Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

# Grammar-based Generation of Stochastic Local Search Heuristics Through Automatic Algorithm Configuration Tools

Franco Mascia*, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle

*IRIDIA, CoDE, Université libre de Bruxelles (ULB), 1050 Brussels, Belgium*

**Abstract**

Several grammar-based genetic programming algorithms have been proposed in the literature to automatically generate heuristics for hard optimization problems. These approaches specify the algorithmic building blocks and the way in which they can be combined in a grammar; the best heuristic for the problem being tackled is found by an evolutionary algorithm that searches in the algorithm design space defined by the grammar.

In this work, we propose a novel representation of the grammar by a sequence of categorical, integer, and real-valued parameters. We then use a tool for automatic algorithm configuration to search for the best algorithm for the problem at hand. Our experimental evaluation on the one-dimensional bin packing problem and the permutation flowshop problem with weighted tardiness objective shows that the proposed approach produces better algorithms than grammatical evolution, a well-established variant of grammar-based genetic programming. The reasons behind such improvement lie both in the representation proposed, as well as in the method used to search the algorithm design space.

*Keywords:* Heuristics, grammatical evolution, automatic algorithm configuration, bin packing, flowshop scheduling

## 1. Introduction

Recent advances in the development of methods for the automatic configuration of optimization algorithms (also known as offline configuration) have shown the benefits of configuring optimization algorithms to specific problems [1–9], and several works in the literature use such methods to generate new optimization algorithms [10, 11]. We call these latter approaches *top-down approaches* for automatic algorithm design, since they use a parametrized algorithmic framework to produce specific algorithms. Such frameworks are normally designed starting from a general (and usually complex) procedure and integrating alternative high-level algorithm components as fully-functioning blocks. In such a top-down approach, the parameter space is easily defined according to these alternative components. However, the flexibility of the framework is determined by the complexity of the general procedure.

A different, *bottom-up* approach for automatic algorithm design is used by grammar-based genetic programming [12–17]. In this approach, the algorithm design space is described by a set of production rules, and valid algorithms are instantiated by repeated applications of these rules. The benefit of a bottom-up approach is an increased flexibility when defining valid combinations of algorithmic components. Moreover, thanks to this flexibility, algorithmic components in bottom-up approaches are often more fine-grained than in top-down approaches. So far, the search for the best instantiation of the grammar has been done by genetic programming and other evolutionary methods.

In this paper, we investigate whether automatic algorithm configuration methods can be applied in a bottom-up approach. We answer this question in two steps. First, we replace the evolutionary algorithm in grammatical evolution (GE) [18], a type of grammar-based genetic programming, by an automatic configuration method, irace [6], using the same representation of the grammar as in the evolutionary algorithm. Second, we propose a method to generate a parameter space from a grammar, such that instantiations of the grammar can be represented by parameter configurations, which are more natural for automatic configuration methods. We compare these proposals with a pure GE method recently tested for the one-dimensional bin packing problem (1BPP) [17].

Experimental results show that irace using the parameter space generated by our method finds better algorithms than the GE method. We confirm these results by extending our analysis to the permutation flowshop with weighted tardiness problem (PFSP-WT).

This paper is structured as follows. First, we examine related works and we describe the GE method. Second, we explain our proposal for generating parameter spaces from a grammar.

*Corresponding author

*Email addresses:* fmascia@ulb.ac.be (Franco Mascia), manuel.lopez-ibanez@ulb.ac.be (Manuel López-Ibáñez), jeremie.dubois-lacoste@ulb.ac.be (Jérémie Dubois-Lacoste), stuetzle@ulb.ac.be (Thomas Stützle)

[1]This research has received funding from the COMEX project (Nr: P7156) within the Interuniversity Attraction Poles Programme of the Belgian Science Policy Office and the Meta-X project (Nr: AUWB-08/13-ULB2), funded by the Scientific Research Directorate of the French Community of Belgium, and the MIBISOC network, an Initial Training Network funded by the European Commission, grant PITN-GA-2009-238819. Franco Mascia, Manuel López-Ibáñez and Thomas Stützle acknowledge support of the F.R.S.-FNRS of which they are post-doctoral researchers and a senior research associate, respectively.

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

Third, we apply our proposal to the 1BPP, compare it to the GE method from [17] and test it on a new, more challenging case study on the PFSP-WT. Finally, we present our conclusions and discuss the new research directions opened by this paper.

## 2. Top-down vs. Bottom-up Approaches for Automatic Algorithm Design

### 2.1. Top-down Approaches

Automatic algorithm configuration methods were conceived for tuning the parameters of stochastic optimization algorithms given a set of training instances representative of the problem of interest. These methods allow algorithm designers to test many more parameter configurations than what is typically feasible when algorithms are tuned by hand in an ad-hoc manner. Moreover, automatic algorithm configuration methods avoid inherent biases of human designers when selecting which parameters to tune and which experiments to carry out.

When designing an optimization algorithm for a specific problem, algorithm designers are likely to implement alternative design choices (either new or coming from the literature) for further testing. It is only a small step to implement these design choices as parameters within an algorithm [19]. Such an algorithm quickly becomes an algorithm framework, with components that represent alternative design choices exposed as parameters of the framework. A particular instantiation of the parameters of such a framework leads to the selection of specific design choices and, hence, a specific algorithm. By applying automatic configuration to algorithm frameworks, it is therefore possible to automatically design algorithms for specific problems. We call this method a *top-down* approach, and we can find various examples in the literature.

KhudaBukhsh et al. [10] built a parametrized algorithmic framework for the satisfiability (SAT) problem from components of algorithms that had shown good results in previous editions of the SAT competition. By setting the parameters of the framework to specific values, they could instantiate various successful SAT solvers and generate new variants. They used ParamILS [3] to find the best variant to tackle specific types of SAT instances. The multi-objective ant colony optimization (MOACO) framework [11] follows a similar idea. Unique algorithmic components of various MOACO algorithms from the literature have been identified and incorporated into a common algorithmic framework, where alternative components may be selected by means of parameters. This framework was able to instantiate most of the MOACO algorithms from the literature and to generate hundreds of new algorithm designs. Using irace [6], it was possible to find a configuration of the framework that outperformed the MOACO algorithms from the literature for the bi-objective travelling salesman problem.

### 2.2. Bottom-up Approaches

A *bottom-up* approach combines algorithmic components, which can range from a single operator to fully-functioning procedures, to form valid expressions in a language, which can be either pseudo-code or a specific programming language. In contrast to top-down approaches, in a bottom-up approach there is no need for a general algorithm framework, where many higher-level alternative design choices co-exist. This provides a greater flexibility when defining the space of valid algorithms, but it complicates the representation of valid algorithms and the search for the best one.

In bottom-up approaches, the space of valid algorithms is often given as a context-free grammar, that is, a set of production rules that describe how terminal and non-terminal symbols can be combined to produce valid sentences in the language. Fig. 2 shows an example grammar expressed in Backus-Naur Form (BNF), where each production rule is of the form `<non-terminal> ::=` *`expression`*`.` Each rule describes how the non-terminal symbol on the left-hand side can be replaced by the expression on the right-hand side. Expressions are strings of terminal and/or non-terminal symbols. If there are alternative strings of symbols for the replacement of the non terminal on the left-hand side, the alternative strings are separated by the symbol "`|`".

