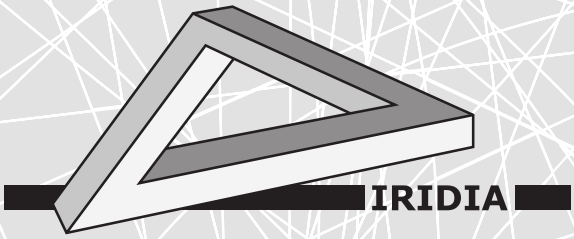

Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

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

F. Pagnozzi and T. Stützle

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2018-005

April 2018

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:
IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2018-005

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

Federico Pagnozzi, Thomas Stützle

[a]*IRIDIA, Université Libre de Bruxelles (ULB), 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 state-of-the-art 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 under the three objectives makespan, total completion time and total tardiness, which outperform some of the best algorithms obtained by a manual algorithm engineering process.

1. Introduction

Permutation flow-shop problems (PFSPs) are one of the most widely studied classes of scheduling problems [1], the arguably most studied variant being the one to minimize a schedule's makespan [2]. Other widely studied variants include those that consider minimizing the sum of the completion times of the jobs [3] or the sum of the jobs' tardiness if due dates are

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

considered. As these variants (with few exceptions such as the two machine case for makespan minimization [4]) 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 [2, 1, 3], often obtained after a significant, manual algorithm engineering effort. Even if these efforts seem often rather disconnected from each other, for a number of basic PFSP variants the best available algorithms share some similarities. For example, many recent, high-performing algorithms rely on iterated local search or iterated greedy type algorithms [2, 5, 6, 3, 7, 8].

In this paper, we show that for the PFSP variants with makespan, sum completion time and total tardiness objectives we can generate automatically very high-performing 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 [9, 10]. 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 [9, 10], 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. [11]) and the exploitation of automatic algorithm configuration techniques such as irace [12].

With these ingredients, we generate SLS algorithms for three of the most studied PFSP objectives: the minimization of (i) the makespan, PFSP_{MS} [2]; (ii) the sum of completion times, PFSP_{TCT} [3] and (iii) the total tardiness, PFSP_{TT} [7, 8]. A comparison of our automatically generated algorithms to state-of-the-art algorithms 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. Finally, 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 [2].

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} [3].

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 [13, 7, 8]. Some indication on the state-of-the-art methods to solve 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 [14], 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 [14] 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 [9, 10] 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 optimization problems.

From a high-level perspective, the algorithms that may be instantiated by EMILI follow the structure of an iterated local search (ILS) [15] 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

Algorithm 1 ILS

```
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
```

decides whether to accept a new solution or not (line 7). As explained in [9, 10], 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. A basic GRASP algorithm may be obtained by implementing a greedy randomized adaptive solution construction as perturbation, improving the resulting solution by an SLS algorithm and not applying any specific acceptance criterion. (Note that the the outline doesn't make explicit keeping track of the best solution found so far, but this is of course done.)

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 [16, 17]. Instead of directly instantiating grammar rules, we adopt the approach proposed by Mascia et al. [11], 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 ParamILS [18], SMAC [19], irace [12]. The second is that numerical parameters may be configured directly with the mechanisms available in 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; we refer to [11] for more details on this.

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 [9, 10]. One simplification is to replace the dependence on the ParadisEO framework [20] 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 [21], frameworks

<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.

for multi-objective ACO algorithms [22], or ACO algorithms for continuous optimization [23].

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

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

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 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. Construction heuristics add iteratively solution components to an initially empty partial solution, typically using a greedy approach. For convenience, we describe construction heuristics by three elements:

1. a greedy function h that assigns a value to each solution component;
2. a greedy strategy that prescribes which solution component to select depending on h;
3. and 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. An example of how the two type of heuristics build a solution is in Figure 2.

