_{io}, instead, the removed jobs are reinserted in the partial solution following the descending order of the sum of processing times. Another possibility is the re-optimization of the partial solutions obtained after the destruction phase (before the construction phase) (Dubois-Lacoste et al., 2017). IG_{lsps} and $IG_{tb+lsps}$ implement this possibility using for the construction phase *NEH* and NEH_{tb}, respectively.

We also implemented a compound perturbation (*CP*) (Li et al., 2015) that instead of one generates ω perturbed solutions, each obtained by executing d random steps in the *insert* and *transpose* neighborhoods. In particular, at each random step, with probability pc the *insert* and with probability $1 - pc$ the *transpose* neighborhood is chosen. If no improvement is found, a distance metric is used to return the solution with the largest distance from the current solution.

3.8. Acceptance criterion

The acceptance criterion influences the balance between intensification and diversification in an SLS algorithm. The acceptance criteria we used in this study are listed In Table

Perturbation	Problem	Parameters	Perturbation	Problem	Parameters
random move	any	$\langle neighborhood, num \rangle$	IG_{lsps}	PFSP	$\langle d, \mathsf{localsearch} \rangle$
IG	PFSP	$\langle d \rangle$	$IG_{tb+lsps}$	PFSP$_{MS}$	$\langle d, \mathsf{localsearch} \rangle$
IG_{tb}	PFSP	$\langle d \rangle$	CP	PFSP	$\langle d, \omega, pc \rangle$
IG_{io}	PFSP	$\langle d \rangle$			

Table 5: Perturbations used to generate algorithms in this study

6. The *better* acceptance criteria accepts a candidate solution only if it improves on the current one. A way to introduce diversification would be to accept non-improving solutions for k_n iterations, similar to what is proposed by Hong et al. (1997). We implemented the *diversify occasionally* acceptance criterion that does so if the incumbent solution is not improved for k_t iterations, which is taken as an indication of search stagnation. We denote this condition as C in Table 6. Moreover, with *ft*, *psa* and *sa* we implemented different kinds of Metropolis conditions (Metropolis et al., 1953), which is a probabilistic acceptance criterion used in simulated annealing. Given a current candidate solution π and a new one $\pi^{'}$, it accepts $\pi^{'}$ if it has better or equal quality as π or, otherwise, with a probability depending on the amount of worsening and a parameter T called temperature; more formally, the acceptance probability P_a is given by

$$P_a = \begin{cases} 1 & \text{if } f(\pi^{'}) \leq f(\pi) \\ exp(\frac{f(\pi)-f(\pi^{'})}{T}) & otherwise \end{cases} \tag{6}$$

In SA algorithms the parameter T changes its value at run-time. It typically starts at a high temperature, T_s, and is lowered until a final, low temperature T_e or another termination criterion is reached. We update the temperature every *it* iterations following the equation $T_{n+1} = \alpha \cdot T_n - \beta$ where α and β are real values between 0 and 1; for $\beta = 0$, the usual geometric cooling scheme results. The acceptance criteria *sa* and *psa* update the temperature using this method, where *psa* additionally enforces that α is set to one. The Metropolis condition can be also used with a fixed temperature value that does not change at run-time. This is implemented in *ft*. The acceptance criterion used in the IG_{rs} algorithm (Ruiz and Stützle, 2007) for PFSP$_{MS}$, uses a fixed temperature Metropolis condition,

Acceptance Criterion	Condition	Problem	Parameters
better	$\pi' < \pi$	any	$\langle \emptyset \rangle$
improve plateau	C	any	$\langle s_t, s_n \rangle$
ft	P_a	any	$\langle T \rangle$
psa	P_a	any	$\langle T_s, T_e, \beta, it \rangle$
sa	P_a	any	$\langle T_s, T_e, \beta, it, \alpha \rangle$
rsacc	P_a	PFSP	$\langle T_p \rangle$
karacc	P_a	PFSP	$\langle T_p \rangle$

Table 6: Acceptance criteria used to generate algorithms in this study

rsacc, with the temperature set to

$$T_{rs} = T_p \cdot \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} p_{\pi(i),j}}{n \cdot m \cdot 10}, \tag{7}$$

where T_p is a parameter. In this way, the temperature is linked to the average processing times of the specific problem instance to be solved. The *karacc* acceptance criterion described by Karabulut (2016) adapts *rsacc* to PFSP$_{TT}$ by calculating the temperature as

$$T_{kar} = T_p \cdot \frac{\sum_{j=1}^{n} LB_{Cmax} - d_j}{n \cdot 10}$$

where LB_{Cmax} is the lower bound for the makespan as defined by Taillard (1993) and d_j is the due date of job j.

3.9. Tabu Search

Tabu Search is an SLS method that uses memory to direct the search and to escape local optima, typically by forbidding certain solutions or solution components to avoid reversing moves and revisiting solutions.

In EMILI, the tabu memory is the component of Tabu Search that stores the solution characteristics that are prohibited according to the algorithm and that is used to determine if a neighboring candidate solution is admissible. We implemented three types of tabu memory for PFSP. The first is *solution*, which stores entire solutions and forbids any candidate solution equal to one of the solutions in the tabu memory. The second is *hash*, which uses a hash code to represent solutions. The hash is calculated considering the permutation as a string and using the hash function of the C++ standard library. A candidate solution is forbidden if its hash is in the tabu memory. The third is *move*, which forbids a move changing the position of a job from i to j to forbid reverse moves of jobs.

