Intl. Trans. in Op. Res. xx (2020) 1–25

Evaluating the impact of grammar complexity in automatic algorithm design

Federico Pagnozzi[a,*], Thomas Stützle[a]

[a]*IRIDIA, Université Libre de Bruxelles (ULB), Belgium*
E-mail: federico.pagnozzi@ulb.ac.be [F. Pagnozzi]; stuetzle@ulb.ac.be [T. Stützle]

Received DD MMMM YYYY; received in revised form DD MMMM YYYY; accepted DD MMMM YYYY

Abstract

Grammar-based automatic algorithm design has been shown to generate stochastic local search algorithms that compete with or outperform state-of-the-art methods. In such systems, algorithms are divided in components and a grammar is used to describe how to properly combine the components to create a working algorithm. In our approach, the grammar is converted in parameters and an automatic parameter configuration tool is used to find the best configuration. This approach allows to consider and hybridize different metaheuristic templates producing combinations never tested before, but this flexibility leads to a very large configuration space to explore. Is such complexity really needed to achieve state-of-the-art performance?
In this paper, we investigate this question by creating grammars that allow the hybridization of stochastic local search algorithms at most two, one or zero times. We use these grammars to generate algorithms for the three most studied objectives of the permutation flowshop problem: makespan, total completion time and total tardiness. The generated algorithms are compared using benchmark sets from the literature as well as a quantitative measure of algorithm complexity using a metric based on concept directed acyclic graphs. The experiments show that our system tends to generate hybridized algorithms only when they can provide a substantial performance improvement. On the contrary, when such algorithms do not improve performance, the system generates simpler algorithms regardless of the grammar used.

Keywords: Grammar-based automatic algorithm design; stochastic local search; permutation flowshop

1. Introduction

Designing stochastic local search (SLS) algorithms is not a trivial task. The process requires, first, to choose the most appropriate SLS algorithm and then to adapt the chosen algorithm to the problem. Finally, the designer has to choose the right configuration for the parameters of the algorithm. Automatic algorithm configuration has been proposed to solve the configuration problem using different techniques

*Author to whom all correspondence should be addressed (e-mail: federico.pagnozzi@ulb.ac.be).

such as continuous optimization (Hansen and Ostermeier, 2001), SLS methods (Grefenstette, 1986), the racing algorithm (Maron and Moore, 1997) and model-based approaches Mockus (1989). Nowadays several, publicly available, automatic configuration tools such as irace (López-Ibáñez et al., 2016), paramILS (Hutter et al., 2007) or SMAC (Hutter et al., 2011) can be used to optimize the performance of any algorithm.

The idea of automatic algorithm design (*AAD*) consists of expressing all these choices as parameters and then use an automatic configuration tool, like irace, to select the best configuration and, consequently, the best algorithm. This design paradigm has been called design by optimization (Hoos, 2012). In a top-down approach, the algorithmic template is fixed (e.g. iterated local search or simulated annealing) and all the design choices regarding the single components and their parameters. In a bottom-up approach, instead, different templates can be instantiated and hybridized and a context-free grammar is used to set the rules that describes how to build algorithms by combining algorithmic components.

The hybridization of different kinds of SLS algorithms has shown to be able to generate state-of-the-art algorithms when applied to permutation flowshop (Pagnozzi and Stützle, 2019; Marmion et al., 2013) as well as hybrid flowshop (Alfaro-Fernández et al., 2020), the unconstrained binary quadratic programming and the traveling salesman problem with time windows (López-Ibáñez et al., 2017). However, allowing hybridization can generate huge parameter spaces and generate algorithms with a complex, nested structure. For instance, the grammar based approach used in Pagnozzi and Stützle (2019) for the permutation flowshop problem with the makespan objective had 502 parameters.

In this paper, we try to understand if this complexity is really needed by using the *AAD* system presented in a previous paper to generate algorithms with varying levels of allowed hybridization for the three most studied objectives of the permutation flowshop problem (PFSP): makespan (PFSP_{MS}), total completion time (PFSP_{TCT}) and the total tardiness (PFSP_{TT}) (Pagnozzi and Stützle, 2019).

In order to measure algorithm complexity, we use a quantitative measure based on the algorithms' similarity calculated by a method based on directed acyclic graphs (Xu et al., 2016). This similarity metric has been proposed to evaluate the differences between different parameters configuration and it has been used to compare different algorithms generated by SATenstein (KhudaBukhsh et al., 2016). In a different publication, the same metric is used in a tool, called CAVE (Biedenkapp et al., 2018), to compare and analyzed the results of automatic algorithm configurators.

The proposed similarity metric considers each algorithm as its parameter configuration and each configuration is organized in a directed acyclic graph (DAG). When comparing two configurations, the similarity is calculated as the cost of the changes that need to be applied to one DAG to transform it into the other. In particular, a different cost is calculated for parameters that have to be deleted, inserted or modified. In addition, since often parameters are linked to others, each cost is weighted considering how many parameters will be effected by the change. Our proposition is to use this distance metric to evaluate the complexity of an algorithm by measuring its distance from the most simple algorithm that can be defined according to the grammar.

For each objective we consider two grammars, a base grammar comprising different SLS templates and one that explicitly includes an hybridized SLS as one of the available templates. For each combination of objective and grammar, we allow no hybridization, up to one level of hybridization and up to two levels. For each of these combinations we generate 10 algorithms that are compared considering solution quality and algorithm complexity for a total of 180 algorithms.

The expected result is that more complex algorithms will be generated only when they can provide

better results. In the case where a simple algorithm has better performance, we expect the generated algorithms to share, more or less, the same complexity no matter how many levels of hybridization are allowed. The experiments show that our *AAD* system generates a more complex algorithm only if it performs similarly or better than a less complex algorithm. Furthermore, the results show that the huge parameter spaces produced by allowing hybridization do not seem to generate less performing algorithms when simple algorithms perform better.

To the best of our knowledge, there is little work directly relevant to quantify the complexity of algorithm structure, let alone similarity of algorithm configurations. In a relevant work, algorithm similarity is assessed using the notion of edit distance (Nikolić et al., 2009). Given two strings of texts, the edit distance is defined as the minimum number of edit operations needed to change one string of text to another. Although it is possible to turn two parameter configurations into strings of text, this method would not suit our needs. In fact, simply turning the parameter configurations in strings of text would ignore the dependencies between parameters set by conditional parameters, that is, parameters whose value depends on the value of other parameters.

The structure of the paper is as follows. In Section 2, the permutation flowshop problem and the objectives tackled in this paper are described. In Section 3, we present the automatic design system and how the grammar influence the algorithms. The experimental setup as well as the experiment results are described in Section 4. Finally, in Section 5, we summarize the results and report our conclusions.

2. Permutation flowshop problem

In the permutation flowshop problems there are n jobs that have to be processed on m machines. Each job has to be processed on each machine in order without preemption. The processing time of job i on machine j is defined as $p_{i,j}$. The solution to a PFSP instance consists in a permutation $\{\pi(1), ..., \pi(i), ..., \pi(n)\}$ that specifies the order in which the jobs should be processed. The completion time of job i on machine j, $C_{\pi(i),j}$, is calculated as

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

In this work, we consider three of the most common objectives: makespan, total completion time or sum of completion time and the total tardiness. The makespan C_{max} is calculated as the completion of the last job on the last machine, that is,

$$C_{max} = C_{\pi(n),m}.$$

The total completion time C_{tct} is calculated as the sum of the completion times of all jobs on the last machine

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

The tardiness of a job $\pi(i)$ is calculated as $T_{\pi(i)} = max(C_{i,m} - d_{\pi(i)}, 0)$, where $d_{\pi(i)}$ is the due date of job $\pi(i)$. Consequently, the total tardiness, C_{tt}, is calculated as

$$C_{tt} = \sum_{i=1}^{n} T_i,$$

that is, the sum of the tardiness of all jobs. For all three of these objectives the permutation flowshop is $\mathcal{NP}$-hard (with the exception of the two machine case for the makespan objective (Johnson, 1954)). The current state-of-the-art algorithms for these objective have been developed using automatic algorithm design (Pagnozzi and Stützle, 2019). In particular, the algorithm generated for the makespan objective is an iterated greedy algorithm quite similar to the previous state of the art (Dubois-Lacoste et al., 2017). On the contrary, the algorithms generated for the total completion time and total tardiness objectives are hybridized SLS algorithms composed of two nested SLS.