How to search for the best algorithm in the design space defined by the grammar and how to represent the sequence of derivation rules that represent an algorithm is the object of different methodologies in genetic programming (GP) [15, 20]. In the context of generating stochastic optimization algorithms, Caseau et al. [12] design hybrid large neighborhood search algorithms for vehicle routing problems by using a genetic algorithm that applies crossover and mutation to a list of algebraic terms extracted from the grammar. Fukunaga [13, 14] uses a strongly-typed genetic programming algorithm to evolve Lisp-like S-expressions that represent local search heuristics for SAT. Three recent works [16, 17, 21] use *grammatical evolution* (GE) [18], which is a variant of GP that represents an instantiation of the grammar as a sequence of integers. Given its simplicity and its recent popularity for the automatic bottom-up design of algorithms, we explain this latter approach in more detail in the following section. In a hyper-heuristics context, the bottom-up approaches for generating algorithms are also referred to as "heuristic generation technologies"; for a review of these we refer to the recent survey paper [22].

### 2.3. Grammatical Evolution

In GE, a sentence (in our case, an algorithm) is instantiated from a grammar by a sequence of integers called *codons*. The codons encode the sequence of derivation rules applied to generate the sentence. Given a grammar and a specific sequence of integers, the actual sentence is obtained as follows. A special non-terminal symbol (e.g., `<start>`) is replaced first by the expression on the right-hand size of its corresponding derivation rule. Then, the left-most non-terminal in the current expression is replaced by applying its corresponding derivation rule. If there are alternative rules for the non-terminal, a codon from the sequence is consumed. The codon integer value, modulo the number of alternatives, determines which alternative rule is applied. This process continues until the derivation is complete, that is, there are no more non-terminals to be replaced. Once the derivation is complete, the unused codons are discarded. If

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

all codons are consumed before the derivation is complete, the process continues from the first codon, an operation known as "wrapping around" the sequence of codons. A maximum number of wraps limits the number of times that each rule can be applied, and, thus, the length of the sentence (the algorithm) encoded by the codons. If this limit is reached, the algorithm represented by this sequence of codons is considered invalid.

Although the method that searches for the best sentence in GE is usually an evolutionary algorithm (EA), other search methods could, in principle, use the same codon-based representation. A related question is whether the codon-based representation is indeed the most appropriate for generating algorithms. Automatic algorithm configuration methods are able to handle complex parametric spaces including categorical, numerical and conditional parameters, whereas the codon-representation is restricted to categorical alternatives (although the categories are represented by integers, there is no implicit order or distance among them). On the other hand, it is not completely obvious how to obtain a parametric space that represents a given grammar.

## 3. Methods

### *3.1. Automatic Configuration Methods as Search Methods in Grammatical Evolution*

GE uses an evolutionary algorithm (EA) to search in the space of codon values. A recent example of such EA [17] uses fitness-proportionate selection, one-point cross-over, point mutation, a duplication operator (which selects and inserts a copy of a random subsequence of codons at the penultimate position of the sequence), and a pruning operator (which removes unused codons from the sequence). These operators are applied iteratively to replace the 90% worst individuals from the population at each generation. The length of each sequence of codons is variable. Each individual encodes an algorithm, and the individual is evaluated by running it on a single training instance and assigning a fitness to the best solution found. All individuals are re-evaluated at each generation using a different initial solution in order to take into account the stochasticity of the algorithms so that good individuals show good performance in multiple independent runs. Moreover, evaluation runs within each generation use the same initial solution in order to reduce the variance.

In principle, we could replace the above EA with any other search method, for example, those developed for automatic algorithm configuration. Automatic algorithm configuration methods are designed to handle not only the stochasticity of the algorithm being tuned, but also the heterogeneity of multiple training instances. In fact, the objective of automatic algorithm configuration methods is not to find the highest-performing algorithm for the given training instances, but an algorithm that maximizes the performance over unseen instances of the same problem.

We test this idea by replacing the EA in GE with irace [5, 6]. irace is an automatic algorithm configuration method that alternates between *racing* configurations to discard the worst-performing ones and sampling new candidate configurations from a probabilistic model. Within each race, candidate configurations are run on one instance at a time and a statistical test is used to discard configurations whenever there is sufficient statistical evidence that they perform worse than the best one. The race stops when only a small number of configurations remains in the race or a budget of runs assigned to this race is consumed. The best configurations found are used to adjust a probabilistic model, from which new configurations are sampled. A new race starts using the best configurations from the previous race and the newly sampled ones. This procedure continues until a maximum budget of runs is consumed.

In order to replace the EA in GE with irace, we use parameters to represent the codons. In the original work [17], each codon is a consecutive group of 8 bits in the genome; hence we use categorical parameters that can take any value between 0 and $2^8 - 1$. The rationale for using categorical values to represent the codons is that the values of the codons are simply labels for the alternative rules in the grammar and they do not have any implicit order. In the experiments with irace we will use 30 parameters, which is the median of the range of $[10, 50]$ codons used in the original GE [17]. Sections 5 and 6 compare these two approaches on the 1BPP and on the PFSP-WT.

### *3.2. From Grammars to Parameters*

Although the codon-based representation may be used directly in an automatic configuration method, this representation is characterized by a high decoupling between the sequence of codons describing the rule application and the algorithm that is produced. The decoupling is due to several factors: 1) the modulo operations for mapping a codon to one of the alternative rules to be applied makes the mapping non injective, i.e., there are many sequences of codons that encode the same algorithm; 2) the wrapping of the sequence implies that the same codon is mapped to different rules for the derivation of the algorithm; 3) part of the sequence may not be needed to complete the derivation of the algorithm, which leads to different sequences mapping to the same algorithm; and 4) deriving the left-most non-terminal symbol in the expression implies that the value of a codon determines also the meaning of all subsequent codons in the sequence [15]. Moreover, an automatic configuration method using a codon-based representation loses one of the advantages of such methods, i.e., the ability to handle complex parametric spaces with both numerical and categorical parameters.

We propose next a method for mapping the derivations in a grammar to a parameter space. Our method first creates a parameter space representation for a given grammar. This parameter space representation can then be used to generate parameter configurations that correspond to specific instantiations of the grammar. To map the derivations in a grammar into a list of parameters, we go through two phases. In the first one, the grammar, which is usually designed to be clear for the human designer, is preprocessed to simplify some derivations. In the second phase we generate the actual parameters from the simplified grammar.

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

```
1: x := randomized_first_fit()
2: for i = 1 to 100 iterations do
3:     x* := ig_step(x)
4:     if fitness(x*) < fitness(x) then
5:         x := x*
6:     end if
7: end for
8: return x
```

**Fig. 1.** Algorithmic scheme of the IG for the 1BPP.

```
1:       <start> ::= <select_bins> remove_items_from_bins() <repack>
2: <select_bins> ::= <type> | <type> <select_bins>
3:        <type> ::= highest_filled(<num>, <ignore>, <remove>)
4:                 | lowest_filled(<num>, <ignore>, <remove>)
5:                 | random_bins(<num>, <ignore>, <remove>)
6:                 | gap_lessthan(<num>, <threshold>, <ignore>, <remove>)
7:                 | num_of_items(<num>, <numitems>, <ignore>, <remove>)
8:         <num> ::= 2 | 5 | 10 | 20 | 50
9:   <threshold> ::= average | minimum | maximum
10:   <numitems> ::= 1 | 2 | 3 | 4 | 5 | 6
11:     <ignore> ::= 0.995 | 0.997 | 0.999 | 1.0 | 1.1
12:     <remove> ::= ALL | ONE
13:     <repack> ::= best-fit-decreasing | worst-fit-decreasing
14:                | first-fit-decreasing
```

**Fig. 2.** Grammar for generating ig_step in Fig. 1 for the 1BPP [17].

*3.2.1. Preprocessing*

The preprocessing phase consists of the following steps: a) rules that contain the same non-terminal symbol on both sides of the rule (*recursive rules*) are simplified to reduce the number of generated parameters; b) rules that contain no actual alternatives are removed by replacing the left-side non-terminal symbol by the right-side expression in every other rule; and c) rules that are defined but are not reachable through a derivation from the `<start>` non-terminal are removed. After the preprocessing phase, the grammar consists of non-terminal symbols that expand to alternatives of terminal and non-terminal symbols. From such a preprocessed grammar, the parameters are generated in the subsequent translation phase. The preprocessing is useful as it often reduces the number of parameters required.