4. Experimental Results

We use our automated design approach to generate algorithms for three of the most studied PFSP objectives. The main steps to apply the automated design approach have been the following. As a first step, we adapted our generic grammar for each of the objectives by extending some rules with problem specific, possible derivations. Examples are the possibility of using the insert neighborhood for PFSP$_{MS}$ with the speed-ups proposed by Taillard or the usage of the NEH_{edd} heuristic, which, due to its use of due dates is applicable only for PFSP$_{TT}$. In any case, these modifications are straightforward and just consist in adding or deleting some terms in the grammar. From each of the resulting three grammars, we generated a set of parameters following the method outlined in Mascia et al. (2013). This , combined with the different number of problem specific components, resulted in a number of 502, 446 and 481 parameters to be configured for PFSP$_{MS}$, PFSP$_{TCT}$ and PFSP$_{TT}$, respectively. However, many of these parameters are conditional parameters, which are only relevant for specific settings of other, categorical parameters whose values define the algorithm design. As the automatic configuration method to search the parameter spaces we use irace, a publicly available AAC tool that implements the iterated racing procedure described in detail in López-Ibáñez et al. (2016).

The training set used for the tuning process for all the problems we consider here is composed of 40 randomly generated instances. In particular, following the procedure described in Minella et al. (2008), five instances were generated for each number of jobs $\{50, 60, 70, 80, 90, 100\}$ with 20 machines as well as five instances of size 250×30 and 250×50. For PFSP$_{TCT}$ appropriately defined due dates are used for the instances. For each configuration task, we run irace twice in the following way. In a first run, irace generates the initial configurations by uniformly sampling the parameters space. In the second run, we use the best configurations given at the end of the first run as initial configurations. The configuration used for the experiments is the one considered to be the best by irace at the end of the second run. In our study, the number of algorithm executions per irace run is set to 10^5, which gives a total configuration budget of $2 \cdot 10^5$ algorithm executions for each objective. This results into an overall time for the configuration of ca. two days wallclock time using about 100 computing cores. For irace, the default settings are used apart from the above described setup. The tuning was executed on a Intel Xeon E5410 CPUs running at 2.33 GHz while the test results were generated on AMD Opteron 6272 CPUs running at 2.1 GHz; all machines run CentOS 6.2 Linux and each execution of EMILI is single-thread.

In the following, for each objective, we present the results of the experiments comparing the best configuration found by irace with the state-of-the-art algorithm on the commonly used benchmark sets in the literature. The time limit was set to $n \cdot (m/2) \cdot t$ milliseconds using for t the values of 60, 120 and 240. To take into account the stochasticity of the tested algorithms, all algorithms were executed 30 times on each instance considering different random seeds. All state-of-the-art algorithms used for the comparison were coded in EMILI and for the tests we used the parameter configurations given by the authors, which were in all cases obtained after in part rather extensive tuning experiments. Detailed information on the automatically obtained algorithms, their parameter settings and a high-level comparison to how they compare to the known best-performing algorithms is available at the article's supplementary page (Pagnozzi and Stützle, 2018).

4.1. PFSP: Makespan

The PFSP$_{MS}$ is the most studied PFSP variant. In fact, many high performing algorithms have been proposed for this objective (Fernandez-Viagas et al., 2017). Among these, the IG algorithm by Ruiz and Stützle has for a long time been the best performing algorithm for this problem (Ruiz and Stützle, 2007) and only rather recently two improved variants of it have been proposed: IG_{tb} (Fernandez-Viagas and Framiñán, 2014) and IG_{all} (Dubois-Lacoste et al., 2017). IG_{tb} improves on IG_{rs} by using an improved *NEH* heuristic and an *IG* perturbation that use a new tie-breaking strategy, while keeping the same

Algorithm 2 IG_{all}

```
1: Output The best solution found $\pi^*$,
2: $\pi := \mathsf{FRB5}()$;
3: $\pi := \mathsf{II}(\pi, local\ minima, tainsert)$;
4: $\pi^* := \pi$
5: while ! time is over do
6:     $\pi' := \mathsf{IG}_{tb+lsps}(\pi, ls(first, local\ minima, tainsert))$;
7:     $\pi' := \mathsf{II}(\pi', first, local\ minima, tainsert)$;
8:     $\pi := \mathsf{rsacc}(\pi, \pi')$;
9:     if $f(\pi') < f(\pi^*)$ then
10:        $\pi^* := \pi'$
11:    end if
12: end while
13: Return $\pi^*$
```

general structure and parameter setting. The current state-of-the-art algorithm, IG_{all}, improves on the previous IG algorithms by using the *FRB5* heuristic for the initialization and a variant of the IG perturbation, where a local search is used to improve the partial solution obtained after the destruction phase. The outline of this algorithm is shown in Algorithm 2.

The algorithm generated by irace, IG_{irms}, is shown in Algorithm 3 with the parameters in Table 7. IG_{irms} shares some similarities to IG_{all} in the sense that it is an iterated greedy algorithm using a first-improvement local search and applies also a local search in the perturbation, that is, it uses the IG_{lsps} perturbation. However, beyond settings of numerical parameters, there are some differences between the algorithms that concern the acceptance critieria and the usage of a best-improvement local search for optimizing partial solutions in the IG_{lsps} perturbation. For more details we refer to in the supplementary material (Pagnozzi and Stützle, 2018). Interestingly, IG_{irms} does not use the recursive grammar rules but corresponds to a plain iterated greedy algorithm; this indicates that the automatic configuration process can generate simple algorithms and has no a priori bias towards overly complex hybrid methods.