3. Methodology

3.1. Automatic algorithm design

In the literature, two main approaches are reported: top-down and bottom-up. In the first, top-down, the SLS algorithm is fixed, and only singular components can be changed. For instance, a simulated annealing algorithm where different choices can be made regarding its major design choices such as neighborhood exploration, cooling scheme and acceptance criterion. Examples of the top-down approach are the SAT solver SATenstein (KhudaBukhsh et al., 2016), the MOACO framework for multi objective ant colony optimization algorithms (López-Ibáñez and Stützle, 2012) as well as other algorithmic frameworks for multi objective evolutionary algorithms (Bezerra et al., 2019, 2020), simulated annealing (Franzin and Stützle, 2019) and iterated local search (ILS) (Brum and Ritt, 2018b,a; De Souza and Ritt, 2018b,a). This approach is less complex and the number of different categorical combinations is relatively low.

The approach followed by our system is the bottom-up approach. This approach uses a general template that allows, by using the proper parameter setting, to instantiate several different SLS algorithms. Moreover, this approach allows SLS algorithm hybridization, where different SLS algorithms can be combined. For instance, an ILS algorithm could be configured to use a tabu search or a simulated annealing as local search. This general template approach combined with hybridization can possibly lead to new, never seen combinations. In early examples, this approach has been used in a system based on irace as automatic configurator and the ParadisEO framework (Humeau et al., 2013) for the algorithmic components. With this system, new state-of-the-art algorithms were generated for the permutation flowshop problem with the weighted tardiness objective (Marmion et al., 2013), the unconstrained binary quadratic programming and the traveling salesman problem with time windows (López-Ibáñez et al., 2017). In more recent examples, a system based on irace and the EMILI framework has been used to generate new state-of-the-art algorithms for problems such as the permutation flowshop problem and the hybrid flowshop problem (Pagnozzi and Stützle, 2019; Alfaro-Fernández et al., 2020). Another automatic design method is the one related to generative hyperheuristics (Burke et al., 2019). These methods build heuristics and other algorithmic components based on techniques such as genetic programming (Koza, 1992), grammatical evolution (Burke et al., 2012) and gene expression programming (Sabar et al., 2015).

In order to support the bottom-up approach, we need an algorithmic framework that is flexible enough

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
```

to allow the definition of algorithmic components and templates with minimal effort. We use the EMILI framework that was designed exclusively for this purpose. A key complication introduced by this approach is that the configuration space is larger and there are parameter configurations that do not represent a legal algorithm, that is, an algorithm that can be instantiated. Additionally, some legal configuration may be undesirable, such as an ILS without a perturbation. To solve this problem, grammars have been proposed to limit the search space to only legal configurations (Mascia et al., 2014).

The EMILI framework design is based on a generalized version of a widely known SLS algorithm, iterated local search (ILS). The outline of this algorithm is shown in Algorithm 1. The ILS starts with an initial solution typically generated by a construction heuristic and combines an intensification phase, to reach quickly a local minimum, and a diversification phase, to explore the search space and try to escape local minima. The balance between intensification and diversification is controlled by an acceptance criterion. Finally, the algorithm stops when a termination condition is reached, typically dependent on running time or the number of algorithm iterations.

In its implementation for the PFSP, the framework considers, typically, first improvement, best improvement and variable neighborhood descend (VND) for the intensification phase. Compared with the ILS template, these algorithms require only an initial solution, a termination condition and to define the neighborhood to explore or the set of neighborhood definitions. In Table 1 we report the components used in this study divided by type. Given the focus of this paper, we will not provide a detailed description of all the components. For such description we remind the reader to our previous publication (Pagnozzi and Stützle, 2019).

3.2. Grammars

In the following, we describe in detail how the rules composing a context-free grammar are translated into parameters. Context-free grammars are defined as a four-tuple $G = (V, \Sigma, R, S)$, where V represents a set of non-terminal symbols, Σ is a set of terminal symbols, R is a set of rules that maps every non-terminal symbol in V to the set $(V \cup \Sigma)^*$, and S specifies the starting non-terminal symbol. The rules in Σ are also known as production rules and link every non-terminal symbol to a combination of terminal and/or non-terminal symbols. Applying one production rule consists in replacing a non-terminal with the combination of symbols indicated by the rule. While non-terminal symbols can be seen as variables, terminal symbols represent values that can be either strings of text or numbers. An example of production rule is the following

Type	Component	Parameters
Construction Heuristics	*NEH* (Nawaz et al., 1983), *NEH$_{tb}$* (Fernandez-Viagas and Framiñán, 2014)	-
	NEH$_{edd}$ (Kim, 1993), *LR* (Liu and Reeves, 2001), *NLR*	-
	FRB5 (Rad et al., 2009), *RZ* (Rajendran and Ziegler, 1997), *NRZ*, *NRZ$_2$*,*SLACK*,	-
	NEH$_{rs}$, *BS* (Fernandez-Viagas and Framiñán, 2017), *BSCH* (Fernandez-Viagas et al., 2018)	-
Iterative improvements	First Improvement	$\langle In, T, N\rangle$
	Best improvement	$\langle In, T, N\rangle$
	VND	$\langle P, In, T, \{N_1, ..., N_k\}\rangle$
	iRZ (Pan and Ruiz, 2012)	$\langle In\rangle$
	ALS (Brum and Ritt, 2018a)	$\langle l_1, l_2\rangle$
Neighborhoods	*transpose*, *exchange*, *insert*, *binsert*	-
	finsert, *atctinsert* (Pagnozzi and Stützle, 2019), *attinsert* (Pagnozzi and Stützle, 2019)	-
	twinsert, *karneigh* (Karabulut, 2016)	-
Termination criteria	*local minimum*	-
	maxsteps	$\langle maxi\rangle$
	maxstepsorlocmin	$\langle maxi\rangle$
Perturbation criteria	*random_move*	$\langle N, num\rangle$
	vr_move	$\langle\{d, num, (N_1, ..., N_k)\}\rangle$
	IG$_{lsps}$ (Dubois-Lacoste et al., 2017)	$\langle d\rangle$
	IG,*IG$_{rs}$* (Ruiz and Stützle, 2007)	$\langle d\rangle$
	IG$_{io}$ (Pagnozzi and Stützle, 2019)	$\langle d\rangle$
	mrsilsp (Wang et al., 2014)	$\langle p\rangle$
	CP3 (Li et al., 2015)	$\langle d, \omega, pc\rangle$
Acceptance criteria	*better*	$\langle\emptyset\rangle$
	improve plateau	$\langle s_t, s_n\rangle$
	ft (Metropolis et al., 1953)	$\langle T\rangle$
	psa (Metropolis et al., 1953)	$\langle T_s, T_e, \beta, it\rangle$
	sa (Metropolis et al., 1953)	$\langle T_s, T_e, \beta, \alpha, it\rangle$
	rsacc (Ruiz and Stützle, 2007)	$\langle T_p\rangle$
	karacc (Karabulut, 2016)	$\langle T_p\rangle$

Table 1: Algorithmic components implemented in the EMILI framework that were used in this study

$$<\text{A}> :: \text{‘b’} \mid \text{‘c’},$$

where the non terminal $<$A$>$ can be expanded either as the terminal ‘b’ or ‘c’. Every legal sentence according to G can be produced by applying the production rules in R starting with S.