Sometimes, an iterative improvement algorithm is used to refine the solutions produced by a construction heuristic. The local search methods used in the heuristics implemented in our framework use a first improvement pivoting rule and use 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

PFSP instance of 5 jobs	$J_1, \ldots, J_5$
A partial solution	$\pi_2 = \{J_2, J_3\}$
A function $h_1(J_i)$	$h_1(J_4) > h_1(J_5) > h_1(J_1)$
A greedy strategy	$\max(h_1(j))$
Dispatching Heuristic	
$\pi_3 = \{J_2, J_3, J_4\}$	
$\pi_4 = \{J_2, J_3, J_4, J_5\}$	
$\pi_5 = \{J_2, J_3, J_4, J_5, J_1\}$	
Insertion Heuristic	
$\pi_3 =$ best among	$\{J_4, J_2, J_3\}$ $\{J_2, J_4, J_3\}$ $\{J_2, J_3, J_4\}$
$\pi_4 =$ best among all permutations produced by inserting J_5 in π_3	
$\pi_5 =$ best among all permutations produced by inserting J_1 in π_4	

Figure 2: Example displaying the difference between dispatching heuristics and insertion heuristics when building a solution.

NEH insertion heuristic [24], 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*$_{tb}$ [25] and *FRB5* [26] for PFSP$_{MS}$ and *NEH*$_{edd}$ [27] 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* [28] 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. At each search step, the local search selects a job and then inserts it in the position of the permutation that minimizes the objective function value. The search step is applied to all jobs in the order defined by the initial solution.

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* [29] heuristic builds the solution choos-

Heuristic	$h(J_i)$	Greedy Strategy	Construction	Local Search∗	Objectives
NEH [24]	$\sum_1^m p_{ij}$	$max(h(J_i))$	Insert	-	All
NEH_{tb} [25]	$\sum_1^m p_{ij}$	$max(h(J_i))$	Insert	-	PFSP_{MS}
NEH_{edd} [27]	dd_i	$max(h(J_i))$	Insert	-	PFSP_{TT}
FRB5 [26]	$\sum_1^m p_{ij}$	$max(h(J_i))$	Insert	local minimum	PFSP_{MS}
RZ [28]	$\frac{1}{g_i}\sum_{j=k}^m (m-j+1)\cdot p_{ij}$	$min(h(J_i))$	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 [29]	$\sum_{j=2}^m max\{C_{i,j-1} - C_{k,j}, 0\}$	$min(h(J_i))$	Append	-	All
LR [30]	$IT_{ik} = \sum_{j=2}^m w_{jk} max\{C_{i,j-1} - C_{k,j}, 0\}$ $\quad (n-k-2)IT_{ik} + AT_{ik}$	$min(h(J_i))$	Append	-	All
NLR	$IT_{ik} = \sum_{j=2}^m w_{jk} max\{C_{i,j-1} - C_{k,j}, 0\}$ $\quad (n-k-2)IT_{ik} + AT_{ik}$	$min(h(J_i))$	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.

ing 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 [30] 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 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 [31]. 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

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

found is returned.

In addition, we implemented two algorithms specifically designed for PFSP_{TCT} and pfsptt, respectively, namely, *iRZ* [3] and CH6 [7]. *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 that we implemented. These variants take as parameters the initial solution from which iterative improvement starts, *In*, the termination 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

Frequently, the neighborhood $N(\pi)$ of a solution π is 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, obtained by the following operators.

The **transpose** operator changes the position of two contiguous jobs, that is, a candidate solution $\pi = \{\pi(1)...\pi(i)\pi(i+1)...\pi(n)\}$ is modified to $\pi' = \{\pi(1)...\pi(i+1)\pi(i)...\pi(n)\}$ for positions $i = 1, \ldots, n-1$. This neighborhood is of size $n-1$ and can, thus, be very quickly explored.