We compared the algorithms using the benchmark set of Taillard (1993) and the

Algorithm 3 IG_{irms}

```
1: Output The best solution found $\pi^*$,
2: $\pi$ := $\mathit{FRB5}()$;
3: $\pi$ := $\mathrm{II}(\pi, maxstepsorlocmin(s), tainsert)$;
4: $\pi^* := \pi$
5: while ! time is over do
6:     $\pi'$ := $IG_{lsps}(\pi, ls(best, local\ minima, tainsert))$;
7:     $\pi'$ := $\mathrm{II}(\pi', first, maxstepsorlocmin, tainsert)$;
8:     $\pi$ := $psa(\pi, \pi')$;
9:     if $f(\pi') < f(\pi^*)$ then
10:        $\pi^* := \pi'$
11:    end if
12: end while
13: Return $\pi^*$
```

large instance set of the VRF benchmark Vallada et al. (2015). The former consists of 120 instances divided in groups of ten. The instances have a number of jobs $n \in \{20, 50, 100, 200, 500\}$ and machines $m \in \{5, 10, 20\}$. The latter consists of 240 instances divided in 24 groups of $n \in \{100, 200, 300, 400, 500, 600, 700, 800\}$ jobs and $m \in \{20, 40, 60\}$ machines.

For each algorithm and each instance we measured the average relative percentage deviation (ARPD) from the best results published on Taillard's website[1]. The results of the experiments are reported in Figure 2 and Figure 3. In the Table 8 and Table 9, the values are grouped by instance size. IG_{irms} is clearly the best performing algorithm overall. The differences considering the taillard benchmark are rather minor altough statistically significant. On the other hand, the VRF benchmark shows that there is a huge margin between IG_{irms} and IG_{all}. This results are remarkable given the very advanced state of the art for the PFSP under makespan objective (Fernandez-Viagas et al., 2017).

	Component	Parameter	Value
IG_{irms}			
	maxstepsorlocmin	*maxi*	77
	IG_{lsps}	d	1
	psa	T_s	4.6512
		T_e	0.9837
		β	0.0234
		it	324

Table 7: Parameter settings for IG_{irms}

	$t = 60$		$t = 120$		$t = 240$	
Instances	IG_{all}	IG_{irms}	IG_{all}	IG_{irms}	IG_{all}	IG_{irms}
20x5	0.03	0.03	0.02	0.02	0.02	0.01
20x10	**0.003**	0.01	**0**	0.01	**0**	0.01
20x20	**0**	0.01	**0**	0.01	**0**	0.01
50x5	0	0	0	0	0	0
50x10	0.36	**0.30**	0.33	**0.28**	0.30	**0.25**
50x20	0.48	0.47	0.40	0.39	0.34	0.34
100x5	0	0	0	0	0	0
100x10	0.04	0.03	0.02	0.03	0.02	0.02
100x20	0.78	**0.62**	0.67	**0.52**	0.56	**0.44**
200x10	0.04	**0.03**	0.03	**0.03**	0.03	**0.03**
200x20	0.80	**0.66**	0.70	**0.57**	0.63	**0.51**
500x20	0.34	**0.29**	0.30	**0.26**	0.28	**0.24**
Average	0.24	**0.21**	0.21	**0.18**	0.18	**0.16**

Table 8: ARPD results of IG_{all} and IG_{irms} for the two running times. If an algorithm is statistically significantly better, according to the Wilcoxon signed-rank test with a 95% confidence, this is shown in bold face.

4.2. PFSP: Sum completion times

Over the years, various metaheuristic algorithms have been proposed to tackle this problem. These include population-based (Rajendran and Ziegler, 2004; Tasgetiren et al., 2007; Tseng and Lin, 2009) and trajectory-based algorithms (Pan et al., 2008; Dong et al., 2009). The current state-of-the-art algorithm is an IG algorithm, *IGA* (Pan and Ruiz, 2012) shown in Algorithm 4. This algorithm uses as local search an exploration scheme inspired by the improvement phase of the *RZ* heuristic. This scheme starts by considering the jobs in the order of the current solution. It then determines the best insertion for a given job

[1] `http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/flowshop.dir/best_lb_up.txt`

	$T = 60$		$T = 120$		$T = 240$	
Instances	IG_{all}	IG_{irms}	IG_{all}	IG_{irms}	IG_{all}	IG_{irms}
100x20	0.71	**0.57**	0.56	**0.42**	0.44	**0.29**
100x40	0.83	**0.67**	0.66	**0.49**	0.52	**0.35**
100x60	0.78	**0.64**	0.62	**0.48**	0.48	**0.34**
200x20	0.59	**0.45**	0.47	**0.32**	0.37	**0.22**
200x40	0.95	**0.79**	0.71	**0.52**	0.52	**0.30**
200x60	0.93	**0.74**	0.73	**0.50**	0.53	**0.28**
300x20	0.48	**0.35**	0.39	**0.24**	0.30	**0.17**
300x40	0.84	**0.70**	0.64	**0.48**	0.44	**0.24**
300x60	0.94	**0.77**	0.73	**0.53**	0.51	**0.29**
400x20	0.28	**0.21**	0.23	**0.16**	0.19	**0.12**
400x40	0.75	**0.62**	0.56	**0.42**	0.36	**0.22**
400x60	0.80	**0.68**	0.60	**0.46**	0.42	**0.23**
500x20	0.22	**0.17**	0.19	**0.13**	0.15	**0.09**
500x40	0.68	**0.54**	0.50	**0.37**	0.34	**0.20**
500x60	0.72	**0.61**	0.53	**0.41**	0.35	**0.21**
600x20	0.19	**0.17**	0.16	**0.13**	0.12	**0.09**
600x40	0.63	**0.52**	0.46	**0.34**	0.29	**0.17**
600x60	0.70	**0.62**	0.51	**0.42**	0.34	**0.21**
700x20	0.16	**0.14**	0.13	**0.10**	0.10	**0.07**
700x40	0.61	**0.48**	0.45	**0.31**	0.29	**0.15**
700x60	0.66	**0.57**	0.49	**0.37**	0.31	**0.19**
800x20	0.15	**0.15**	0.13	**0.10**	0.10	**0.07**
800x40	0.56	**0.46**	0.40	**0.29**	0.25	**0.14**
800x60	0.61	**0.51**	0.45	**0.32**	0.28	**0.15**
Average	0.62	**0.50**	0.47	**0.35**	0.34	**0.20**