*3.2.2. Translation to Parameters*

In the second phase, we traverse the tree of derivations with a depth first search. Each node is identified as one of the following types: terminals, rules that contain alternative choices, rules that contain numerical ranges, and recursive rules. From each rule we will produce one or more parameters.

The grammar in Fig. 2 describes how to build different heuristics for an IG algorithm (see algorithm in Fig. 1) for the 1BPP. The grammar will be explained in detail in Section 5. In the following, we will use this grammar as an example to explain how the different rules are mapped to parameters.

*Alternative choices.* Rules with alternative choices are represented as categorical parameters. This is specially natural in the case of rules that consist only of alternative terminals, such as (see line 9 of Fig. 2):

```
<threshold> ::= average | minimum | maximum
```

*Numerical ranges.* Numeric terminals, such as (see line 10 of Fig. 2):

```
<numitems> ::= 1 | 2 | 3 | 4 | 5 | 6
```

can be represented as categorical parameters or more naturally by numerical parameters with a defined range.

*Recursive rules.* The only difficulty appears when a rule can be applied more than once. In such a case, each application of the rule requires its own parameter. While theoretically infinitely many applications of the rule may seem possible, in any instantiation of the grammar, the number of repetitions must be finite. GE encodes this limit in the number of wrappings. In our method we use an explicit limit on the number of repetitions per rule. This limits the number of parameters required to describe such rules and also the length of the generated algorithm. In fact, when generating algorithms from grammars, such rules are rarely applied more than a small number of times, thus, the limit can also be small.

Since generating parameters for recursive rules is the non-trivial case, we explain it in more detail in the following. First of all, after the preprocessing phase, only rules with the same non-terminal symbol on the left-hand side and on the right-hand side are recognized as recursive rules. (More complex grammars with cycles that span across multiple derivation rules are currently not handled and we leave such extension for future work). To map the recursive rule to parameters allowing for at most $n$ applications of the rule, we loop on the rule $n-1$ times, and at the $n$-th visit, we restrict the values that can be assumed by the parameters to avoid further applications. Suppose we want to limit to $n = 3$ applications the following rules:

```
<select_bins> ::= <type> | <type> <select_bins>
       <type> ::= highest_filled(...)
                | lowest_filled(...)
```

for better clarity we will rename the terminal and non terminal symbols as in the grammar below:

```
<A> ::= <B> | <B> <A>
<B> ::= c | d
```

After the first visit to the recursive rule, we generate from the first derivation, a categorical parameter `A1` that can assume the values `B` and `BA`. To translate the second rule (`<B> ::= c | d`), two categorical parameters `B1` and `B2` are generated with domain `c` and `d`. The parameter `B1` will be taken in consideration only if `A1` assumes the value `B`, whereas `B2` will be taken in consideration only if `A1` assumes the value `BA`. Automatic configuration methods, such as irace, often handle such conditional parameters, that is, parameters only enabled for certain values of other parameters, in order to reduce the size of the parameter space.

After the second visit to the recursive rule, three additional parameters are generated. A categorical parameter `A2` will encode the second level of recursion; it will be considered only if `A1` assumes the value `BA`, and it will have the same domain as `A1`. Two further categorical parameters `B3` and `B4` will encode

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

the choice between `c` and `d` for the two possible values in the domain of `A2`.

Each further level of recursion will produce three further parameters. Such number can be reduced by changing the current grammar into an equivalent one:

```
<A> ::= <B> <T>
<T> ::= "" | <B> <T>
<B> ::= c | d
```

where `""` is the empty string that allows to stop the recursion. For such grammar, three visits to the recursive rule will produce five parameters instead of nine. The reduction becomes more important with more complex grammars and more levels of recursion. The five parameters generated are the following. At the first visit of the recursive rule the categorical parameters `B1` and `T1` are generated. The first can assume the values `c` and `d`, whereas the second one has for domain `{"", BT}`. At the second visit of the recursive rule, two analogous parameters `B2` and `T2` will be generated, and they will be taken in consideration only if `T1` assumes a value different from `""`. At the third and last visit of the recursive rule only a parameter `B3` will be generated, which will be taken in consideration only if `T2` assumes a value different from `""`.

The example discussed above is actually a particular case of recursion that expresses the concept that a valid program contains a list of *at least one* `<B>`. In the general case, all rules in the form `<rule> ::= x | x <rule> | x ...`, where *x* is a terminal or non-terminal symbol that appears alone and in each alternative, could be simplified to reduce the number of parameters generated.

Mapping the derivations in the grammar to parameters is implemented by grammar2code, a tool that takes as input a grammar and outputs the list of parameters as required by automatic algorithm configuration methods. In our previous work [23], this translation was done by hand. grammar2code is also able to instantiate the source code of the target algorithm given a specific parameter configuration and a grammar. This means that once the algorithm designer defines the grammar and a list of tuning instances, the process of designing alternative implementations, compiling them, testing them, and evaluating their performances to choose the best algorithmic design is completely automated.

The grammar can describe algorithms in pseudo-code or directly in a programming language ready to be compiled or interpreted. In this paper, the grammars are presented in pseudo-code for better clarity, whereas in the actual experiments they generate directly algorithms in Python (for the 1BPP) and C++ (for the PFSP-WT) without the need for further translations.

## 4. Experimental Evaluation

In the experimental evaluation, we compare three methods to generate algorithms from a grammar:

- evolutionary corresponds to the GE approach used by Burke et al. [17], that is, the grammar is linearized into a sequence of codons represented as 8-bit integers, and the search is performed by an EA. The EA has a population size of 50 and the number of generations is set to 50 for a total of 2500 function evaluations (algorithm runs). At each generation, 90% of the population is replaced by new offspring as follows. First, two individuals generate two offspring individuals by one point crossover with probability 0.9; otherwise, the two individuals are cloned as the new offspring. Second, the offspring individuals undergo mutation, duplication, and pruning of the codons with a probability of 0.001 for each operation. Each individual is evaluated on a single training instance. The EA in the published paper [17] was not completely specified; in particular, it is not clear how to handle invalid sequences of codons. In our implementation, if a genetic operator generates an individual with an invalid codon-sequence, we discard the individual and reapply the operator. Moreover, the original proposal says that new individuals are generated in pairs. It also says that 90% of a population of 50 are replaced at each iteration, that is, 45 new individuals need to be generated per iteration. In our implementation, we generate 23 pairs of individuals and discard the last individual generated without evaluating it.