The **exchange** operator transforms a permutation $\pi = \{\pi(1)...\pi(i)$ $...\pi(k)...\pi(n)\}$ into $\pi = \{\pi(1)...\pi(k)...\pi(i)...\pi(n)\}$ with $i, k = 1, \ldots, n; i \neq k$. 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 different position k, that is, it transforms a permutation $\pi = \{\pi(1)...\pi(i-1)\pi(i)\pi(i+1)...\pi(k)\pi(k+1)..., \pi(n)\}$ into $\pi = \{\pi(1)...\pi(i-1)\pi(i+1)...\pi(k)$

Neighborhood	Objectives
transpose	any
exchange	any
insert	any
tainsert	PFSP_{MS}
attinsert	PFSP_{TT}
atctinsert	PFSP_{TCT}
karneigh	PFSP_{TT}

Table 3: Neighborhood implementations

$\pi(i)\pi(k+1)...,\pi(n)\}$ if $k > i$ (analogously for $k < i$); we may have $i, k = 1, \ldots n$, with $i \neq k$. The insert neighborhood is of size $n(n-1)$ and contains the transpose neighborhood as a subset. Occasionally, the insert neighborhood is split into left or right insert, for which then hold $k < i$ and $k > i$, respectively.

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} uses the well known Taillard's accelerations [32] while *attinsert* and *atctinsert*, the neighborhoods implemented specifically for PFSP_{TT} and PFSP_{TCT}, use the speed-up technique presented in [33]. Finally, *karneigh* is a neighborhood relation proposed by Karabulut [8] for the PFSP_{TT} that at each step generates randomly either an insert or an exchange neighbor.

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

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

improvement over the incumbent solution 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 [15]. Parameters of *random move* are the neighborhood and *num*.

Another effective perturbation technique is used in *IG* algorithms [5]; 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 (that is, before the subsequent construction phase) [6]. 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*) [7] 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

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

Table 5: Perturbations used to generate algorithms in this study

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 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. [34]. 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 [35], 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

Acceptance Criterion	Condition	Problem	Parameters
better	$\pi' < \pi$	any	$\langle \emptyset \rangle$
diversify occasionally	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

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 [5] to solve PFSP$_{MS}$, uses a fixed temperature Metropolis condition, *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 that is multiplied with the average processing time across all jobs divided by 10. In this way, the temperature is linked to the specific problem instance to be solved. The *karacc* acceptance criterion described by Karabulut [8] 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 [36] 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 forbids entire solutions and forbids any candidate solution that is 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 position i to position j, which is used 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 gramar. From each of the resulting three grammars, we generated a set of parameters following the method outlined in [37] and limiting the maximum depth for the recursive expansion of a rule to three. This 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. (When later in the text we give values of numerical parameters, we only give the values of those whose conditions are satisfied.) As the automatic configuration method to explore these parameter spaces we use the irace package [12]. Irace implements an iterated racing approach that takes as input the algorithm to be tuned, a definition of the parameters including their respective types and domains, and a set of training instances. At each iteration, first irace creates a number of algorithm configurations based on some probabilistic sampling model. In the first iteration, algorithm configurations are generated by sampling the relevant parameters uniformly at random. The sampling distribution is updated after every iteration biasing more and more strongly the generation of candidate configurations towards the best performing algorithm candidates. At each iteration, a race is performed by evaluating the algorithm configurations instance-by-instance on training instances that are representative for the specific task to which the algorithm is to be applied. A race terminates if either its budget is exhausted or the number of surviving algorithm candidates drops below some number. The surviving candidate configurations are then used to bias the sampling model. For a detailed description of irace we refer to [12].

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 [38], 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 experiments were executed on AMD Opteron 6272 CPUs running at 2.1 GHz. Each execution of EMILI is single-thread. Both machines run CentOS 6.2 Linux.

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 tt 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.