Once defined, the grammar rules can be converted in two types of parameters, numerical or categorical. Numerical parameters are used to represent numbers that can be either integer or real. Categorical parameters are used to represent a set of values that are unrelated and cannot be ordered. Usually, this type of parameters is used to represent different design choices. Although it is not used in this approach, most *AAC* tools supports also ordinal parameters which represent parameters with values that can be ordered but not compared. Additionally, parameters can be conditional, which means that they are active, that is, they are required to assume a value if a certain condition is verified; otherwise, their value is not considered and the parameter is inactive. Usually, the most common condition is that another parameter assumes a specific value. For instance, considering the termination condition of a SLS algorithm, if the parameter controlling the termination condition is set to stop the algorithm after reaching a local minimum, the numerical parameter controlling the maximum number of iterations is inactive.

When converting grammar rules to parameters, we have to take into account several cases. Simple rules, that is production rules that presents only terminal symbols on the right side, like the one shown above for ‘A’, can be directly converted into parameters. These rules are translated either in numerical

or categorical parameter. Rules that have non terminals on the right side, recursive rules and groups of rules arranged in a loop, needs more than one parameter to be converted. These rules need a parameter for the non terminal on the left side of the rule and then a new parameter for each non terminal on the right side. Recursive rules and loops can be applied more than once, thus, a new set of parameters has to be created every time the recursion is applied. For instance, lets consider the following production rule

$$\text{<C> :: 'b' <C> | 'd'}, \tag{1}$$

where <C> can either be 'd' or 'b' and then there is a recursive call to <C>. A parameter P_c is created for <C> that comprises the choice between 'b' and 'd'. The recursion is then treated by creating a new parameter P_{c2} in the same way as P_c that can assume a value only if P_c assumes the value 'b'. This process is stopped after R_c times, otherwise it would be impossible to transform the grammar in a set of parameters making the use of parameter tuners impossible. For instance, when the limit R_c is set to 1, the production rule <C> is transformed in a set of two production rules, shown in Equation 2. The two rules are directly translated in a set of two parameters with the parameter for <C2> that has to be set when <C> assumes the value 'b'.

$$\begin{array}{l} \text{<C>} \quad \text{:: 'b'} \quad \text{<C2>} \quad | \quad \text{'d'} \\ \text{<C2>} \quad \text{:: 'd'} \end{array} \tag{2}$$

A similar consideration is made when taking into account loops, with the difference that all the rules involved in the loop have to be replicated. We can show an example by modifying the example in Equation 1 as

$$\begin{array}{l} \text{<C> :: 'b'} \quad \text{<D>} \quad | \quad \text{'d'} \\ \text{<D> :: 'a'} \quad \text{<C>} \end{array} \tag{3}$$

In Equation 3, there is a loop composed of <C> and <D>. Considering a value of $R_c = 1$, the loop is translated in the group of production rules shown in Equation 4.

$$\begin{array}{l} \text{<C>} \quad \text{:: 'b'} \quad \text{<D>} \quad | \quad \text{'d'} \\ \text{<D>} \quad \text{:: 'a'} \quad \text{<C2>} \\ \text{<C2> :: 'b'} \quad \text{<D2>} \quad | \quad \text{'d'} \\ \text{<D2> :: 'a'} \end{array} \tag{4}$$

The grammar used for this study is shown in Figure 1, where the starting production rule is <SLS>. In the figure we omitted simple rules and numerical parameters to better highlight the parts of the grammar that influence the complexity. The general ILS implemented in the EMILI framework is described by the rule <ILS>, while the hybridization is allowed by the loop composed of the rules <ILS> and <LocalSearch>. When converting the grammar into parameters, expanding this loop requires the duplication of all the rules in Figure 1 with the exception of <SLS>, resulting in a huge increase in the number of parameters to tune.

<SLS>	::=	<ILS> | <TabuSearch>
<ILS>	::=	'ils' <LocalSearch> <Termination> <Perturbation> <Acceptance>
<TabuSearch>	::=	<FirstTabuSearch> | <BestTabuSearch>
<FirstTabuSearch>	::=	'tabu' 'first' <InitialSolution> <Termination> <Neighborhood> <TabuMemory>
<BestTabuSearch>	::=	'tabu' 'best' <InitialSolution> <Termination> <Neighborhood> <TabuMemory>
<LocalSearch>	::=	<FirstImprovement> | <BestImprovement> | <TabuSearch> | <VND> | <ILS> | <EmptyLocalSearch>
<FirstImprovement>	::=	'first' <InitialSolution> <Termination> <Neighborhood>
<BestImprovement>	::=	'best' <InitialSolution> <Termination> <Neighborhood>
<EmptyLocalSearch>	::=	'nols' <InitialSolution>
<VND>	::=	'vnd' <firstVND> | <bestVND>
<firstVND>	::=	'first' <InitialSolution> <Termination> <neighborhoods>
<bestVND>	::=	'best' <InitialSolution> <Termination> <neighborhoods>
<neighborhoods>	::=	<Neighborhood> <neighborhoods> | $\emptyset$
<Perturbation>	::=	$<IG>$ | $<IG_{lsps}>$ | $<random\_move>$ | 'noper'
$<IG_{lsps}>$	::=	'IG_{lsps}' <SimpleImprovement>
<SimpleImprovement>	::=	<FirstImprovement> | <BestImprovement>

Fig. 1: Template of the context-free grammar used for this study.

In empirical experiments conducted, while developing the EMILI framework, it was observed that, with the grammar in Figure 1, there was a low probability of generating hybridized SLS algorithms even when such algorithms would provide better performance. This may be explained by the fact that in order to generate an hybridized algorithm, two parameters have to be set with the right value: <SLS> has to be set to <ILS> and <LocalSearch> has to be either <ILS> or <TabuSearch>. For this reason, the grammar that was used to generate SLS algorithms in Pagnozzi and Stützle (2019) included an explicit rule for generating hybridized SLS algorithms. This rule, shown in Figure 2 as <HSLS>, is almost equal to <ILS> with the only change being the substitution of <LocalSearch> with <ILS>. The effect is to increase the probability of generating an hybridized algorithm, since now it can be selected with one parameter as <ILS> and <TabuSearch>.

The additional rules omitted in Figure 1, describe the different components available for each component type. These components are shown in Table 1. The grammar in Figure 1 was adapted to each objective by adding the specific set of objective specific components. These components represent the only differences between the three grammars prepared for each PFSP objective. Additionally we considered, for each objective, also the grammar with the modification shown in Figure 2, resulting in two grammars for each objective and six grammars in total.

From these grammars, we generate three set of parameters by varying the value of R_c from zero to two that, for the grammars with the explicit hybridization rule, corresponds to going from one to three. This is another effect of the rule <HSLS>, where an hybridized algorithm is defined as a different rule. Defined in this way, <HSLS> is not a loop and, thus, it is converted to parameters even when R_c is set to zero. As a result when setting $R_c = 0$, the set of parameters generated from the grammar with