- irace-ge: We adopt the linearization used in GE, and we map the codons in the genotype to categorical parameters that can assume $2^8$ different values. Then we use irace to search an effective algorithm for the instances in the training set. The tuning budget, corresponding to the function evaluations in evolutionary is set to 2500 algorithm runs. Since irace-ge uses several instances in the training set, and each candidate algorithm is tested on more than one training instance, the number of actual algorithms generated and tested is around five times lower than in evolutionary.

- irace-param: We map the derivation rules in the grammar into a sequence of categorical, real-valued or integer parameters. The mapping takes into consideration a limited number of applications of the recursive rules. In the remainder of this work, the maximum number of recursive rule applications is denoted in subscript after the method's name. Large values give more flexibility to the automatic configuration tool to find the best heuristics, however, they also enlarge the design space of potential heuristics. We then use irace to search an effective algorithm for the instances in the training set. Also in this case the tuning budget is set to 2500 and the total number of algorithms generated is around five times lower than in the case of evolutionary.

We compare the above three methods on two combinatorial optimization problems: 1BPP and PFSP-WT.

## 5. One-Dimensional Bin Packing

Burke et al. [17] propose to use GE to evolve an iterated greedy (IG) algorithm for the 1BPP, which corresponds to the

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

method we call evolutionary. As a first case study, we compare the two proposed methods, irace-ge and irace-param, with evolutionary following the experimental setup proposed in that work.

### 5.1. Problem Definition

The goal in 1BPP is to pack a set of items of different sizes into as few bins (of equal capacity) as possible, while respecting the capacity constraint of the bins. Despite the simplicity of its formulation, the 1BPP is $\mathcal{NP}$-hard [24], and a large number of approximation and heuristic algorithms have been proposed over the years. Among the best-performing heuristics is best-fit-decreasing [25], which assigns each item to the fullest bin among those with enough empty space.

### 5.2. Iterated Greedy: Grammar and Components

IG algorithms, start from a feasible solution and iterate over two phases: 1) a *destruction* phase, which removes a number of items from certain bins, and 2) a *reconstruction* phase, which greedily re-assigns the removed items to bins obtaining a new feasible solution.

The scheme of the IG algorithm for 1BPP is given in Fig. 1. The initial solution is constructed greedily with a randomized first-fit heuristic. After a random shuffle of the items to be assigned, for each item, randomized_first_fit evaluates systematically all bins and assigns the item to the first bin that has enough free space to fit the item. If no such bin exists a new empty bin is added to the constructed solution. At each iteration, a local search heuristic (ig_step) is applied to search in the neighborhood of the current solution. The IG algorithm applies this step 100 times and returns the best solution found.

The local search heuristic (ig_step) is generated from the grammar shown in Fig. 2. Starting from the non-terminal `<start>` (line 1 of Fig. 2), there are two main parts: 1) a destruction procedure that selects a number of bins according to one or more criteria, and removes a number of items from the selected bins; and 2) a reconstruction procedure that repacks the removed items according to a construction heuristic. The destruction is described by the recursive rule `<select_bins>` (line 2 in Fig. 2), which generates a variable number of criteria for selecting bins. These criteria select either the highest-filled bins, the lowest-filled bins, a number of random bins, a number of bins having a gap smaller than a specified threshold, or a number of bins containing exactly a specified number of items (lines 3-9 in Fig. 2). More details can be found in the original publication [17].

### 5.3. Experimental Evaluation

We followed the same experimental protocol and used the same instances as [17] to verify the correctness of our implementation. We were able to reproduce the results in [17] on three out of four instance families: Triples, Uniform500 and Uniform1000. On the fourth one (the Scholl instance family) we obtained results that were better than the ones presented in the original paper. Hence, if our implementation differs from the original, the difference appears to be in favour of evolutionary. The results are detailed in this article's supplementary material [26].

**Table 1**
Mean distance from lower bound obtained by the heuristics generated by each configuration method for the 1BPP. The standard deviation is indicated in parentheses.

| Family | evolutionary | irace-ge | irace-param$_3$ |
|---|---|---|---|
| Scholl | 0.84 (0.16) | 0.77 (0.12) | 0.76 (0.08) |
| Uniform500 | 0.57 (0.35) | 0.39 (0.08) | 0.39 (0.07) |
| Uniform1000 | 0.87 (0.25) | 0.73 (0.11) | 0.70 (0.07) |
| Uniform2000 | 1.39 (0.42) | 1.12 (0.24) | 1.11 (0.21) |
| Uniform4000 | 2.89 (1.10) | 2.54 (0.42) | 2.50 (0.25) |

Although evolutionary only uses one training instance, irace is designed to use more than one. Therefore we generated additional instances, and adapted the experimental protocol to account for this. The generated instances belong to six different families. The four families used by Burke et al. [17] (Scholl, Triples, Uniform500, and Uniform1000), plus two additional families (Uniform2000 and Uniform4000) have been added to test the methods on slightly larger and more challenging instances. For each instance family, we generate a training set of 30 instances, which is used by the tuning methods, and a test set of ten instances, which serves to validate the performance of the generated heuristics. All instances are available in the supplementary material [26]. To deal with the stochasticity of the methods, we perform 30 independent runs of each method with different random seeds. Since evolutionary uses only a single instance, each independent run uses a different instance from the training set. The methods based on irace have all 30 training instances available, but the actual number and instances used in each run depends on the search behavior and the random seed.

For each instance family, we use the three methods to search, in the design space defined by the grammar, for an IG algorithm that solves efficiently the training instances. Then, we measure the performance of the IG on the test instances by computing the distance of the obtained result from the lower bound [27] and by averaging it over 10 runs on the 10 test instances. The results, averaged over 30 repetitions of the procedure described above, are summarized in Table 1. In this table, as well as in the rest of the paper, we omitted the results obtained on the Triples family. The instances of the Triples family are too easy and are solved equally well by any method, leading to floor effects. The results on all families are available in the supplementary material [26].

Figs. 3 and 4 show the results obtained on the Scholl and Uniform500 instance families respectively. The results are plotted in log-scale since for some instance families the outliers would have impacted on the overall readability of the boxplots. Detailed plots on the three methods compared and several others discussed further in this paper are presented in the supplementary material [26].

In Table 2, we compare the three methods by means of a Friedman test blocking on the test instances. On the Uni-

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

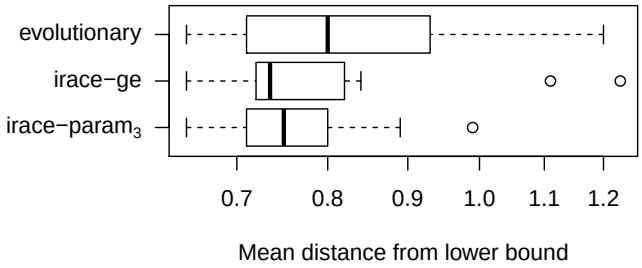


**Fig. 3.** Mean distance from lower bound obtained by the heuristics generated by each configuration method on the `Scholl` family of instances for 1BPP. Results are given across 30 independent runs of each configuration method.