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 [2]. Among these, the IG algorithm by Ruiz and Stützle has for a long time been the best performing algorithm for this problem [5] and only rather recently two improved variants of it have been proposed: IG_{tb} [25] and IG_{all} [6]. 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 general structure and parameter setting. The current state-of-the-art algorithm, IG_{all},

Algorithm 2 IG_{all}

Output The best solution found π^*,
π := $\mathit{FRB5}()$;
π := $\mathsf{II}(\pi, \mathit{local\ minima}, \mathit{tainsert})$;
π^* := π
while ! time is over **do**
    π' := $\mathsf{IG}_{\mathsf{tb+lsps}}(\pi, ls(\mathit{first}, \mathit{local\ minima}, \mathit{tainsert}))$;
    π' := $\mathsf{II}(\pi', \mathit{first}, \mathit{local\ minima}, \mathit{tainsert})$;
    π := $\mathsf{rsacc}(\pi, \pi')$;
    if $f(\pi') < f(\pi^*)$ **then**
        π^* := π'
    end if
end while
Return π^*

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 algorithm selected by irace, IG_{irms}, is shown in Algorithm 3 and can be compared to IG_{all}, which is shown in Algorithm 2. The parameter values of IG_{irms} are shown in Table 7. IG_{irms} can be characterized as an *IG* algorithms that uses a first improvement local search and the IG_{lsps} perturbation. In this sense, given it also uses the *FRB5* initialization, IG_{irms} is rather similar to IG_{all}. This similarity is interesting as IG_{irms} was generated without knowledge of the structure of the algorithm IG_{all}. However, there are also differences between the two algorithms beyond some settings of numerical parameters. The main differences concern the different acceptance criteria used and the pivoting rule in the local search on partial solutions. While both acceptance criteria are based on the Metropolis condition, IG_{all} keeps the temperature constant while IG_{irms} decreases it following an annealing schedule. For the local search on partial solutions, IG_{irms} uses a local search based on the best improvement pivotal rule while the local search in IG_{all} is based on first improvement. This is interesting as the choice taken in IG_{all} would be the logical one for a human designer as this choice would also be implied by the literature on this objective.

We compared the algorithms using the benchmark set of Taillard [36], which 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 $m \in \{5, 10, 20\}$. For each algorithm and each instance we measured the average relative percent-

Algorithm 3 IG_{irms}

Output The best solution found π^*,
$\pi := \mathit{FRB5}()$;
$\pi := \mathsf{II}(\pi, maxstepsorlocmin(s), tainsert)$;
$\pi^* := \pi$
while ! time is over **do**
    $\pi' := \mathit{IG}_{lsps}(\pi, ls(best, local\ minima, tainsert))$;
    $\pi' := \mathsf{II}(\pi', first, maxstepsorlocmin, tainsert)$;
    $\pi := \mathit{psa}(\pi, \pi')$;
    if $f(\pi') < f(\pi^*)$ **then**
        $\pi^* := \pi'$
    end if
end while
Return π^*

	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}

age deviation (ARPD) from the best results published on Taillard's website `http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/flowshop.dir/best_lb_up.txt`. The results of the experiments are reported in Figure 3 and in Table 8, where the values are grouped by instance size. IG_{irms} is clearly the best performing algorithm overall. Even if in absolute terms the differences are rather minor, this result is remarkable given the very advanced state of the art for the PFSP under makespan objective [2].

4.2. PFSP: Sum completion times

Over the years, various metaheuristic algorithms have been proposed to tackle this problem. These include population-based [39, 40, 41] and trajectory-based algorithms [42, 43]. The current state-of-the-art algorithm is an IG algorithm, *IGA* [3]. This algorithm uses as local search an exploration scheme inspired by the improvement phase of the *RZ* heuristic. This

	$T = 60$		$T = 120$		$T = 240$	
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.

scheme starts by considering the jobs in the order of the current solution. it then determines the best insertion for a given job 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.

	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 9: Parameter settings for ALG_{irtct}