```
<SLS>    ::=  <ILS> | <TabuSearch> | <HSLS>
<HSLS>   ::=  'ils' <ILS> <Termination> <Perturbation>
```

Fig. 2: The changes made to the grammar in Figure 1 to increase the probability of generating an hybridized SLS algorithm.

	$R_c = 0$	$R_c = 1$	$R_c = 2$	$R_c = 3$
PFSP$_{MS}$	91	167	243	-
PFSP$_{MS}$ *	-	204	356	508
PFSP$_{TCT}$	97	179	261	-
PFSP$_{TCT}$ *	-	207	371	535
PFSP$_{TT}$	99	187	275	-
PFSP$_{TT}$ *	-	225	401	577

Fig. 3: Number of parameters generated for each objective when converting the grammar with values of R_c from one to three.

additional hybridization is similar to set generated from the base grammar with $R_c = 1$. In Table 3, for each objective, we report the number of parameters generated from the grammar for the four values of R_c. The grammars with the explicit hybridization rule are indicated with *. Considering the same objective, the huge difference in the number of parameters is due to all the parameters that needs to be created by the added rule <HSLS> and the loops it creates. For $R_c = 1$, the different values for the three objectives are due to the different number of objective specific components. In the other two cases, the huge increase in the number of parameters is due to having to handle rule <ILS> and the loop it creates. Having to insert new parameters also increases the initial differences among the size of the grammars.

3.3. Directed acyclic graph based complexity metric

The complexity metric used in this work is derived by a similarity metric proposed to evaluate different parameter configurations (Xu et al., 2016). In this metric, each parameter configuration is represented as a concept DAG.

A concept DAG, G, is defined as a six-tuple $G = (V, E, L^v, R, D, M)$ where V and E contain, respectively, the set of nodes and directed edges and represent an acyclic graph, L^v is a set of node labels composed of text strings, R represent a root node, D is the set of all node labels and, finally, M is an injective mapping that assigns to every node in V a label from L^v.

Every node of the DAG represents a parameter and every edge models a dependency relation among them. If there is an edge from node A to node B it means that B needs A to be set in order to be used.

In the original proposition of this representation, the root node R is an artificial node introduced to link together all the parameters. However, in our case, due to the use of a grammar representation, all the parameter configurations have a tree structure and so the root node is the one representing the first derivation rule of the grammar.

The distance between two parameter configurations is calculated as the cost of transforming the DAG representing the first in the one representing the second configuration. The process of transforming one DAG into another is broken down in a series of four types of operations: deletion, insertion, relabelling and moving. A cost is defined for each operation and the distance is defined as the sum of the costs of all the operations needed to transform one DAG into another. The cost of deleting one node is defined as

$$C(delete(v)) = \frac{1}{|V|} \cdot (height(\textit{DAG}) - depth(v) + 1 + |\textit{DE}(v)|),$$

where $height(\textit{DAG})$ is the height of the DAG, $depth(v)$ is the depth of node v and $\textit{DE}(v)$ is the set of the nodes that depend on v. The insertion cost is calculated as

$$C(insert(u, v)) = \frac{1}{|V|} \cdot (height(\textit{DAG}) - depth(u) + 1 + |\textit{DE}(v)|),$$

where node v is inserted under node u. The moving cost is given by

$$C(moving(u, v)) = \frac{|V| - 2}{2 \cdot |V|} \cdot [C(delete(v)) + C(insert(u, v))],$$

where $|V|$ has to be greater than 2. The relabelling operation models the situation where two parameter configurations have the same parameter with a different value. The cost of this operation is calculated as

$$C(relabel(v, l^v, l^{v^*})) = [C(delete(v)) + C(insert(u, v))] \cdot s(l^v, l^{v^*}),$$

where u is the parent node of v, l^v is the old label and l^{v^*} is the new label; $s(l^v, l^{v^*})$ is a measure of the distance between the two labels. This measure is calculated in different ways depending on the type of the parameter. For parameters representing continuous values $s(l^v, l^{v^*}) = |l^v - l^{v^*}|$. For parameters representing a discrete set of k values $s(l^v, l^{v^*}) = |\frac{num(v,v^*)}{k-1}|$, where $num(v, v^*)$ is the number of intermediate values between v and v^*. Finally, for categorical parameters $s(l^v, l^{v^*}) = 0$ if the parameters have the same label, and 1 otherwise. As in Xu et al. (2016), when calculating the distance the $moving$ operator was never used.

In general, configurations can be represented with very general graphs composed of different connected components depending on how many conditional parameters are defined. When calculating the distance, the series of operators and order in which they are applied influence the final value since operators may change the height of the DAG, the depth of a node, or the number of dependencies. Consequently, calculating the distance between two configurations means finding the minimal transformation cost, which may be a computational expensive task. In this work, we exploit some characteristics of the considered *AAD* system to make the calculation of the distance quite straightforward.

In the case of grammar based AAD, all parameters are conditional with the exception of the one created from the starting rule of the grammar. For this reason, all the parameters and dependencies can be represented in a single general tree, where the root node is the parameter representing the starting rule of the grammar. Moreover, DAGs built from algorithm configurations can be seen as a subtree of this general tree. We exploit this underlying tree structure by defining $|\textit{DE}(v)|$, $height(\textit{DAG})$ and $depth(v)$ referring to the general tree. In particular, $|\textit{DE}(v)|$ is the number of child nodes of v in the general tree, while $height(\textit{DAG})$ and $depth(v)$ are, respectively, the height of the general tree and the depth of node

<A> ::= 'a_1' <B> | 'a_2' <C>
<B> ::= 'b_1' <D> | 'b_2' <D>
<C> ::= 'c_1' <E> | 'c_2' <E>
<D> ::= 'd_1' | 'd_2'
<E> ::= 'e_1' | 'e_2' | 'e_2'

Fig. 4: A simple context-free grammar comprising five non-terminal and eleven terminal symbols.

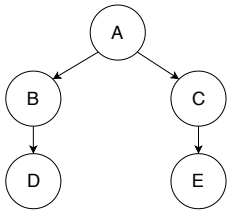

Fig. 5: Graph representing the parameters and the dependencies generated from the grammar in Figure 4

Configuration	A	B	C	D	E
C_1	'a_1'	'b_1'	-	'd_2'	-
C_2	'a_2'	-	'c_2'	-	'e_3'

Fig. 6: Two possible configurations for the parameters derived from the grammar in Figure 4.

v in the general tree. In this way, the order in which operators are applied does not change the distance measured.

To better explain how the distance between two configurations is calculated, we consider the grammar in Figure 4. The grammar does not have any recursive rule or loops, so that it can be transformed in parameter space using five categorical parameters, $\{A, B, C, D, E\}$, where B and C depend on parameter A, and D and E depend on, respectively, B and C. The graph, generated by considering only the parameters and the dependencies, is shown in Figure 5. Let us calculate the distance for two configurations C_1 and C_2, shown in Figure 6. From the tree in Figure 5 we can see that $height(C_1) = height(C_2) = 3$ and $|\boldsymbol{DE}(A)| = 2$, $|\boldsymbol{DE}(B)| = |\boldsymbol{DE}(C)| = 1$ and $|\boldsymbol{DE}(D)| = |\boldsymbol{DE}(E)| = 0$. The calculation of the distance between these two configurations can be decomposed as the sum of the different operators needed to transform C_1 in C_2. More precisely,

$$d(C_1, C_2) = relabel(A, \text{'}a_1\text{'}, \text{'}a_2\text{'}) + delete(B) + insert(C, A) + delete(D) + insert(E, C)$$

which, after applying the operators, becames

$$d(C_1, C_2) = 2.2 + 0.6 + 0.4 + 0.2 + 0.4 = 3.8.$$

The complexity metric we consider in this work is derived by this distance by evaluating the complexity as the distance from a simple baseline algorithm. This algorithm, called A_{bs}, was chosen as the one with the most simple structure that can be instantiated with the grammar and still be considered a local search algorithm. A_{bs} is an ILS that has a first improvement local search that stops once it reaches a local minimum. A_{bs} does not have a perturbation and only accepts improving solutions. Practically, A_{bs} is an iterated improvement algorithm that stops when a local minimum is reached.

4. Experimental Results

Considering the grammar with and without explicit hybridization rule, six parameter sets were generated for each objective allowing 0, up to 1 and up to 2 levels of recursion. For each level, 10 tuning sessions were conducted for a total of 60 sessions per objective. In each session, irace was executed two times, each time with the same budget of 10^5 experiments. The two executions were in succession in the sense that the second execution of irace was seeded with the best configurations produced during the first execution. All the irace sessions used a training set of 40 instances generated using the procedure described by Minella et al. (2008). The training set is composed of five instances with number of jobs in 50, 60, 70, 80, 90, 100 and 20 machines plus five instances with size 250×30 and five 250×50.