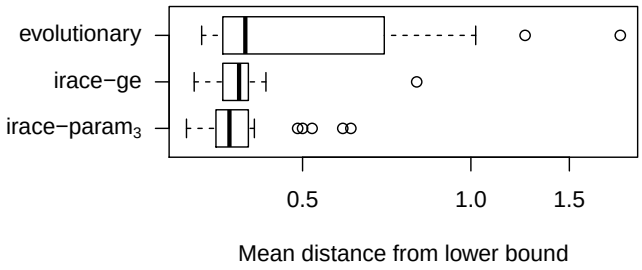


**Fig. 4.** Mean distance from lower bound obtained by the heuristics generated by each configuration method on the `Uniform500` family of instances for 1BPP. Results are given across 30 independent runs of each configuration method.

form500, Uniform2000 and Uniform4000 instances, the algorithms generated by irace using both the GE representation and the proposed parametric representation perform equally well and significantly better than evolutionary. On the Scholl and Uniform1000 instances, the results indicate that the combination of irace with the new parametric representation (irace-param) is significantly better than the other methods. Overall, irace-param is the best method across all instance families.

### 5.4. Analysis of the Results

In the following, we describe further experiments aimed at better understanding the results obtained in the previous section.

*Multiple instances for evolutionary.* The first question we want to address is why irace-ge is better than evolutionary. One reason could be that irace implements a better optimization method than the EA used in evolutionary; the other reason could be that irace is less sensitive to the risk of overtuning because it evaluates each candidate algorithm on more than one instance. To rule out this second hypothesis, we have implemented $\text{evolutionary}_{\text{mi}}$ as a variant of evolutionary that evaluates each candidate algorithm on multiple instances. By default in irace, candidate algorithms are evaluated on at least five instances. Thus, for each run of $\text{evolutionary}_{\text{mi}}$, we select five random instances from the 30 instances in the training set, and at each iteration we evaluate the individuals in the population with a different random seed on the five selected instances. To compensate for the fivefold increase in the number of function evaluations, we consider only 10 generations.

**Table 2**
Comparison of the methods through the Friedman test blocking on the instances of the five benchmark sets. $\Delta R_{\alpha=0.05}$ gives the minimum difference in the sum of ranks between two methods that is statistically significant.

| Family | $\Delta R_{\alpha=0.05}$ | Method ($\Delta R$) |
|---|---|---|
| Scholl | 4.65 | **irace-param$_3$** (0), irace-ge (7), evolutionary (17) |
| Uniform500 | 4.11 | **irace-param$_3$** (0), **irace-ge** (1), evolutionary (15.5) |
| Uniform1000 | 3.96 | **irace-param$_3$** (0), irace-ge (6), evolutionary (18) |
| Uniform2000 | 4.54 | **irace-param$_3$** (0), **irace-ge** (4), evolutionary (17) |
| Uniform4000 | 7.56 | **irace-param$_3$** (0), **irace-ge** (5.5), evolutionary (12.5) |

On all instance families except Uniform500, irace-ge obtains results that are better than $\text{evolutionary}_{\text{mi}}$. For example, on the Uniform4000 family, the mean distance from lower bound obtained with $\text{evolutionary}_{\text{mi}}$ is 3.57 with a standard deviation of 0.76, whereas with irace-ge we obtained a mean distance of 2.54 with a standard deviation of 0.42. The differences between the results obtained with the methods have been assessed with a Wilcoxon rank-sum test at significance level $\alpha = 0.05$, and for all families, except Uniform500, the difference was statistically significant. The results suggest that the difference between irace-ge and evolutionary is rather due to the search strategy of irace and not due to the larger number of instances used in the training phase. The results on all instance families are detailed in the supplementary material [26].

*Recursion depth for irace-param.* With the parametric representation used in irace-param, the complexity of the generated algorithms depends on the limit employed on the number of applications of the recursive rules in the grammar. We compared $\text{irace-param}_3$ with $\text{irace-param}_1$ that allows just one call of a recursive rule in the grammar, and $\text{irace-param}_5$ that allows up to five calls. On the largest Uniform4000 instances, the worst results are obtained with one level of recursion: the mean distance from the bound obtained with $\text{irace-param}_1$ is 5.42 with a standard deviation of 1.11, whereas with $\text{irace-param}_3$ the mean distance obtained is 2.50 with a standard deviation of 0.25. With $\text{irace-param}_5$, the mean distance from the bound is 2.34 with a standard deviation of 0.35. This pattern is consistent across all instance families, where $\text{irace-param}_1$ always obtains the worst results, whereas $\text{irace-param}_3$ and $\text{irace-param}_5$ obtain better results. The differences between the results have been assessed with a pairwise Wilcoxon rank-sum test at significance level $\alpha = 0.05$ with a Bonferroni correction to deal with the multiple comparisons. On all instance families, the difference between $\text{irace-param}_1$ and the other two methods was always statistically significant, whereas we could never reject the null hypothesis of no difference in the results obtained with $\text{irace-param}_3$ and $\text{irace-param}_5$. This suggests that to be effective, the algorithms should have a level of complexity that can be obtained only by repeated applications of the recursive rules in the grammar. The results on all instance families are detailed in the supplementary material [26].

*Random baselines.* To confirm the effectiveness of the methods analysed in this paper, we tested three random variants as control experiments. For evolutionary, we defined random

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

as an algorithm that generates the same number of individuals as evolutionary using the same GE representation. Because the EA used in evolutionary keeps 5 elite individuals at each generation, the total number of individuals generated are $50 + 45 \cdot 49 = 2255$. In random, these 2255 individuals are generated independently at random, they are tested once on the training instance, and the best one is selected. Note that in evolutionary the elite candidates are re-evaluated with a different seed at each generation, therefore if there is a bias, it is in favour of evolutionary since it performs 245 function evaluations more than random. For what concerns irace-ge and irace-param, since the candidates are tested on a variable number of instances, it is not possible to determine a priori how many individuals will be generated. Therefore, we let rand-ge and rand-param$_5$ generate 500 random individuals to be tested on 5 randomly selected training instances for a total of 2500 function evaluations. rand-ge uses the same GE representation employed in both irace-ge and evolutionary, whereas rand-param$_5$ uses a parametric representation with five levels of recursion, and it is therefore a baseline for irace-param$_5$. Finally, random$_{\text{mi}}$ is the baseline for evolutionary$_{\text{mi}}$; it generates independently at random $50 + 45 \cdot 9 = 455$ individuals, it tests each of them on five different training instances, and selects the best one. We assess the statistical significance of the differences by means of a Wilcoxon rank-sum test at significance level $\alpha = 0.05$. On all instance families, irace-ge and irace-param$_5$ are significantly better than their random counterparts, with the only exception being the Uniform500 instance family, where the difference between irace-ge and rand-ge is not statistically significant. Surprisingly, in the case of the comparison between evolutionary and random, the difference in the results was statistically significant only for the Uniform1000 and Uniform2000 instance families. In the first case, the better results are obtained with evolutionary, whereas in the second case it is the random method that produces better results. Concerning evolutionary$_{\text{mi}}$ and random$_{\text{mi}}$, the results are not clear-cut: evolutionary$_{\text{mi}}$ gives worse results than its random counterpart on the Scholl and Uniform4000 instances; it gives better results on the Uniform500 and Uniform1000 families; there was no statistically significant difference on the Uniform2000 family. The results on all instance families are detailed in the supplementary material [26].

In the next section, we carry out the same analysis on a different grammar that generates IG algorithms for a scheduling problem.

## 6. Permutation Flowshop Scheduling

The permutation flowshop scheduling problem (PFSP) is one of the most widely studied scheduling problems, as it models a very common kind of production environment in industries. Many formulations can be conceived by adding additional constraints and/or focusing on particular objectives. Because of its relevance in practice, the PFSP has attracted a large amount of research since it was formally described decades ago [28]. Moreover, since the PFSP is $\mathcal{NP}$-hard [29], tackling real-world instances often requires the use of heuristic algorithms. For these reasons, the PFSP is an important benchmark problem for the design and comparison of heuristics and using automatic generation to design new algorithms can save a significant effort when tackling less-studied PFSP variants.