The algorithm produced with our system, ALG_{irtct}, is a hybrid *IG* that

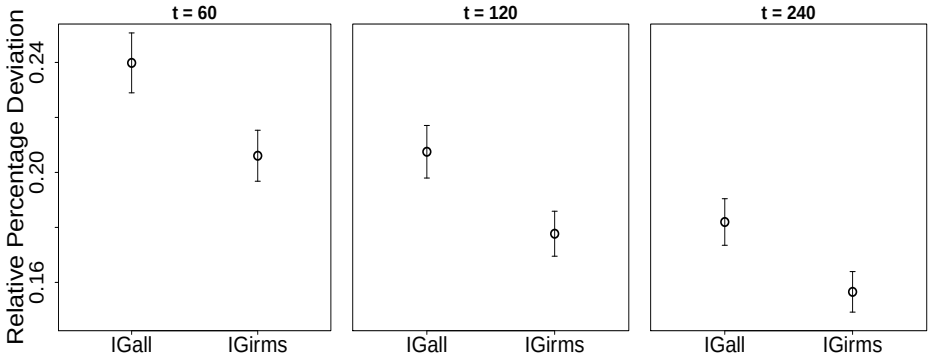

Figure 3: PFSP_{MS} comparison, Average RPD and 95% confidence intervals for $T = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

Algorithm 4 ALG_{irtct}

```
1: Output The best solution found π*,
2: π := NRZ_2();
3: π := ALGirtct2(π);
4: π* := π
5: while ! time is over do
6:     π' := IG_io(π);
7:     π' := ALGirtct2(π');
8:     π := psa(π, π');
9:     if f(π') < f(π*) then
10:        π* := π'
11:    end if
12: end while
13: return π*
```

has as an ILS as local search. The outline of ALG_{irtct} is shown in algorithm 4 and the outline of the inner ILS in algorithm 5. The parameters of both algorithms are shown in Table 9. Taking into account also the numerical parameter settings, the inner ILS has rather a diversifying effect as it has only a rather short local search phase and the cooling of the temperature is rather slow, thus leading to a frequent acceptance of worse candidate solution. Differently, the outer IG, applies a rather strong perturbation with a more restrictive acceptance criterion.

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

Algorithm 5 ALG_{irtct2}

input current solution π
output the best solution found π^*,
$\pi := \mathsf{ii}(\pi, first, local\ minima, insert)$;
$\pi^* := \pi$
while *maxsteps*() **do**
    $\pi' := \mathit{random\ move}(\pi, transpose)$;
    $\pi' := \mathsf{ii}(\pi', first, local\ minima, insert)$;
    $\pi := \mathit{psa}(\pi, \pi')$;
    if $f(\pi') < f(\pi^*)$ **then**
        $\pi^* := \pi'$
    end if
end while
return π^*

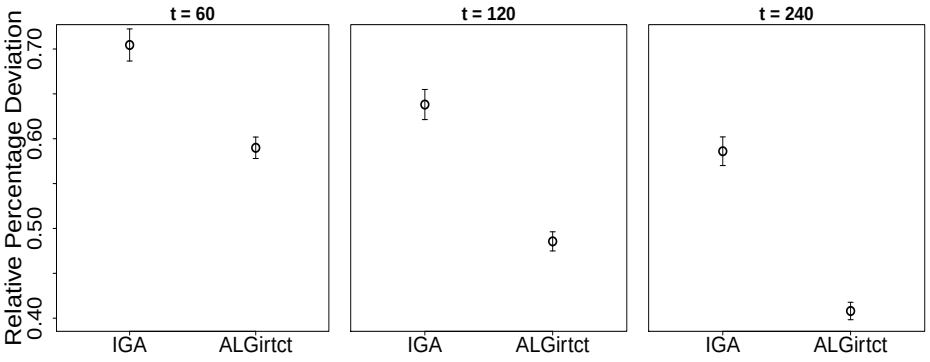

Figure 4: Average RPD and 95% confidence intervals of IGA and ALG_{irtct} for $T = 60$ (left), $T = 120$ (center) and $T = 240$ (right).

ARPD, using the best solutions reported in [3]. The results, in Figure 4 and in Table 10, clearly show that ALG_{irtct} improves over the performance of IGA, which is more visible for the higher computation time limits. For the highest computation time limit, ALG_{irtct} is able to improve the best known solution values for the instances `TA103, TA109, TA114, TA118` and `TA120`.

Recently, a new constructive heuristics for the PFSP$_{TCT}$ was proposed by [44], 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$_{MS}$ [6]) and, hence, an extension for our work would be to add this constructive heuristic as one of our algorithmic components.