All the experiments were carried out on a computing cluster consisting of 32 Opteron 6272 CPUs running at 2.1 Ghz for a total of 512 cores, with 32 GB of RAM memory per CPU. Requiring 30 cores for each run, each execution of irace lasted about six days. Considering two irace execution per grammar and R_c value, collecting the data for this study required in total 360 irace executions. Running this amount of experiments one after the other would have required almost six years. Thanks to the high core count of the computing cluster we were able to run, at the same time, ten irace executions reducing the computing time to seven months. Additionally, the algorithms generated during the experiments are compared with the three state-of-the-art automatically generated algorithms and the three best manually designed algorithms. The three state-of-the-art algorithms are: IG_{irms} for PFSP$_{MS}$, ALG_{irtct} for PFSP$_{TCT}$ and ALG_{irtt} for PFSP$_{TT}$ (Pagnozzi and Stützle, 2019). The three manually designed algorithms are: IG_{all} (Dubois-Lacoste et al., 2017) for PFSP$_{MS}$; MRSILS(BSCH) (Fernandez-Viagas and Framiñán, 2017) for PFSP$_{TCT}$ and IAras for PFSP$_{TT}$ (Fernandez-Viagas and Framiñán, 2018).

All these algorithms are implemented in the EMILI framework and they use their original configuration, with the exception of ALG_{irtct} and ALG_{irtt}. Since the development of ALG_{irtct} and ALG_{irtt}, *BSCH* and *BS*, two new initial solution heuristics based on beam search (Fernandez-Viagas and Framiñán, 2017, 2018) have been proposed for, respectively, PFSP$_{TCT}$ and PFSP$_{TT}$. These heuristics show excellent performances and are used in the manually developed algorithms as well as in the algorithms generated during the experiments. Therefore, to have a better comparison, ALG_{irtt} and ALG_{irtct} were modified to use these components. In the following the modified versions of these algorithms will be indicated as, respectively, $ALG_{irtct}(BSCH)$ and $ALG_{irtt}(BS)$. A more detailed comparison between manual and automatically designed algorithms for the permutation flowshop problem is available in Pagnozzi and Stützle (2019).

From each tuning session we take the best algorithm as indicated by irace. The complexity metric was implemented using the R language, so that it can better interact with irace. For each objective, the algorithms are grouped by level of recursion and grammar resulting in six groups of 10 algorithms. For all algorithms we report the complexity calculated as outline in the previous section and the performance calculated on a benchmark set specific for each objective. The Taillard benchmark (Taillard, 1993) was used for PFSP$_{MS}$ and PFSP$_{TCT}$, which comprises of 120 instances divided in groups of 10 composed by all the combination of number of jobs $n \in \{20, 50, 100\}$ and number of machines $m \in \{5, 10, 20\}$ plus three groups with size $(n, m) \in \{(200, 10), (200, 20), (500, 20)\}$. For PFSP$_{TT}$ instead, we used the benchmark proposed in Vallada et al. (2008), which consists of 540 instances dived in groups of 45 with jobs $n \in \{50, 150, 250, 350\}$ and machines $m \in \{10, 30, 50\}$. Algorithm's performance is reported using the relative percentage deviation (RPD) for PFSP$_{MS}$ and PFSP$_{TCT}$, which is calculated as $\frac{S-S_b}{S_b}$,

where S is the quality of the solution found and S_b is the best known, or sometimes optimal, solution. Since in some cases the best solution can be 0, for PFSP$_{TT}$ we use the relative deviation index (RDI) that is calculated as $\frac{S-S_b}{S_w-S_b}$, where S_w is the worst solution found. Additionally, for each objective, the Wilcoxon paired test is used to assess the significance of the differences between the average performance of the algorithms generated by the two grammars. In particular, the test is executed over the average performance of the best ten algorithms generated by the base grammar and the ten generated by the grammar with additional hybridization. All algorithms are executed for the same amount of time calculated as $n \cdot (m/2) \cdot t$ milliseconds, where n and m are, respectively, the number of jobs and machines and t is set to 60.

Algorithm 2 A1

Input Given a Problem Definition π,
Output The best solution found s^*,
s := init()
s^* := s
while ! termination criterion **do**
    s' := perturbation(s)
    s'' := iterative improvement(s')
    s := acceptance(s, s'')
    if $f(s) < f(s^*)$ **then**
        s^* := s'
    end if
end while
Return s^*

Algorithm 3 A2

Input Given a Problem Definition π,
Output The best solution found s^*,
s := init()
s^* := s
while ! termination criterion **do**
    s' := $IG_{lsps}(s, local\ search)$
    s'' := iterative improvement(s')
    s := acceptance(s, s'')
    if $f(s) < f(s^*)$ **then**
        s^* := s'
    end if
end while
Return s^*

4.1. Makespan

The results for the makespan objectives are shown in Figure 7. The figure show the correlation between complexity and average RPD for the base grammar on the left and the grammar with additional hybridization on the right. In the makespan case, the algorithms generated allowing one and two recursions are not so different from the ones obtained without recursion. In fact, the difference between the average performance of the ten best algorithms for both types of grammar is not significant. The majority of the algorithms generated considering the three grammars are versions of the algorithm A2, shown in Algorithm 3, which is, an *IG* algorithm using an *IG* perturbation with local search (IG_{lsps}), either first improvement or best improvement, on the partial solution. The two less performing algorithms, the two squares at the top in Figure 7 (a), are, instead, two versions of the A1 algorithm, shown in Algorithm 2, using the IG_{io} perturbation.

The algorithms generated using the grammar with explicit recursion, in Figure 7 (b), follow, with the exception of four algorithms, the same classification. There is a first group of A1 algorithms characterized by an iterated greedy perturbation and a second group of A2 algorithms using the IG_{lsps} perturbation. Considering the remaining four algorithms, one is a version of the algorithm A3, shown in Algorithm 4, that is an ILS that uses a *VND* as local search and the IG_{lsps} perturbation. Given the performances of the

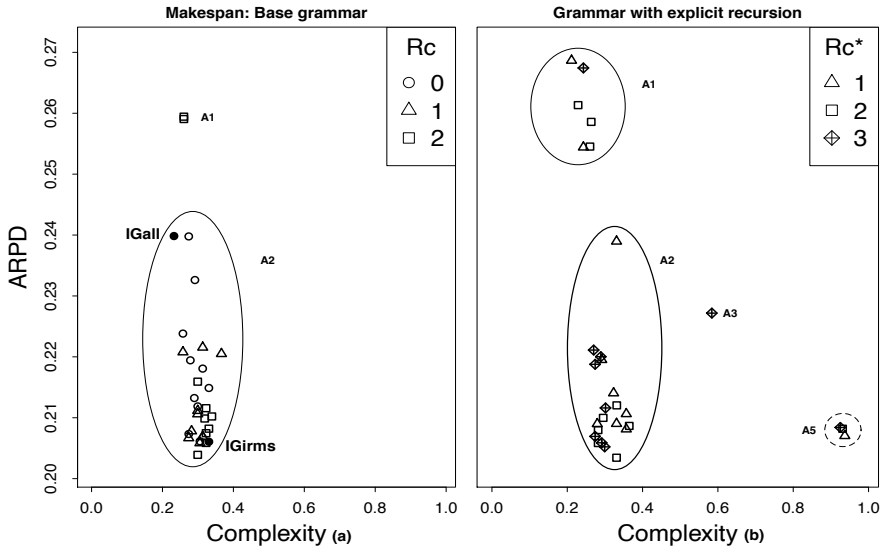

Fig. 7: Correlation between ARPD and algorithm complexity for the PFSP$_{MS}$ considering three levels of maximally allowed recursion. On the left, there are the results for the base grammar while on the right there are the results for the grammar with the additional hybridization rule. Circles indicate algorithms generated not allowing any hybridization, with the triangles one level is allowed and with the squares two levels are allowed. With Rc^* we indicate the adjusted value of Rc after considering the explicit recursion as explained at the end of Section 3.2.

majority of the A2 algorithms, the *VND* does not look better than a simple iterative improvement algorithm. The last three algorithms are hybridized algorithms and can be described as versions of algorithm A5, shown in Algorithm 6, where one ILS algorithm uses another ILS to improve the current solution after the perturbation.

The manually designed algorithm, IG_{all} (Dubois-Lacoste et al., 2017), as well as the current state-of-the-art, IG_{irms}, belong to the A2 algorithms. IG_{all} (shown in Figure 7 (a)) has better performances than the A1 algorithms thanks to the IG_{lsps} perturbation but when considering the A2 group, IG_{all} is one of the worst performing algorithms. This result further highlights the performance improvements that can be achieved by *AAD*. IG_{irms} instead (shown in Figure 7 (a)), sits at the bottom of the A2 group with seven other algorithms showing a better mean, although this difference is not statistically significant according to the Wilcoxon test.

Finally, even considering the different number of parameters due to the different R_c values, we can see that the relative performance of the generated algorithms is quite close. This result suggests that the budget allocated for the experiments is adequate to reach good configurations. The lack of hybridized algorithms, with just three generated over sixty, can be explained by the fact that simple algorithms represent a very deep local minimum that irace encounters quite early in the search. Ultimately, this result is not unexpected. In fact IG_{all} as well as IG_{irms} are *IG* algorithms with the optimization of the