### 6.1. Problem Definition

The goal in the flowshop problem (FSP) is to schedule a set of $n$ jobs $(J_1, \ldots, J_n)$ on $m$ machines $(M_1, \ldots, M_m)$. The specificity of flowshop environments is that all jobs must be processed on the machines in the same order, i.e., all jobs have to be processed on machine $M_1$, then machine $M_2$, and so on until machine $M_m$. A common restriction in the FSP is to forbid job passing between machines, i.e., to restrict to solutions that are permutations of jobs. The resulting problem is the PFSP. In the PFSP, all processing times $p_{ij}$ for a job $J_i$ on a machine $M_j$ are fixed, known in advance, and non-negative. In what follows, $C_{ij}$ denotes the completion time of a job $i$ on machine $j$ and $C_i$ denotes the completion time of a job $i$ on the last machine.

In many practical situations, for instance when products are due to customers at a specific time, jobs have an associated *due date*, $d_i$, for a job $J_i$. Moreover some jobs can be more important than other, which can be expressed by a priority associated to them. The *tardiness* of a job $J_i$ is defined as $T_i = \max\{C_i - d_i, 0\}$ and the *total weighted tardiness* is given by $\sum_{i=1}^{n} w_i \cdot T_i$, where $w_i$ is the weight assigned to job $J_i$ to specify its relative priority.

The PFSP-WT, considers the minimization of the total weighted tardiness; it is $\mathcal{NP}$-hard in the strong sense even for a single machine [30]. Formally, the PFSP-WT consists of finding a job permutation $\pi$, where $\pi_i$ denotes the job in the $i$-th position, such that:

$$\begin{aligned} \min \quad & F(\pi) = \sum_{i=1}^{n} w_i \cdot T_i \\ \text{subject to} \quad & C_{\pi_0 j} = 0 \quad j \in \{1, \ldots, m\}, \\ & C_{\pi_i 0} = 0 \quad i \in \{1, \ldots, n\}, \\ & C_{\pi_i j} = \max\{C_{\pi_{i-1} j}, C_{\pi_i j-1}\} + p_{ij}, \\ & \qquad i \in \{1, \ldots, n\} \quad j \in \{1, \ldots, m\}. \\ & T_i = \max\{C_i - d_i, 0\} \quad i \in \{1, \ldots, n\}. \end{aligned} \tag{1}$$

### 6.2. Iterated Greedy: Grammar and Components

The PFSP has been the object of many studies over the past decades, and it is still attracting a significant amount of research nowadays. Recent studies [31, 32] have shown that the best algorithms to tackle many PFSP variants are based on IG.

The IG algorithm for PFSP-WT is similar to the one explained for the 1BPP (Sec. 5.2). The initial solution is given by a random permutation of the jobs and, instead of a fixed number of iterations, we use a computation time limit. The *destruction* phase removes jobs from the schedule, whereas the *reconstruction* phase reinserts the jobs removed by considering them one by one and inserting them in the best position with respect to some evaluation function computed on the current partial solution. Thus, the main design choices are: 1) which jobs should be removed from the current solution, 2) in which order should

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

they be reinserted, and 3) which function should be optimized when choosing their new positions. These design choices are described by the grammar shown in Fig. 5.

*Jobs selection.* In this grammar, the algorithm selects jobs according to one or more criteria. Each criterion is composed of four components: a heuristic value, the number of jobs to select, and a lower bound and an upper bound for the heuristic value. We consider the following static heuristic values, which are given by the problem instance for each job $J_i$:

- The weight $w_i$ that defines its priority;
- the due date $d_i$;
- the sum of processing times $\sum_{j=1}^{m} p_{ij}$.

Additionally, we compute the following heuristic values from the current solution:

- The position where the job is (the lower the earlier it is in the schedule);
- the tardiness of a job in the current solution;
- the time the job waits between machines without being processed, computed as $\sum_{j=1}^{m-1}(C_{\pi_i j} - C_{\pi_i j+1})$;
- the time during which machines are not processing anything because the job takes too long on the previous machine, computed as $\sum_{j=1}^{m-1}(C_{\pi_{i-1} j} - C_{\pi_i j})$, for $i \neq 0$.

Each criterion removes a percentage `<num>` of those jobs that have a heuristic value in the range specified by [`<low_range>`, `<high_range>` ]. The ranges are specified as a percentage of the actual range of the heuristic values (0% being the minimum value and 100% the maximum). For example, a rule such as `select_jobs(dueDate, 20, 0, 10)` means "select the first 20% of the jobs having a due date within [0%, 10%]" of the actual ones.

*Jobs ordering for reinsertion.* Selected jobs are removed from the current schedule and then reinserted back into a new position. The re-insertion procedure considers such jobs one by one and therefore their relative order is important. The order is decided by one or more criteria. Additional criteria are used to break ties if needed. Each order criterion is defined by a heuristic value and a comparator that determines whether the jobs are ranked in increasing or decreasing order of the heuristic values.

*Objective for reinserting the jobs.* Finally, jobs are reinserted back into the best position in the current schedule according to some function measured on the resulting (possibly partial) schedule. For the PFSP-WT, it seems natural to optimize primarily the weighted tardiness. However, it may happen that for several candidate positions of a job, the weighted tardiness of the resulting solution is the same. For example, if the current partial solution is short, it often happens that the weighted tardiness is zero for every position a job may be inserted. Hence, it may be interesting to break ties using additional criteria. In particular, we use the *sum of completion times* of the partial solution, a well-known function positively correlated with the weighted tardiness; and the *weighted earliness*, computed as $\sum_{i=1}^{n} w_i \cdot (d_i - C_i)$, which is negatively correlated with the weighted tardiness.

In state-of-the-art IGs for PFSP-WT, a local search step is applied to each complete solution after the reconstruction phase [32]. However, in the present paper we do not consider such a component inside our IG algorithm, as the focus is on comparing our approach of generating algorithms to grammatical evolution.

### 6.3. Experimental Evaluation

We generated a benchmark set of 40 PFSP-WT instances of 50 jobs and 20 machines, and 40 instances of 100 jobs and 20 machines. These two sizes are nowadays the most common ones in the literature to evaluate heuristic algorithms on various PFSP variants. Each set is split into 30 instances for the training set and 10 instances for the test set.

The instances were generated as follows. The processing times of the jobs on each machine are integers drawn from a uniform distribution, between 1 and 99 [33]. For the weight and due date associated to each job, the weights are integers generated uniformly at random between 1 and 10, and each due date $d_i$ is generated in a range proportional to the sum of processing times of the job $J_i$ as $d_i = \lfloor r \cdot \sum_{j=1}^{m} p_{ij} \rfloor$, where $r$ is a random real number sampled uniformly between 1 and 4 [34]. For the ease of future comparisons, we make these instances available in the supplementary material [26].