	$T = 60$		$T = 120$		$T = 240$	
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 10: 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.

4.3. PFSP: Total tardiness

Several algorithms have been proposed for the PFSP$_{TT}$, ranging from simulated annealing [13] to genetic algorithms [45]. Recently, new metaheuristic algorithms have been proposed [7, 8], 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 [33]. 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 acceptance criterion is *karacc*.

The algorithm generated by irace, ALG_{irtt}, is shown in Algorithm 6 and 7. The algorithm is a two layer ILS with the outer layer being composed of

the perturbation and acceptance from TSM63 and the inner layer being a rather plain ILS algorithm similar to the inner ILS algorithm of PFSP$_{TCT}$. Analyzing the structure of the algorithm, ALG_{irtt} performs two kinds of perturbations, a default one that is relatively light with a stronger one applied after a certain number of iterations. Moreover, we can see that irace confirms the configuration choice, done for TSM63, of accepting always the perturbed solution returned by *CP*. The parameter settings of ALG_{irtt} are shown in Table 11. These parameter settings imply a difference in the usage of the *CP* perturbation: In ALG_{irtt} the *insert* neighborhood is preferred for the perturbation while the parameter settings in TSM63 the setting of parameter *pc* favors the usage of the *transpose* neighborhood for the perturbation.

	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 11: Parameter settings for ALG_{irtt}

For the tests we used the benchmark proposed in [46]. 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 and best values the ones provided with the benchmark set. From the results, shown in Figure 5, ALG_{irtt}, outperforms both TSM63 and IG_{RLS} with TSM63 being the worst algorithm.

In Table 12 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 and 250x10, where for size 50x50, TSM63 has better results than IG_{RLS} for the longer running times and for size 250x10 TSM63 performs better than ALG_{irtt}. On average, ALG_{irtt} performs the best. When focusing on specific instance sizes, ALG_{irtt} performs often best except for instance sizes 150x10, 150x30 and all instances with 250 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$_{TT}$ [47]. 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 [47] 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, [47] 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. -1.45 on average for $T = 240$ for ALG_{irtt} 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	0.36	-0.13	-0.12	-0.12	-0.37	**-0.45**	-0.46	-0.55	**-0.73**
50x30	1.50	0.03	**-0.38**	0.43	-0.70	**-1.09**	-0.41	-1.40	**-1.71**
50x50	1.75	1.43	**-0.39**	0.56	0.69	**-1.21**	-0.25	-0.02	**-1.89**
150x10	0.23	**-0.51**	0.13	-0.19	**-0.73**	-0.26	-0.57	**-0.89**	-0.60
150x30	1.47	**-1.20**	-0.65	0.47	**-2.02**	-1.63	-0.60	**-2.79**	-2.53
150x50	2.02	-0.70	**-1.22**	0.64	-1.68	**-2.35**	-0.75	-2.59	**-3.34**
250x10	0.09	**-0.80**	0.45	-0.23	**-1.10**	-0.0005	-0.61	**-1.33**	-0.43
250x30	1.12	**-1.66**	-0.37	0.37	**-2.52**	-1.37	-0.45	**-3.30**	-2.29
250x50	1.68	**-1.66**	-0.96	0.71	**-2.63**	-2.21	-0.35	**-3.54**	-3.34
350x10	0.72	0.20	**0.06**	0.53	0.08	**-0.06**	0.37	-0.02	**-0.16**
350x30	1.38	1.99	**0.16**	1.07	1.64	**-0.09**	0.80	1.32	**-0.29**
350x50	1.48	2.84	**0.33**	1.13	2.52	**0.09**	0.84	2.24	**-0.10**
Average	1.15	-0.01	**-0.25**	0.45	-0.57	**-0.89**	-0.20	-1.07	**-1.45**

Table 12: Average RDI results of TSM63, IG_{RLS} and ALG_{irtt} for the two 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 6 ALGirtt

```
1: Output The best solution found π*,
2: π := NEH_edd();
3: π := ALGirtt2(π);
4: π* := π
5: while ! time is over do
6:     π' := CP(π);
7:     π' := ALGirtt2(π');
8:     π := π';
9:     if f(π') < f(π*) then
10:        π* := π'
11:    end if
12: end while
13: Return π*
```

Algorithm 7 ALGirtt2

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

algorithm design [21, 22] demonstrate the large promise of this research direction. In our case, this progress is based on joining a flexible algorithm framework that has been developed with a focus on automatic configurability, an appropriate 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 [48].