Table 9: Average RPD values obtained by IG_{all} and IG_{irms} on the different instance sizes o the VRF-large benchmark set, using three different values for T. The algorithm that is significantly better than the other according to the Wilcoxon signed-rank test with a 95% confidence is shown in bold face.

and the resulting sequence substitutes the current one if it reduces the sum of completion times. This procedure is iterated until a local optimum is met. The initial solution of IGA is based on the LR heuristic and it is set up to consider the first n/m sequences to choose the initial solution. The perturbation as well as the acceptance criterion are the same as in IG_{rs}, that is, IG and $rsacc$. We carefully implemented the IGA algorithm in EMILI, cross-checking to reach or surpass the performance of the algorithm used in the original paper. As already mentioned, the *insert* neighborhood for this objective uses an approximation based speed-up while iRZ uses the speed-up described by the authors, which avoids to reevaluate the parts of the solution that have not changed.

The algorithm produced with our system, ALG_{irtct}, is a hybrid IG that has an ILS as local search; in other words it makes use of the possibility of hybridization as offered in EMILI by running an iterated local search inside an iterated greedy algorithm. For outlines of ALG_{irtct} and the parameter settings we refer to the supplementary material (Pagnozzi

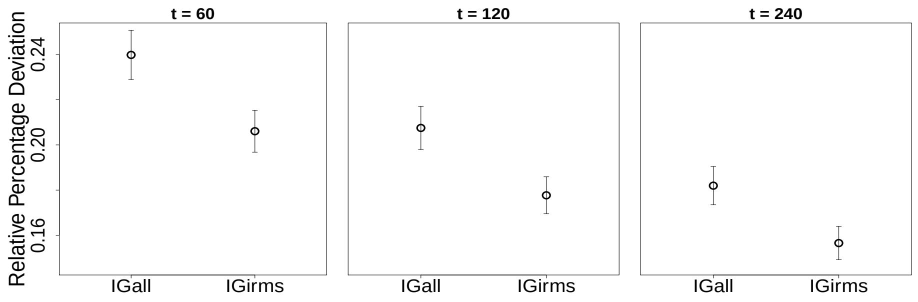

Figure 2: PFSP$_{MS}$ comparison on the Taillard benchmark, Average RPD and 95% confidence intervals of IG_{all} and IG_{irms} for $T = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

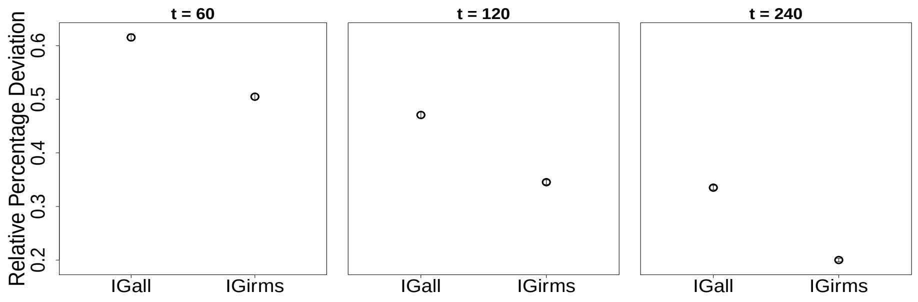

Figure 3: PFSP$_{MS}$ comparison on VRF benchmark, Average RPD and 95% confidence intervals of IG_{all} and IG_{irms} for $T = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

and Stützle, 2018). In a nutshell, the inner ILS in ALG_{irtct} has a rather diversifying effect while the outer IG is more intensifying due to a rather restrictive acceptance criterion. The outline of ALG_{irtct} is shown in Algorithm 5 and 6, with the parameters shown in Table 10.

We evaluate the algorithms on the benchmark set used in the paper where the IGA algorithm was presented; it is composed of the benchmark set from Taillard plus 30 instances in three groups of size 200x5, 500x5, 500x10 (10 instances each group). For both algorithms we compute the ARPD, using the best solutions reported in Pan and Ruiz (2012). The results, in Figure 4 and in Table 11, clearly show that ALG_{irtct} improves over the performance of IGA, which is more visible for the higher computation time limits.

Recently, a new constructive heuristics for the PFSP$_{TCT}$ was proposed by Fernandez-

Algorithm 4 *IGA*

```
1:  Output The best solution found $\pi^*$,
2:  $\pi := \mathit{LR}(n/m)$;
3:  $\pi := \mathit{iRZ}(\pi)$;
4:  $\pi^* := \pi$
5:  while ! time is over do
6:      $\pi' := \mathit{IG}(\pi)$;
7:      $\pi' := \mathit{iRZ}(\pi')$;
8:      $\pi := \mathit{rsacc}(\pi, \pi')$;
9:      if $f(\pi') < f(\pi^*)$ then
10:         $\pi^* := \pi'$
11:     end if
12: end while
13: Return $\pi^*$
```

Algorithm 5 ALG_{irtct}

```
1:  Output The best solution found $\pi^*$,
2:  $\pi := \mathit{NRZ}_2()$;
3:  $\pi := \mathrm{ALGirtct2}(\pi)$;
4:  $\pi^* := \pi$
5:  while ! time is over do
6:      $\pi' := \mathit{IG}_{io}(\pi)$;
7:      $\pi' := \mathrm{ALGirtct2}(\pi')$;
8:      $\pi := \mathit{psa}(\pi, \pi')$;
9:      if $f(\pi') < f(\pi^*)$ then
10:         $\pi^* := \pi'$
11:     end if
12: end while
13: return $\pi^*$
```

Viagas and Framiñán (2017), which is based on beam search. This constructive heuristic apparently has a strongly positive influence on the ILS algorithm they study (similar to the usage of the FRB5 heuristic on the PFSP$_{MS}$ (Dubois-Lacoste et al., 2017)) and, hence, an extension for our work would be to add this heuristic as an algorithmic component.