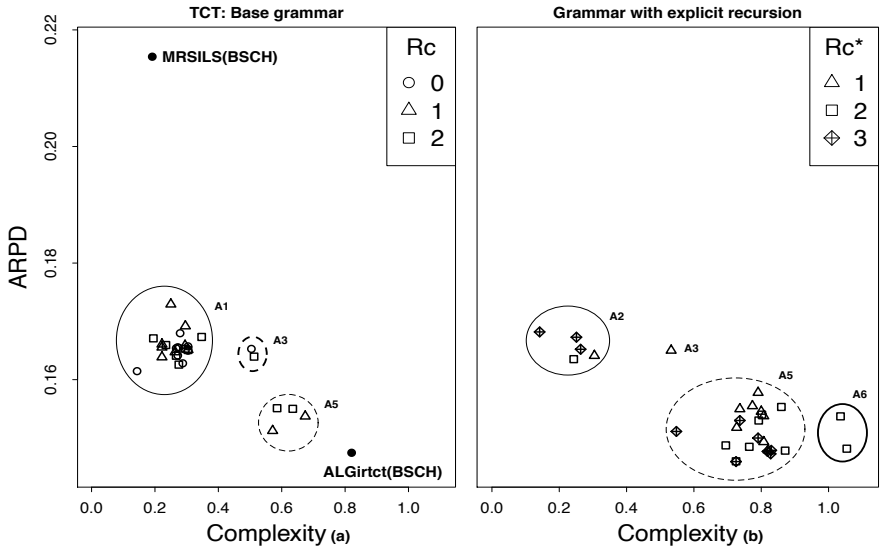

Fig. 8: Correlation between ARPD and algorithm complexity for the PFSP$_{TCT}$ considering three levels of maximally allowed recursion. On the left, there are the results for the base grammar while on the right there are the results for the grammar with the additional hybridization rule. Circles indicate algorithms generated not allowing any hybridization, with the triangles one level is allowed and with the squares two levels are allowed. With Rc^* we indicate the adjusted value of Rc after considering the explicit recursion as explained at the end of Section 3.2.

partial solutions in the perturbation.

4.2. Total completion times

The situation in PFSP$_{TCT}$ is more interesting with clearly different results between the two grammars. Looking at the correlation plot for the base grammar, shown in Figure 8 (a), we can see that the algorithms are clustered in three groups. The first group consist of A1 algorithms, such as Algorithm 2, differing mostly on the termination condition of the first improvement local search, the kind of *IG* perturbation and the acceptance criteria. The second group is composed of two A3 algorithms having a *VND* and an *IG* perturbation with performances comparable to the ones in the first group. The four algorithms in the final group, that are also the ones with the better performances, are A5 algorithms, which combines two SLS algorithms. In all four cases, one of the two SLS algorithms has a stronger perturbation combined with an accepting criteria biased towards intensification, that is accepting only or with an high probability improving solutions. The other of the two instead, has a weaker perturbation combined with an accepting criteria having an higher probability of accepting non improving solutions. MRSILS(BSCH) (Fernandez-Viagas and Framiñán, 2017) is shown at the top of Figure 8 (a) and shows

Algorithm 4 A3

Input Given a Problem Definition π,
Output The best solution found s^*,
s := init()
s^* := s
while ! termination criterion **do**
    s' := perturbation$(s, iterative\ improvement)$
    s'' := VND$(s', neighborhoods)$
    s := acceptance(s, s'')
    if $f(s) < f(s^*)$ **then**
        s^* := s'
    end if
end while
Return s^*

Algorithm 5 A4

Input Given a Problem Definition π,
Output The best solution found s^*,
s := init()
s^* := s
a := 0
while ! termination criterion **do**
    s' := perturbation$(s, iterative\ improvement)$
    ALS procedure
    if $a \mod 2 = 0$ **then**
        s'' := iterative improvement 1(s')
    else
        s'' := iterative improvement 2(s')
    end if
    s := acceptance(s, s'')
    if $f(s) < f(s^*)$ **then**
        s^* := s'
    end if
    a := $a + 1$
end while
Return s^*

a significant difference with the automatically generated algorithms. This algorithm belongs to the A1 group but differs from the other algorithms in this group for the perturbation and the acceptance criterion. In fact, MRSILS(BSCH) uses a perturbation based on random moves in the insert neighborhood instead of the *IG* perturbation and accepts only improving solution as acceptance criterion instead of an acceptance criterion based on the Metropolis condition. $ALG_{irtct}(BSCH)$ instead, sits at the bottom

of Figure 8 (a) and belongs to the A5 group. The use of the *BSCH* heuristic improved significantly the performance of the algorithm that went from having a mean of 0.59 (as reported in Pagnozzi and Stützle (2019)) to 0.15. Considering the algorithms generated in this work, there are three algorithms with a better mean that $ALG_{irtct}(BSCH)$ but, as in the case of PFSP_{MS}, the difference is not significant according to the Wilcoxon test.

The algorithms generated with the grammar with explicit recursion, shown in Figure 8 (b), can be grouped in almost the same way with a first group composed of A1 algorithms, one A3 algorithm, a third bigger group composed of A5 algorithms and, finally, two algorithms composed of three nested SLS algorithms similar to the A6 algorithm shown in Algorithm 7. The main difference with the base grammar is that, in this case, the majority of the generated algorithms is composed of hybridized algorithms. Moreover, irace was able to generate, on average, algorithms with better performance compared with the base grammar. This difference between the two grammars has been found to be statistically significant by the Wilcoxon paired test.

Considering Figure 8, it seems that passing a certain complexity threshold, that consists of having a two nested SLS structure, delivers better performances. This results is in accordance with ALG_{irtct} (Pagnozzi and Stützle, 2019), which is also a composed algorithm consisting of nested SLS algorithms. Although better performing, there is a low probability of generating hybridized algorithms when using the base grammar. This result could mean that common SLS algorithms represent a local minima in the configurations search space. On the contrary, the results with the more complex grammar, in Figure 8 (b), indicate that increasing the chances of generating more complex algorithms seems to help irace to overcome this local minima.

4.3. Total tardiness

While complexity seems to lead to better performances for PFSP_{TCT}, the same does not seem to be the case for PFSP_{TT} when considering the base grammar, as shown in Figure 9 (a) where all algorithms but two are versions of the A1 algorithm. One of the two is a nested algorithm similar to the A5 algorithm as in Algorithm 6. The other is similar to the A4 algorithm shown in Algorithm 5. This algorithm uses the *ALS* local search, where two iterative improvement algorithms are executed alternatively. The best manually designed algorithm, IAras (Fernandez-Viagas and Framiñán, 2018) has by far the worst performances and it is shown at the top of Figure 9 (a). As in the case of MRSILS(BSCH) for PFSP_{TCT}, IAras belongs to the A1 group. This algorithm is a simple ILS with a perturbation based on random moves in the transpose neighborhood and an acceptance criterion based on the Metropolis condition with a fixed temperature. Considering that almost all the algorithms in the A1 group share this same structure, the difference in performance shown by IAras is probably caused by the configuration of the numerical parameters. $ALG_{irtt}(BS)$, shown in the middle in Figure 9 (a), belongs to the A5 group, being composed of two nested SLS algorithms. Considering the performance, $ALG_{irtt}(BS)$ shows a significant improvement thanks to the *BS* heuristic, passing from an RDI of 0.032[1] to 0.025. When comparing with the algorithms generated for this work, there are nine algorithms, three in the A1 group and three in the

[1]This value is calculated using the data reported in Pagnozzi and Stützle (2019) and taking into account the best and worst solutions used in this study.

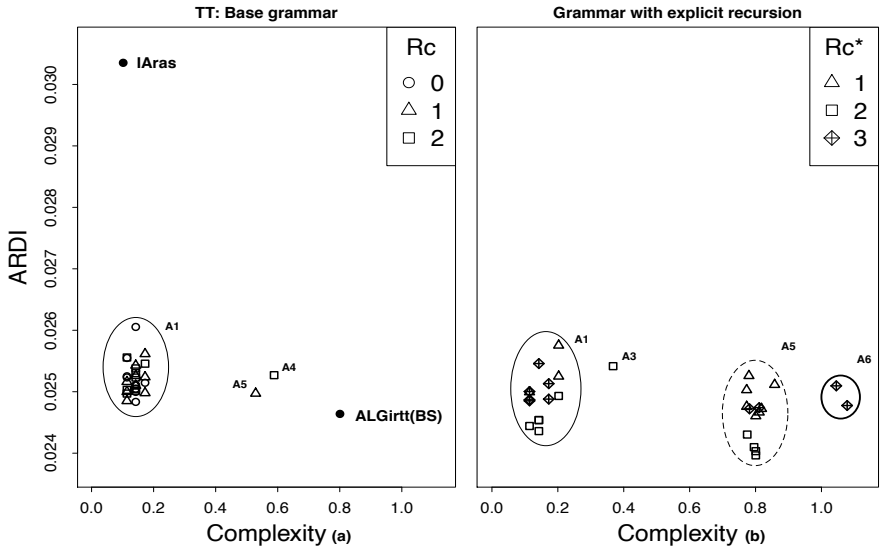

Fig. 9: Correlation between ARDI and algorithm complexity for the PFSP$_{TT}$ considering three levels of maximally allowed recursion. On the left, there are the results for the base grammar while on the right there are the results for the grammar with the additional hybridization rule. Circles indicate algorithms generated not allowing any hybridization, with the triangles one level is allowed and with the squares two levels are allowed. With Rc^* we indicate the adjusted value of Rc after considering the explicit recursion as explained at the end of Section 3.2.

A5 group, that have a better mean than $ALG_{irtt}(BS)$. This result is statistically significant for seven of these algorithms. A probable explanation for this result is that $ALG_{irtt}(BS)$ does not have the best balance between intensification and diversification to take fully advantage of the initial solution generated by *BS*.

In the case of the grammar with explicit hybridization, the results, presented in Figure 9 (b) show a different outcome. In fact, the algorithms can be grouped in four types. The first group of algorithms is composed of A1 algorithms with random move perturbations. There is an A3 algorithm, a group composed of hybridized algorithms such as the A5 algorithm and, finally, two algorithms composed of three SLS algorithms such as the A6 algorithm. Interestingly, the grammar with the additional hybridization led to better performing algorithms with and without hybridization. This result is significant according to the Wilcoxon paired test. A possible explanation is that forcing irace to consider more hybridized algorithms may help it in the exploration of the parameter space. The results suggests that in this case the difference in performance of SLS and nested SLS algorithms is not very big. Considering the number of non nested algorithms, these results further confirms that the system tends to prefer simple algorithms with the base grammar. On the contrary, using the grammar with explicit hybridization can lead to better algorithms, both simple and nested.

Algorithm 6 A5