For each of the two families, we use the three methods already described for the 1BPP (evolutionary, irace-ge, and irace-param$_3$) to search the design space. In these experiments, we use a grammar equivalent to the one in Fig. 5 to directly generate C++ code. The code is compiled with GCC 4.4.6 with optimization level `-O3`. Experiments were run on a single core of an AMD Opteron 6272 CPU (2.1 GHz, 16 MB L2/L3 cache size) running under Cluster Rocks Linux version 6/CentOS 6.3, 64bits. We measure the effectiveness of the generated IG algorithms by running them for $0.001 \cdot n \cdot m$ seconds, and by computing the relative percentage deviation (RPD) from the best solutions obtained in our tests. The RPD is averaged over 10 runs on the 10 instances of the test set. The experiments are repeated 30 times using different random seeds, as described earlier for the 1BPP. The results obtained are summarized in Table 3, whereas the distributions are depicted as boxplots in Fig. 6 for the 50x20 family and in Fig. 7 for the 100x20 family. In Table 4, we report the results of the Friedman test. On both instance families, the results obtained by irace-param$_3$ are significantly better than those obtained with evolutionary; in the case of the larger 100x20 instances, irace-param$_3$ is also significantly better than irace-ge.

### 6.4. Analysis of the Results

We perform the same analysis here as done earlier for the 1BPP in Section 5.4.

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

```
 1:                 <start> ::= procedure ig_step()
 2:                                <select_jobs>
 3:                                remove_selected()
 4:                                sort_removed_jobs(<ordering_criteria>)
 5:                                insert_jobs(construction_criteria)
 6:           <select_jobs> ::= <job_criteria> | <job_criteria> <select_jobs>
 7:          <job_criteria> ::= select_job(<heuristic>, <num>, <low_range>, <high_range>)
 8:             <heuristic> ::= priority | position | sumProcessingTimes | dueDate
 9:                           | tardiness | waitingTime | idleTime
10:                  <num> ::= {0, 1, ..., 100}
11:            <low_range> ::= {0, 1, ..., 99}
12:           <high_range> ::= {0, 1, ..., 100}
13:            <comparator> ::= "<" | ">"
14:     <ordering_criteria> ::= order(<comparator>, <heuristic>)
15:                           | order_and_break_tie(<comparator>, <heuristic>, <order_criteria>)
16: <construction_criteria> ::= weightedTardiness
17:                           | weightedTardiness, sumCompletionTimes
18:                           | weightedTardiness, sumCompletionTimes, weightedEarlyness
19:                           | weightedTardiness, weightedEarlyness
20:                           | weightedTardiness, weightedEarlyness, sumCompletionTimes
```

**Fig. 5.** Grammar for generating ig_step in Fig. 1 for the PFSP-WT. The derivation rules in lines 10-12 use a compact notation for defining numerical ranges.

**Table 3**
Mean relative percentage deviation obtained by the heuristics generated by each tuning method for the PFSP-WT. The standard deviation is indicated in parentheses.

| Family | evolutionary | irace-ge | irace-param$_3$ |
|---|---|---|---|
| 50x20 | 25.47 (8.78) | 17.01 (4.63) | 16.62 (5.04) |
| 100x20 | 5.37 (1.66) | 3.88 (0.98) | 3.43 (0.53) |

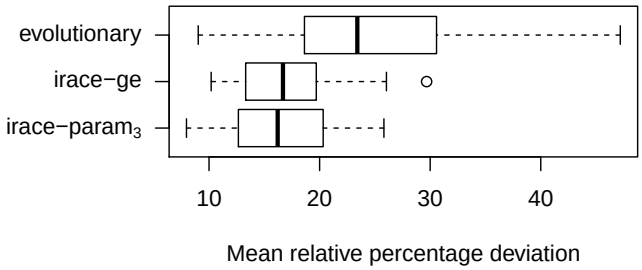


**Fig. 6.** Mean relative percentage deviation obtained by the heuristics generated by each tuning method on 50x20 family of instances for PFSP-WT. Results are given across 30 independent runs of each configuration method.

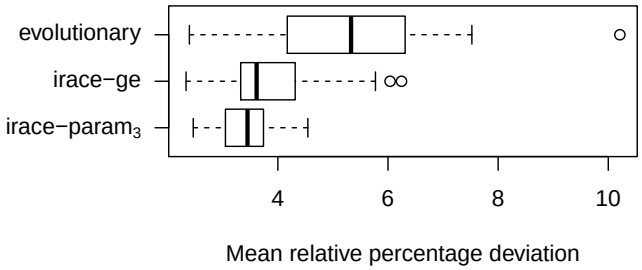


**Fig. 7.** Mean relative percentage deviation obtained by the heuristics generated by each tuning method on 100x20 family of instances for PFSP-WT. Results are given across 30 independent runs of each configuration method.

**Table 4**
Comparison of the methods through the Friedman test blocking on the instances of the two benchmark sets. $\Delta R_{\alpha=0.05}$ gives the minimum difference in the sum of ranks between two methods that is statistically significant.

| Family | $\Delta R_{\alpha=0.05}$ | Method ($\Delta R$) |
|---|---|---|
| 50x20 | 6.18 | **irace-ge** (0), **irace-param$_3$** (1), evolutionary (14) |
| 100x20 | 3.96 | **irace-param$_3$** (0), irace-ge (6), evolutionary (18) |

*Multiple instances for evolutionary.* We tested evolutionary$_{\text{mi}}$ on the 50x20 and 100x20 instance families and compared the results obtained with irace-ge. On the small 50x20 instances, evolutionary$_{\text{mi}}$ obtains a mean RPD 20.88 with standard deviation 3.53 whereas irace-ge obtains a mean RPD of 17.01 with standard deviation 4.63. The difference is statistically significant as assessed with a Wilcoxon rank-sum test at significance level $\alpha = 0.05$. On the larger 100x20 instances the difference in the performance between the two methods was not statistically significant. This suggests that the better performance obtained by irace-ge with respect to evolutionary is due to the optimization method employed by irace, and not due to the fact that irace-ge considers more than one training instance for evaluating the candidate algorithms analogous to what was observed in Section 5.4.

*Recursion depth for irace-param.* We compared the results obtained by irace-param$_1$, irace-param$_3$, and irace-param$_5$ by means of a pairwise Wilcoxon rank-sum test at significance level $\alpha = 0.05$ with a Bonferroni correction to deal with the multiple comparisons. The only difference that was statistically significant is on the 50x20 family between irace-param$_1$ and its counterparts with more levels of recursion, where irace-param$_1$ was significantly worse.

*Random baselines.* Comparing the methods with their random baselines we got results that are similar to the ones obtained on the 1BPP. In fact, as assessed with a Wilcoxon rank-sum test at significance level $\alpha = 0.05$, irace-param and irace-ge are always better than their random counterparts. On the contrary, the results obtained with evolutionary and evolutionary$_{\text{mi}}$ could never be distinguished from those obtained by their random counterparts.

## 7. Conclusion

In this paper, we have proposed to apply automatic configuration methods to generate optimization algorithms from a gram-

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

mar description. Our results show that simply replacing the EA in GE with irace and keeping the same linearization of the grammar is sufficient to improve over the results obtained with GE approaches [17]. We also propose a method for describing a grammar as a parametric space and instantiations of the grammar as parameter configurations. This parametric representation is much more natural for automatic configuration methods. Our results show that the variant of irace using the parametric representation obtains better results than the one using the GE representation.