4.3. PFSP: Total tardiness

Several algorithms have been proposed for the PFSP$_{TT}$, ranging from simulated annealing (Hasija and Rajendran, 2004) to genetic algorithms (Vallada and Ruiz, 2010). Recently, new metaheuristic algorithms have been proposed (Li et al., 2015; Karabulut,

Algorithm 6 ALG_{irtct2}

```
1: input current solution $\pi$
2: output the best solution found $\pi^*$,
3: $\pi$ := ii($\pi$, $first$, $local\ minima$, $insert$);
4: $\pi^* := \pi$
5: while maxsteps() do
6:     $\pi'$ := random move($\pi$, $transpose$);
7:     $\pi'$ := ii($\pi'$, $first$, $local\ minima$, $insert$);
8:     $\pi$ := psa($\pi$, $\pi'$);
9:     if $f(\pi') < f(\pi^*)$ then
10:        $\pi^* := \pi'$
11:    end if
12: end while
13: return $\pi^*$
```

	Component	Parameter	Value		Component	Parameter	Value
ALG_{irtct}				ALG_{irtct2}			
	IG_{io}	d	11		*maxsteps*	$maxi$	173
	sa	t_s	1.0448		*random move*	num	8
		t_e	0.8470		*psa*	t_s	4.9498
		β	0.0944			t_e	0.0339
		it	423			β	0.0944
		α	0.9555			it	349

Table 10: Parameter settings for ALG_{irtct}

2016), which were shown to outperform the others. The first one is an ILS algorithm, called TSM63, that uses NEH_{edd} to generate the initial solution, the CH6 local search, the *CP* perturbation, and as acceptance criterion the algorithm accepts the best solution generated by the perturbation plus local search phase. We contacted the authors and they kindly sent us the source code of the algorithm. We used the original code in Java as a guide to implement TSM63 within EMILI. This allowed us to use also the speed-ups for the insert neighborhood (Pagnozzi and Stützle, 2017). We verified that implementing TSM63 within our framework improved significantly the performance when compared to the authors' original Java implementation, thus making also a comparison to EMILI more fair. The second algorithm, IG_{RLS} is an *IG* algorithm that uses a first improvement local search that uses the neighborhood *karneigh* and the termination condition *non_imp_it*. The initial solution is generated with NEH_{edd}, the perturbation is the *IG* perturbation and the

	$t = 60$		$t = 120$		$t = 240$	
Instances	IGA	ALG_{irtct}	IGA	ALG_{irtct}	IGA	ALG_{irtct}
20x5	0.15	**0.003**	0.15	**0.001**	0.15	**0.0001**
20x10	0.0002	0	0	0	0	0
20x20	0	0	0	0	0	0
50x5	0.64	**0.47**	0.54	**0.38**	0.48	**0.31**
50x10	1.10	**0.51**	1.04	**0.41**	0.99	**0.35**
50x20	0.72	**0.45**	0.66	**0.35**	0.61	**0.29**
100x5	1.17	**0.99**	1.08	**0.89**	0.99	**0.81**
100x10	1.49	**1.03**	1.37	**0.90**	1.29	**0.79**
100x20	1.54	**1.15**	1.40	**0.97**	1.30	**0.83**
200x5	**0.40**	0.50	**0.35**	0.41	**0.30**	0.36
200x10	1.27	**0.86**	1.17	**0.73**	1.09	**0.64**
200x20	1.09	**0.70**	0.92	**0.53**	0.80	**0.39**
500x5	**0.21**	0.81	**0.20**	0.72	**0.19**	0.65
500x10	**0.29**	0.76	**0.26**	0.58	**0.24**	0.45
500x20	**0.49**	0.63	0.42	0.42	0.36	**0.24**
Average	0.70	**0.59**	0.64	**0.49**	0.59	**0.41**

Table 11: Average RPD results of IGA and ALG_{irtct} for the three running times.The algorithm that is significantly better than the other, according to the Wilcoxon signed-rank test with a 95% confidence, is shown in bold face.

acceptance criterion is $karacc$. The outline of both algorithms is shown in Algorithm 7 and 8.

The algorithm generated by irace, ALG_{irtt}, shown in Algorithm 9 and 10, the outer layer being composed of the perturbation and acceptance from TSM63 and the inner layer being a rather plain ILS algorithm. The parameter settings of ALG_{irtt} are shown in Table 12.

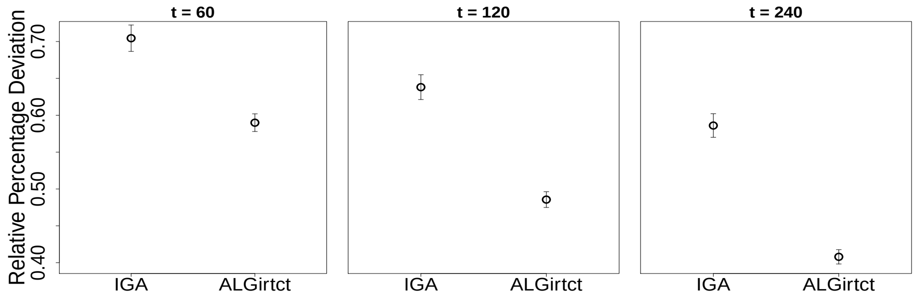

Figure 4: Average RPD and 95% confidence intervals of IGA and ALG_{irtct} for $T = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

	Component	Parameter	Value		Component	Parameter	Value
ALG_{irtt}				ALG_{irtt2}			
	CP	d	2		$maxsteps$	$maxi$	39
		ω	29		$random\ move$	num	7
		pc	0.7955		$rsacc$	T_{rs}	0.5203

Table 12: Parameter settings for ALG_{irtt}

For the tests we used the benchmark proposed in Vallada et al. (2008). It consists of 540 instances divided in groups of 45 instances of the number of jobs $n \in \{50, 150, 250, 350\}$ and machines $m \in \{10, 30, 50\}$. Since some of these instances have 0 as best solution value, we used the Relative Deviation Index to compare the algorithms, where $RDI = \frac{V-V_b}{V_w-V_b}$ and V_b is the best value and V_w is the worst. We used as worst values the ones provided with the benchmark set while the best values are the best ones found during the experiments and are available in the supplementary pages. From the results, shown in Figure 5, ALG_{irtt}, outperforms both TSM63 and IG_{RLS} with TSM63 being the worst.