```
1: Input Given a Problem Definition π,
2: Output The best solution found s*,
3: s := init()
4: s* := s
5: while ! termination criterion do
6:     s' := perturbation(s)
7:     s'' := A1(s')
8:     s := acceptance(s, s'')
9:     if f(s) < f(s*) then
10:        s* := s'
11:    end if
12: end while
13: Return s*
```

Algorithm 7 A6

```
1: Input Given a Problem Definition π,
2: Output The best solution found s*,
3: s := init()
4: s* := s
5: while ! termination criterion do
6:     s' := perturbation(s)
7:     s'' := A5(s')
8:     s := acceptance(s, s'')
9:     if f(s) < f(s*) then
10:        s* := s'
11:    end if
12: end while
13: Return s*
```

5. Discussion and conclusions

In this paper we analyzed how grammars influence the generation of complex algorithms in the automatic algorithm design system. We generated algorithms with grammars allowing increasing levels of complexity for the three most studied PFSP objectives, PFSP$_{MS}$, PFSP$_{TCT}$ and PFSP$_{TT}$. For each objective and level of complexity we used a base grammar and a grammar with an additional rule to directly build hybridized algorithms. The complexity of the algorithms generated with these grammars was compared using a metric based on DAGs.

The experiments with PFSP$_{TCT}$ and PFSP$_{TT}$ have shown that an *AAD* system generates a more complex algorithm only if it performs similarly or better than a less complex algorithm. Furthermore, the results for PFSP$_{MS}$ show that the huge parameter spaces produced by allowing hybridization do not seem to generate less performing algorithms when simple algorithms perform better. In fact, the majority of the algorithms produced for this objective are quite clustered around a similar ARPD value. Additionally, our results show that using a grammar that favors hybridization not only leads to generating better performing algorithms when complex algorithms works best, as in PFSP$_{TCT}$ and PFSP$_{TT}$, but also it does not seem to add a significant bias towards complex algorithms when they do not perform as well as simple ones. These observations make us conclude that algorithm hybridization should always be allowed, at least for PFSP problems.

The generated algorithms were compared with the state-of-the-art as well as with the best manually designed algorithms. The results further confirms that *AAD* is one of the best methodology to generate high performing algorithms even in cases where constructing the best algorithm seems obvious, as in adding a better initial solution heuristic to the state-of-the-art algorithm ALG_{irtt}.

Finally, in this paper we have seen that automatically generated algorithms tends to have a simple structure for problems with a large literature, such as PFSP$_{MS}$, for which there are many papers proposing high-performing heuristics, neighborhoods, perturbations and, in general, problem-specific components. Considering this observation, a future direction for this line of research would be to understand whether

there is a link between the availability of high-performing problem-specific components and structural complexity of automatically generated algorithms.

Acknowledgments

Thomas Stützle acknowledges support from the Belgian F.R.S.-FNRS, of which he is a Research Director. The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 681872).

References

Alfaro-Fernández, P., Ruiz, R., Pagnozzi, F., Stützle, T., 2020. Automatic algorithm design for hybrid flowshop scheduling problems. *European Journal of Operational Research* 282, 835–845.

Bezerra, L.C.T., López-Ibáñez, M., Stützle, T., 2019. Automatically designing state-of-the-art multi- and many-objective evolutionary algorithms. *Evolutionary Computation*. doi: 10.1162/evco_a_00263.

Bezerra, L.C.T., López-Ibáñez, M., Stützle, T., 2020. Automatic configuration of multi-objective optimizers and multi-objective configuration. In Bartz-Beielstein, T., Filipič, B., Korošec, P. and Talbi, E.G. (eds), *High-Performance Simulation-Based Optimization*. Springer International Publishing, Cham, Switzerland, pp. 69–92.

Biedenkapp, A., Marben, J., Lindauer, M., Hutter, F., 2018. Cave: Configuration assessment, visualization and evaluation. In Battiti, R., Brunato, M., Kotsireas, I. and Pardalos, P.M. (eds), *Learning and Intelligent Optimization, 12th International Conference, LION 12*, Springer, Cham, Switzerland, pp. 115–130.

Brum, A., Ritt, M., 2018a. Automatic algorithm configuration for the permutation flow shop scheduling problem minimizing total completion time. In Liefooghe, A. and López-Ibáñez, M. (eds), *Proceedings of EvoCOP 2018 – 18th European Conference on Evolutionary Computation in Combinatorial Optimization*, Springer, Heidelberg, Germany, pp. 85–100.

Brum, A., Ritt, M., 2018b. Automatic design of heuristics for minimizing the makespan in permutation flow shops. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, IEEE Press, Piscataway, NJ, pp. 1–8.

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

Burke, E.K., Hyde, M.R., Kendall, G., Ochoa, G., Özcan, E., Woodward, J.R., 2019. A classification of hyper-heuristic approaches: Revisited. In Gendreau, M. and Potvin, J.Y. (eds), *Handbook of Metaheuristics*, *International Series in Operations Research & Management Science*. Vol. 272. Springer, chapter 14, pp. 453–477.

De Souza, M., Ritt, M., 2018a. Automatic grammar-based design of heuristic algorithms for unconstrained binary quadratic programming. In Liefooghe, A. and López-Ibáñez, M. (eds), *Proceedings of EvoCOP 2018 – 18th European Conference on Evolutionary Computation in Combinatorial Optimization*, Springer, Heidelberg, Germany, pp. 67–84.

De Souza, M., Ritt, M., 2018b. An automatically designed recombination heuristic for the test-assignment problem. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, IEEE Press, Piscataway, NJ, pp. 1–8.

Dubois-Lacoste, J., Pagnozzi, F., 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., 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., Framiñán, J.M., 2017. A beam-search-based constructive heuristic for the PFSP to minimise total flowtime. *Computers & Operations Research* 81, 167–177.

Fernandez-Viagas, V., 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.

Fernandez-Viagas, V., Valente, J.M.S., 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.

Franzin, A., Stützle, T., 2019. Revisiting simulated annealing: A component-based analysis. *Computers & Operations Research*

104, 191 – 206.

Grefenstette, J.J., 1986. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics* 16, 1, 122–128.

Hansen, N., Ostermeier, A., 2001. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation* 9, 2, 159–195.

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

Humeau, J., Liefooghe, A., Talbi, E.G., Verel, S., 2013. ParadisEO-MO: From fitness landscape analysis to efficient local search algorithms. *Journal of Heuristics* 19, 6, 881–915.

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

Hutter, F., Hoos, H.H., Stützle, T., 2007. Automatic algorithm configuration based on local search. In Holte, R.C. and Howe, A. (eds), *Proc. of the Twenty-Second Conference on Artifical Intelligence (AAAI '07)*, AAAI Press/MIT Press, Menlo Park, CA, pp. 1152–1157.

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., Leyton-Brown, K., 2016. SATenstein: Automatically building local search SAT Solvers from Components. *Artificial Intelligence* 232, 20–42.