The results above were tested on the 1BPP in order to compare with results from a previous study [17]. We significantly extended this previous study by considering more challenging instance sets and comparing with random baselines. The extended analysis shows that on some instance families, GE can be significantly worse than its random counterpart. We also confirmed our results on an additional, more challenging, combinatorial optimization problem, the PFSP-WT. We ruled out in our experiments the possibility that the advantage of irace over GE is due to the fact that irace uses more than one training instance, and is less sensitive to overtuning. The instance families used here are very homogeneous, and thus, using more than one instance for training does apparently not yield a significant advantage of irace in this case. Therefore, there is an advantage of irace over the EA used in GE, independently of the heterogeneity of the instances. This advantage becomes larger when using a parametric representation rather than the GE linearization probably because irace has been designed to deal with complex parametric spaces, while there is a high decoupling between the genotype (the codons) and the phenotype (the corresponding code produced) in the GE linearization.

The grammars explored in this paper (and previous studies [16, 17, 21]) are relatively simple and concerned with simple metaheuristics (IG in our case). There are grammars that cannot be handled by the method proposed here, whereas they could be handled by GE. Example of these are grammars where the derivation graph contains cycles that are not simple loops (that is, rules where the same non-terminal appears on both sides). Moreover, the number of parameters required to represent a particular grammar could be further reduced by more aggressive simplifications. We are working on extending our method to handle such cases. Nonetheless, the methods proposed here can already handle grammars that generate much more complex algorithms. For example, in a follow-up work, we have used the methods proposed here to generate hybrid metaheuristics from a composition of problem-independent and problem-dependent grammars [35].

## References


[1] M. Birattari, Tuning Metaheuristics: A Machine Learning Perspective, vol. 197 of *Studies in Computational Intelligence*, Springer, Berlin/Heidelberg, Germany, doi:10.1007/978-3-642-00483-4, 2009.

[2] V. Nannen, A. E. Eiben, A Method for Parameter Calibration and Relevance Estimation in Evolutionary Algorithms, in: M. Cattolico, et al. (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2006, ACM Press, New York, NY, 183–190, doi: 10.1145/1143997.1144029, 2006.

[3] 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.

[4] T. Bartz-Beielstein, C. Lasarczyk, M. Preuss, The Sequential Parameter Optimization Toolbox, in: T. Bartz-Beielstein, M. Chiarandini, L. Paquete, M. Preuss (Eds.), Experimental Methods for the Analysis of Optimization Algorithms, Springer, Berlin, Germany, 337–360, 2010.

[5] M. Birattari, Z. Yuan, P. Balaprakash, T. Stützle, F-Race and Iterated F-Race: An Overview, in: T. Bartz-Beielstein, M. Chiarandini, L. Paquete, M. Preuss (Eds.), Experimental Methods for the Analysis of Optimization Algorithms, Springer, Berlin, Germany, 311–336, 2010.

[6] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, M. Birattari, The irace package, Iterated Race for Automatic Algorithm Configuration, Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, URL `http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2011-004.pdf`, 2011.

[7] 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, 507–523, 2011.

[8] Z. Yuan, M. A. Montes de Oca, T. Stützle, M. Birattari, Continuous Optimization Algorithms for Tuning Real and Integer Algorithm Parameters of Swarm Intelligence Algorithms, Swarm Intelligence 6 (1) (2012) 49–75.

[9] E. Montero, M.-C. Riff, B. Neveu, A Beginner's Buide to Tuning Methods, Applied Soft Computing 17 (2014) 39–51, doi:10.1016/j.asoc.2013.12.017.

[10] 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, 517–524, 2009.

[11] 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, doi:10.1109/TEVC.2011.2182651.

[12] Y. Caseau, G. Silverstein, F. Laburthe, Learning Hybrid Algorithms for Vehicle Routing Problems, Theory and Practice of Logic Programming 1 (6) (2001) 779–806.

[13] A. S. Fukunaga, Evolving Local Search Heuristics for SAT Using Genetic Programming, in: K. Deb, et al. (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2004, Part II, vol. 3103 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, 483–494, 2004.

[14] A. S. Fukunaga, Automated Discovery of Local Search Heuristics for Satisfiability Testing, Evolutionary Computation 16 (1) (2008) 31–61, doi:10.1162/evco.2008.16.1.31.

[15] 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, doi:10.1007/s10710-010-9109-y.

[16] J. A. Vázquez-Rodríguez, G. Ochoa, On the automatic discovery of variants of the NEH procedure for flow shop scheduling using genetic programming, Journal of the Operational Research Society 62 (2) (2010) 381–396.

[17] E. K. Burke, M. R. Hyde, G. Kendall, Grammatical Evolution of Local Search Heuristics, IEEE Transactions on Evolutionary Computation 16 (7) (2012) 406–417, doi:10.1109/TEVC.2011.2160401.

[18] M. O'Neill, C. Ryan, Grammatical Evolution, IEEE Transactions on Evolutionary Computation 5 (4) (2001) 349–358.

[19] H. H. Hoos, Programming by optimization, Communications of the ACM 55 (2) (2012) 70–80, doi:10.1145/2076450.2076469.

[20] J. Koza, Genetic Programming: On the Programming of Computers By the Means of Natural Selection, MIT Press, 1992.

[21] J. Tavares, F. B. Pereira, Automatic Design of Ant Algorithms with Grammatical Evolution, in: A. Moraglio, S. Silva, K. Krawiec, P. Machado, C. Cotta (Eds.), Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012, vol. 7244 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, 206–217, 2012.

[22] E. K. Burke, M. Gendreau, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan,

Please cite this article as: Mascia, F., López-Ibáñez, M., Dubois-Lacoste J., & Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Computers & Operations Research 51:190-199, 2014. doi: 10.1016/j.cor.2014.05.020

R. Qu, Hyper-heuristics: A Survey of the State of the Art, Journal of the Operational Research Society 64 (12) (2013) 1695–1724.

[23] 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. Pardalos, G. Nicosia (Eds.), Learning and Intelligent Optimization, 7th International Conference, LION 7, vol. 7997 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, 321–334, doi:10.1007/978-3-642-44973-4_36, 2013.

[24] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman & Co, San Francisco, CA, 1979.

[25] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Chichester, UK, 1990.

[26] F. Mascia, M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools: Supplementary material, `http://iridia.ulb.ac.be/supp/IridiaSupp2013-009/`, 2013.

[27] S. Martello, P. Toth, Lower bounds and reduction procedures for the bin packing problem, Discrete Applied Mathematics 28 (1) (1990) 59–70, doi:10.1016/0166-218X(90)90094-S.

[28] D. S. Johnson, Optimal Two- and Three-stage Production Scheduling with Setup Times Included, Naval Research Logistics Quarterly 1 (1954) 61–68.

[29] M. R. Garey, D. S. Johnson, R. Sethi, The Complexity of Flowshop and Jobshop Scheduling, Mathematics of Operations Research 1 (1976) 117–129.

[30] J. Du, J. Y.-T. Leung, Minimizing Total Tardiness on One Machine is NP-Hard, Mathematics of Operations Research 15 (3) (1990) 483–495.

[31] 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.

[32] J. Dubois-Lacoste, M. López-Ibáñez, T. Stützle, A Hybrid TP+PLS Algorithm for Bi-objective Flow-Shop Scheduling Problems, Computers & Operations Research 38 (8) (2011) 1219–1236, doi:10.1016/j.cor.2010.10.008.

[33] É. D. Taillard, Benchmarks for Basic Scheduling Problems, European Journal of Operational Research 64 (2) (1993) 278–285.

[34] 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.

[35] 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, ISBN 978-3-642-38515-5, 144–158, doi:10.1007/978-3-642-38516-2_12, 2013.