In Table 13 we report the results grouped by instance size. TSM63 is always outperformed by ALG_{irtt} and IG_{RLS}, with the exception of the instance size 50x50 , where TSM63 has better results than IG_{RLS} for the longer running time. On average, ALG_{irtt} performs the best. When focusing on specific instance sizes, ALG_{irtt} performs often best except for instance sizes 150x10, 250x50 and all instances with 350 jobs, where IG_{RLS} is significantly better than ALG_{irtt}. Finally, it is quite interesting to note that CP is used in the best and worst algorithm of the three.

Recently, an ILS algorithm, IA_{ras}, has been proposed for PFSP$_{TT}$ (Fernandez-Viagas et al., 2018). IA_{ras} is composed of an insertion based local search, a perturbation based on random moves in the transpose neighborhood and the same acceptance criterion as IG_{RLS}. The key characteristic of the algorithm is a beam search based heuristic to generate the initial solution. The experimental results reported by the authors show that IA_{ras} performs better than both TSM63 and IG_{RLS}. Although we were unable to replicate the performance of the BS heuristic, and, thus, could not confirm the performance of the proposed algorithm, it is possible to compare the experiments reported in Fernandez-Viagas et al. (2018) with ours because of the same benchmark set, the same way of computing the RDI values, and the same stopping criteria. (Note that for the way we define the stopping criterion here, we would have to divide T by two, as we use the formula $n \cdot (m/2) \cdot T$ while they use $n \cdot m \cdot T$.) However, Fernandez-Viagas et al. (2018) used a Core i7 3770 CPU, which according to the passmark benchmark for single-thread ratings is 2.98 times faster than the CPU we are using. Even if we do not take this speed difference in their favor into account, we conclude that ALG_{irtt} would outperform IA_{ras} as it has a much smaller average RDI when using the same settings of computation time limit T (e.g. calculating the RDI using the same best solutions, ALG_{irtt} as an average RDI of -1.45 for $T = 240$ versus -0.79 for IA_{ras}).

	$t = 60$			$t = 120$			$t = 240$		
Instances	TSM63	IG_{RLS}	ALG_{irtt}	TSM63	IG_{RLS}	ALG_{irtt}	TSM63	IG_{RLS}	ALG_{irtt}
50x10	3.37	3.02	**2.11**	2.58	2.46	**1.31**	1.84	1.94	**0.64**
50x30	1.46	2.09	**0.63**	1.13	1.91	**0.40**	0.86	1.73	**0.20**
50x50	0.62	0.99	**0.23**	0.47	0.87	**0.13**	0.35	0.77	**0.06**
150x10	0.18	0.42	**0.05**	0.14	0.37	**0.03**	0.09	0.31	**0.02**
150x30	5.20	**3.05**	3.62	4.12	**2.17**	2.56	3.13	**1.35**	1.62
150x50	3.57	2.76	**1.86**	2.82	2.21	**1.24**	2.13	1.69	**0.73**
250x10	3.10	2.49	**1.70**	2.50	2.00	**1.20**	1.96	1.54	**0.72**
250x30	2.36	1.84	**1.20**	1.95	1.43	**0.82**	1.53	1.09	**0.47**
250x50	6.79	**3.62**	4.79	5.58	**2.67**	3.57	4.46	**1.83**	2.49
350x10	5.14	**2.85**	3.07	4.12	**2.08**	2.29	3.15	**1.45**	1.57
350x30	5.06	**2.70**	3.34	4.20	**1.98**	2.53	3.41	**1.31**	1.80
350x50	4.94	**2.82**	3.16	4.14	**2.13**	2.36	3.34	**1.48**	1.64
Average	3.48	2.39	**2.15**	2.81	1.86	**1.54**	2.19	1.37	**1.00**

Table 13: Average RDI results of TSM63, IG_{RLS} and ALG_{irtt} for the three running times. The algorithm that is significantly better than the others, according to the Wilcoxon signed-rank test with a 95% confidence corrected using Bonferroni, is shown in bold face.

5. Discussion and conclusions

In this paper we have generated automatically new high-performing algorithms for three of the most widely studied permutation flowshop scheduling problems, where in two cases they are the new state of the art. We think that this is a relevant achievement, as for each of these three problems that current state of the art is already very much advanced. Hence, these results together with those of other successful approaches towards automated algorithm design (KhudaBukhsh et al., 2009; López-Ibáñez and Stützle, 2012) demonstrate

Algorithm 7 TSM63

```
1:  Output The best solution found $\pi^*$,
2:  $\pi$ := $NEH_{edd}()$;
3:  $\pi$ := CH6($\pi$);
4:  $\pi^* := \pi$
5:  while ! time is over do
6:      $\pi'$ := $CP(\pi)$;
7:      $\pi'$ := CH6($\pi'$);
8:      $\pi := \pi'$;
9:      if $f(\pi') < f(\pi^*)$ then
10:         $\pi^* := \pi'$
11:     end if
12: end while
13: Return $\pi^*$
```