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

Koza, J., 1992. *Genetic Programming: On the Programming of Computers By the Means of Natural Selection*. MIT Press, Cambridge, MA.

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

Liu, J., 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., Birattari, M., 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3, 43–58.

López-Ibáñez, M., Kessaci, M.E., Stützle, T., 2017. Automatic Design of Hybrid Metaheuristics from Algorithmic Components. Technical Report TR/IRIDIA/2017-012, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium.

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

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

Maron, O., Moore, A.W., 1997. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Research* 11, 1–5, 193–225.

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

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

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

Mockus, J., 1989. *Bayesian Approach to Global Optimization: Theory and Applications*. Kluwer Academic Publishers.

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

Nikolić, M., Marić, F., Janičić, P., 2009. Instance-based selection of policies for SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, Springer, pp. 326–340.

Pagnozzi, F., Stützle, T., 2019. Automatic design of hybrid stochastic local search algorithms for permutation flowshop prob-

lems. *European Journal of Operational Research* 276, 409–421.

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

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

Rajendran, C., 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.

Ruiz, R., 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.

Sabar, N.R., Ayob, M., Kendall, G., Qu, R., 2015. Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems. *IEEE Transactions on Evolutionary Computation* 19, 3, 309–325.

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

Vallada, E., Ruiz, R., 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, Y., Dong, X., Chen, P., Lin, Y., 2014. Iterated local search algorithms for the sequence-dependent setup times flow shop scheduling problem minimizing makespan. In *Foundations of Intelligent Systems*. Springer, pp. 329–338.

Xu, L., KhudaBukhsh, A.R., Hoos, H.H., Leyton-Brown, K., 2016. Quantifying the similarity of algorithm configurations. In *International Conference on Learning and Intelligent Optimization*, Springer, pp. 203–217.

Additional data

Results summary for PFSP$_{MS}$

Base Grammar			Grammar with explicit recursion					
	Complexity	ARPD		Complexity	ARPD		Complexity	ARPD
$Rc = 0$			$Rc^* = 1$					
A0	0.299	0.212	A0	0.357	0.208	IG_{all}	0.232	0.240
A1	0.290	0.213	A1	0.331	0.239	IG_{irms}	0.331	0.206
A2	0.313	0.218	A2	0.937	0.207			
A3	0.279	0.219	A3	0.357	0.211			
A4	0.258	0.224	A4	0.243	0.254			
A5	0.274	0.240	A5	0.211	0.269			
A6	0.331	0.215	A6	0.280	0.209			
A7	0.274	0.207	A7	0.292	0.220			
A8	0.306	0.206	A8	0.323	0.214			
A9	0.292	0.233	A9	0.331	0.209			
$Rc = 1$			$Rc^* = 2$					
A0	0.299	0.211	A0	0.295	0.210			
A1	0.258	0.221	A1	0.331	0.212			
A2	0.313	0.222	A2	0.365	0.209			
A3	0.282	0.208	A3	0.264	0.259			
A4	0.306	0.206	A4	0.931	0.208			
A5	0.274	0.207	A5	0.282	0.206			
A6	0.299	0.211	A6	0.260	0.255			
A7	0.313	0.207	A7	0.331	0.203			
A8	0.313	0.207	A8	0.282	0.208			
A9	0.366	0.220	A9	0.229	0.261			
$Rc = 2$			$Rc^* = 3$					
A0	0.299	0.216	A0	0.289	0.220			
A1	0.331	0.208	A1	0.584	0.227			
A2	0.319	0.210	A2	0.243	0.267			
A3	0.323	0.206	A3	0.299	0.205			
A4	0.339	0.210	A4	0.274	0.219			
A5	0.260	0.259	A5	0.292	0.206			
A6	0.299	0.204	A6	0.270	0.221			
A7	0.323	0.207	A7	0.274	0.207			
A8	0.260	0.259	A8	0.925	0.208			
A9	0.323	0.212	A9	0.302	0.212			

Table A1: The measured values of algorithm complexity and performance for the makespan objective. On the left and center we report the results of the algorithms generated during the experiments grouped by the type of grammar and the Rc or the Rc^* value. On the right we report the values of the best algorithms designed manually and automatically

Results summary for PFSP$_{TCT}$

Base Grammar			Grammar with explicit recursion					
	Complexity	ARPD		Complexity	ARPD		Complexity	ARPD
$Rc = 0$			$Rc^* = 1$					
A0	0.304	0.165	A0	0.532	0.165	MRSILS(BSCH)	0.192	0.215
A1	0.304	0.166	A1	0.791	0.158	$ALG_{irtct}(BSCH)$	0.821	0.147
A2	0.272	0.164	A2	0.727	0.152			
A3	0.279	0.168	A3	0.736	0.155			
A4	0.272	0.166	A4	0.821	0.148			
A5	0.267	0.165	A5	0.807	0.154			
A6	0.144	0.161	A6	0.800	0.155			
A7	0.275	0.165	A7	0.807	0.149			
A8	0.288	0.163	A8	0.774	0.155			
A9	0.504	0.165	A9	0.304	0.164			
$Rc = 1$			$Rc^* = 2$					
A0	0.222	0.166	A0	0.725	0.146			
A1	0.571	0.151	A1	1.036	0.154			
A2	0.674	0.154	A2	0.764	0.148			
A3	0.222	0.164	A3	0.695	0.149			
A4	0.250	0.173	A4	1.054	0.148			
A5	0.264	0.165	A5	0.802	0.154			
A6	0.296	0.169	A6	0.243	0.164			
A7	0.222	0.166	A7	0.871	0.148			
A8	0.295	0.166	A8	0.860	0.155			
A9	0.304	0.165	A9	0.793	0.153			
$Rc = 2$			$Rc^* = 3$					
A0	0.347	0.167	A0	0.819	0.148			
A1	0.635	0.155	A1	0.828	0.147			
A2	0.195	0.167	A2	0.251	0.167			
A3	0.304	0.165	A3	0.830	0.148			
A4	0.235	0.166	A4	0.142	0.168			
A5	0.275	0.163	A5	0.725	0.146			
A6	0.267	0.164	A6	0.264	0.165			
A7	0.513	0.164	A7	0.736	0.153			
A8	0.296	0.165	A8	0.791	0.150			
A9	0.585	0.155	A9	0.548	0.151			

Table A2: Summary of the results reporting algorithm complexity and performance for the total completion time objective. On the left and center we report the results of the algorithms generated during the experiments grouped by the type of grammar and the Rc or the Rc^* value. On the right we report the values of the best algorithms designed manually and automatically.

Results summary for PFSP$_{TT}$

Base Grammar			Grammar with explicit recursion					
	Complexity	ARPD		Complexity	ARPD		Complexity	ARPD
$Rc = 0$			$Rc^* = 1$					
A0	0.773	0.0207	A0	0.142	0.0207	IAras	0.102	0.0304
A1	0.202	0.0205	A1	0.142	0.0209	$ALG_{irtt}(BS)$	0.801	0.0204
A2	0.202	0.0209	A2	0.114	0.0214			
A3	0.811	0.0209	A3	0.114	0.0203			
A4	0.858	0.0208	A4	0.172	0.0208			
A5	0.780	0.0217	A5	0.142	0.0209			
A6	0.773	0.0207	A6	0.142	0.0204			
A7	0.801	0.0212	A7	0.114	0.0203			
A8	0.818	0.0208	A8	0.142	0.0204			
A9	0.114	0.0208	A9	0.142	0.0207			
$Rc = 1$			$Rc^* = 2$					
A0	0.368	0.0208	A0	0.142	0.0211			
A1	0.795	0.0207	A1	0.172	0.0198			
A2	0.142	0.0205	A2	0.114	0.0202			
A3	0.801	0.0206	A3	0.529	0.0196			
A4	0.202	0.0213	A4	0.172	0.0206			
A5	0.114	0.0208	A5	0.114	0.0201			
A6	0.801	0.0211	A6	0.142	0.0197			
A7	0.142	0.0207	A7	0.114	0.0202			
A8	0.142	0.0209	A8	0.142	0.0200			
A9	0.775	0.0209	A9	0.172	0.0200			
$Rc = 2$			$Rc^* = 3$					
A0	0.811	0.0211	A0	0.142	0.0204			
A1	0.114	0.0210	A1	0.142	0.0207			
A2	0.172	0.0211	A2	0.172	0.0206			
A3	0.114	0.0207	A3	0.142	0.0205			
A4	1.045	0.0209	A4	0.142	0.0208			
A5	0.172	0.0212	A5	0.114	0.0208			
A6	0.142	0.0206	A6	0.114	0.0211			
A7	0.782	0.0207	A7	0.114	0.0204			
A8	1.078	0.0209	A8	0.588	0.0204			
A9	0.114	0.0207	A9	0.142	0.0205			

Table A3: Summary of the results reporting algorithm complexity and performance for the total tardiness objective. On the left and center we report the results of the algorithms generated during the experiments grouped by the type of grammar and the Rc or the Rc^* value. On the right we report the values of the best algorithms designed manually and automatically