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 impact on the design of algo-

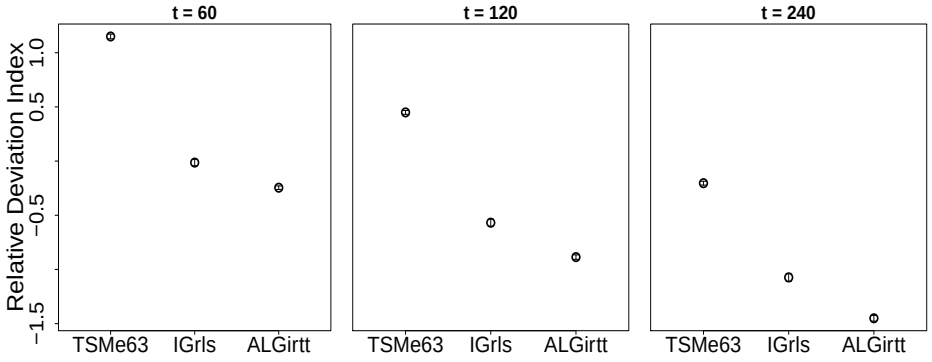

Figure 5: Average RDI and 95% confidence intervals of TSM63, TSMe63 and ALGirtt for $t = 60$(left), TSMe63 and ALGirtt for $T = 120$(center) and $T = 240$ (right).

rithms. 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.

References

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

[2] V. Fernandez-Viagas, R. Ruiz, J. M. Framiñán, 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) (2017) 707–721.

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

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

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

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

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

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

[9] M.-E. Marmion, F. Mascia, M. López-Ibáñez, T. Stützle, Automatic design of hybrid stochastic local search algorithms, in: M. J. Blesa, C. Blum, P. Festa, A. Roli, M. Sampels (Eds.), Hybrid Metaheuristics, Vol. 7919 of Lecture Notes in Computer Science, Springer, Heidelberg, Germany, 2013, pp. 144–158.

[10] M. López-Ibáñez, M.-E. Kessaci, T. Stützle, Automatic design of hybrid metaheuristics from algorithmic components, Submitted.

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

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

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

[14] H. H. Hoos, T. Stützle, Stochastic Local Search—Foundations and Applications, Morgan Kaufmann Publishers, San Francisco, CA, 2005.

[15] H. R. Lourenço, O. Martin, T. Stützle, Iterated local search: Framework and applications, in: Gendreau and Potvin [49], Ch. 9, pp. 363–397.

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

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

[18] F. Hutter, H. H. Hoos, K. Leyton-Brown, T. Stützle, ParamILS: an automatic algorithm configuration framework, Journal of Artificial Intelligence Research 36 (2009) 267–306.

[19] F. Hutter, H. H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: C. A. Coello Coello (Ed.), Learning and Intelligent Optimization, 5th International Conference, LION 5, Vol. 6683

of Lecture Notes in Computer Science, Springer, Heidelberg, Germany, 2011, pp. 507–523.

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

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

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

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

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

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

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

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

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

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

[30] J. Liu, C. R. Reeves, Constructive and composite heuristic solutions to the p//σci scheduling problem, European Journal of Operational Research 132 (2) (2001) 439–452.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[48] H. H. Hoos, Programming by optimization, Communications of the ACM 55 (2) (2012) 70–80.

[49] M. Gendreau, J.-Y. Potvin (Eds.), Handbook of Metaheuristics, 2nd Edition, Vol. 146 of International Series in Operations Research & Management Science, Springer, New York, NY, 2010.