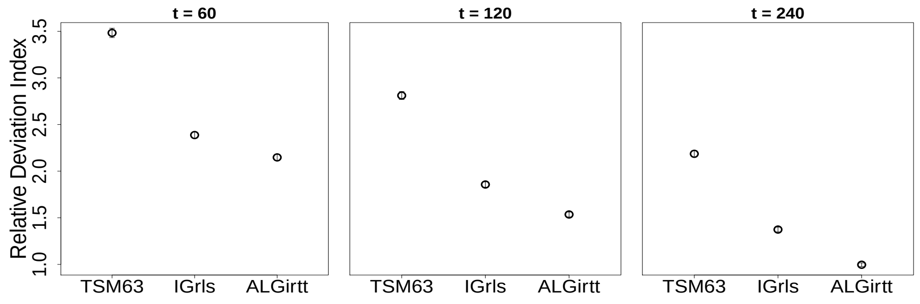

Figure 5: Average RDI and 95% confidence intervals of TSM63, IG_{RLS} and ALG_{irtt} for $t = 60$(left), $T = 120$(center) and $T = 240$ (right).

the large promise of this direction. In our case, this progress is based on joining a flexible algorithm framework that has been developed with a focus on automatic configurability, a scheme of how to instantiate various SLS methods and combinations thereof, and the increased performance of automatic algorithm configuration techniques. As a result, the EMILI framework and the associated automated design process is among the most advanced applications of the programming by optimization paradigm (Hoos, 2012).

There are a number of directions for future work. First, it would be interesting to study how the grammar impacts the automatic configuration and the complexity of the resulting algorithm and how different schemes for the combination of algorithm components

Algorithm 8 IG_{RLS}

```
1: Output The best solution found $\pi^*$,
2: $\pi := \mathit{NEH}_{edd}()$;
3: $\pi := \mathsf{II}(\pi, first, local\ minima, karneigh)$;
4: $\pi^* := \pi$
5: while ! time is over do
6:     $\pi' := \mathit{IG}(\pi)$;
7:     $\pi' := \mathsf{II}(\pi, first, local\ minima, karneigh)$;
8:     $\pi := \mathit{karacc}(\pi, \pi')$;
9:     if $f(\pi') < f(\pi^*)$ then
10:        $\pi^* := \pi'$
11:    end if
12: end while
13: Return $\pi^*$
```

Algorithm 9 ALGirtt

```
1: Output The best solution found $\pi^*$,
2: $\pi := \mathit{NEH}_{edd}()$;
3: $\pi := \mathsf{ALGirtt2}(\pi)$;
4: $\pi^* := \pi$
5: while ! time is over do
6:     $\pi' := \mathit{CP}(\pi)$;
7:     $\pi' := \mathsf{ALGirtt2}(\pi')$;
8:     $\pi := \pi'$;
9:     if $f(\pi') < f(\pi^*)$ then
10:        $\pi^* := \pi'$
11:    end if
12: end while
13: Return $\pi^*$
```

impact on the design of algorithms. A second direction is to extend the EMILI framework by increasing the set of available algorithm components and by representing also other types of SLS methods such as population-based ones. While one reason for the success of our approach is the integration of components into EMILI that capture the advanced knowledge in the literature on tackling specific problems, a third direction for future work is to add to this the possibility of generating automatically new components, heuristics or search strategies.

Algorithm 10 ALGirtt2

```
1: Input Current solution $\pi$
2: Output The best solution found $\pi^*$,
3: $\pi$ := $\mathsf{ll}(\pi, first, local\ minima, insert)$;
4: $\pi^* := \pi$
5: while $\mathsf{maxsteps}()$ do
6:     $\pi'$ := $\mathsf{random\ move}(\pi, transpose)$;
7:     $\pi'$ := $\mathsf{ll}(\pi', first, local\ minima, insert)$;
8:     $\pi$ := $\mathsf{rsacc}(\pi, \pi')$;
9:     if $f(\pi') < f(\pi^*)$ then
10:        $\pi^* := \pi'$
11:    end if
12: end while
13: Return $\pi^*$
```

References

Burke, E. K., Hyde, M. R., and Kendall, G. (2012). Grammatical evolution of local search heuristics. *IEEE Transactions on Evolutionary Computation*, 16(7):406–417.

Cahon, S., Melab, N., and Talbi, E.-G. (2004). ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380.

Dong, X., Huang, H., and Chen, P. (2009). An iterated local search algorithm for the permutation flowshop problem with total flowtime criterion. *Computers & Operations Research*, 36(5):1664–1669.

Dubois-Lacoste, J., Pagnozzi, F., and Stützle, T. (2017). An iterated greedy algorithm with optimization of partial solutions for the permutation flowshop problem. *Computers & Operations Research*, 81:160–166.

Fernandez-Viagas, V. and Framiñán, J. M. (2014). On insertion tie-breaking rules in heuristics for the permutation flowshop scheduling problem. *Computers & Operations Research*, 45:60–67.

Fernandez-Viagas, V. and Framiñán, J. M. (2017). A beam-search-based cnstructive heuristic for the PFSP to minimise total flowtime. *Computers & Operations Research*, 81:167–177.

Fernandez-Viagas, V., Ruiz, R., and Framiñán, J. M. (2017). A new vision of approximate methods for the permutation flowshop to minimise makespan: State-of-the-art and computational evaluation. *European Journal of Operational Research*, 257(3):707–721.

Fernandez-Viagas, V., Valente, J. M. S., and Framiñán, J. M. (2018). Iterated-greedy-based algorithms with beam search initialization for the permutation flowshop to minimise total tardiness. *Expert Systems with Applications*, 94:58 – 69.

Framiñán, J. M., Leisten, R., and Ruiz, R. (2014). *Manufacturing Scheduling Systems: An Integrated View on Models, Methods, and Tools.* Springer, New York, NY.

Hansen, P. and Mladenović, N. (2001). Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467.

Hasija, S. and Rajendran, C. (2004). Scheduling in flowshops to minimize total tardiness of jobs. *International Journal of Production Research*, 42(11):2289 – 2301.

Hong, I., Kahng, A. B., and Moon, B. R. (1997). Improved large-step Markov chain variants for the symmetric TSP. *Journal of Heuristics*, 3(1):63–81.

Hoos, H. H. (2012). Programming by optimization. *Communications of the ACM*, 55(2):70–80.

Hoos, H. H. and Stützle, T. (2005). *Stochastic Local Search—Foundations and Applications*. Morgan Kaufmann Publishers, San Francisco, CA.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In Coello Coello, C. A., editor, *Learning and Intelligent Optimization, 5th International Conference, LION 5*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, Heidelberg, Germany.

Johnson, D. S. (1954). Optimal two- and three-stage production scheduling with setup times included. *Naval Research Logistics Quarterly*, 1:61–68.

Karabulut, K. (2016). A hybrid iterated greedy algorithm for total tardiness minimization in permutation flowshops. *Computers and Industrial Engineering*, 98(Supplement C):300 – 307.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., and Leyton-Brown, K. (2009). SATenstein: Automatically building local search SAT solvers from components. In Boutilier, C., editor, *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 517–524. AAAI Press, Menlo Park, CA.

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

Li, X., Chen, L., Xu, H., and Gupta, J. N. (2015). Trajectory scheduling methods for minimizing total tardiness in a flowshop. *Operations Research Perspectives*, 2:13–23.

Liao, T., Stützle, T., Montes de Oca, M. A., and Dorigo, M. (2014). A unified ant colony optimization algorithm for continuous optimization. *European Journal of Operational Research*, 234(3):597–609.

Liu, J. and Reeves, C. R. (2001). Constructive and composite heuristic solutions to the P//ΣCi scheduling problem. *European Journal of Operational Research*, 132(2):439–452.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., and Birattari, M. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

López-Ibáñez, M., Kessaci, M.-E., and Stützle, T. (2017). Automatic design of hybrid metaheuristics from algorithmic components. *Submitted.*

López-Ibáñez, M. and Stützle, T. (2012). The automatic design of multi-objective ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation*, 16(6):861–875.

Lourenço, H. R., Martin, O., and Stützle, T. (2010). Iterated local search: Framework and applications. In Gendreau, M. and Potvin, J.-Y., editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, chapter 9, pages 363–397. Springer, New York, NY, 2 edition.

Marmion, M.-E., Mascia, F., López-Ibáñez, M., and Stützle, T. (2013). Automatic design of hybrid stochastic local search algorithms. In Blesa, M. J., Blum, C., Festa, P., Roli, A., and Sampels, M., editors, *Hybrid Metaheuristics*, volume 7919 of *Lecture Notes in Computer Science*, pages 144–158. Springer, Heidelberg, Germany.

Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J., and Stützle, T. (2013). From grammars to parameters: Automatic iterated greedy design for the permutation flow-shop problem with weighted tardiness. In Pardalos, P. M. and Nicosia, G., editors, *Learning and Intelligent Optimization, 7th International Conference, LION 7*, volume 7997 of *Lecture Notes in Computer Science*, pages 321–334. Springer, Heidelberg, Germany.

Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J., and Stützle, T. (2014). Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research*, 51:190–199.

Mckay, R. I., Hoai, N. X., Whigham, P. A., Shan, Y., and O'Neill, M. (2010). Grammar-based genetic programming: A survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A., and Teller, E. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092.

Minella, G., Ruiz, R., and Ciavotta, M. (2008). A review and evaluation of multiobjective algorithms for the flowshop scheduling problem. *INFORMS Journal on Computing*, 20(3):451–471.

Nawaz, M., Enscore, Jr, E., and Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95.

Pagnozzi, F. and Stützle, T. (2017). Speeding up local search for the insert neighborhood in the weighted tardiness permutation flowshop problem. *Optimization Letters*, 11:1283–1292.

Pagnozzi, F. and Stützle, T. (2018). Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems: Supplementary material. `http://iridia.ulb.ac.be/supp/IridiaSupp2018-002/`.

Pan, Q.-K. and Ruiz, R. (2012). Local search methods for the flowshop scheduling problem with flowtime minimization. *European Journal of Operational Research*, 222(1):31–43.

Pan, Q.-K., Tasgetiren, M. F., and Liang, Y.-C. (2008). A discrete differential evolution algorithm for the permutation flowshop scheduling problem. *Computers and Industrial Engineering*, 55(4):795 – 816.

Rad, S. F., Ruiz, R., and Boroojerdian, N. (2009). New high performing heuristics for minimizing makespan in permutation flowshops. *Omega*, 37(2):331–345.

Rajendran, C. and Ziegler, H. (1997). An efficient heuristic for scheduling in a flowshop to minimize total weighted flowtime of jobs. *European Journal of Operational Research*, 103(1):129 – 138.

Rajendran, C. and Ziegler, H. (2004). Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs. *European Journal of Operational Research*, 155(2):426–438.

Ruiz, R. and Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049.

Taillard, É. D. (1990). Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1):65–74.

Taillard, É. D. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285.

Tasgetiren, M. F., Liang, Y.-C., Sevkli, M., and Gencyilmaz, G. (2007). A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem. *European Journal of Operational Research*, 177(3):1930 – 1947.

Tseng, L.-Y. and Lin, Y.-T. (2009). A hybrid genetic local search algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 198(1):84–92.

Vallada, E. and Ruiz, R. (2010). Genetic algorithms with path relinking for the minimum tardiness permutation flowshop problem. *Omega*, 38(1–2):57–67.

Vallada, E., Ruiz, R., and Framiñán, J. M. (2015). New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3):666–677.

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

Wang, C., Chu, C., and Proth, J.-M. (1997). Heuristic approaches for n/m/F/ΣCi scheduling problems. *European Journal of Operational Research*, 96(3):636–644.