Automatic Design of Hybrid Metaheuristics from Algorithmic Components

Manuel López-Ibáñez, Marie-Eléonore Kessaci, and Thomas Stützle

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2017-012

December 2017

IRIDIA – Technical Report Series
ISSN 1781-3794

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

Technical report number TR/IRIDIA/2017-012

Automatic Design of Hybrid Metaheuristics from Algorithmic Components

Manuel López-Ibáñez, Marie-Eléonore Kessaci, and Thomas Stützle, *Fellow, IEEE*

Abstract

Metaheuristic algorithms are traditionally designed following a manual, iterative algorithm development process. While this process sometimes leads to high performing algorithms, it is labor-intensive, error-prone, difficult to reproduce, and explores only a limited number of design alternatives. In this article, we advocate the automatic design of hybrid metaheuristic algorithms. For an effective design process, we propose as main ingredients a unified view on metaheuristic algorithms, an effective implementation of the metaheuristic components but also of problem-specific algorithm components inside a flexible algorithm framework, and the exploitation of automated algorithm configuration techniques. With these ingredients we show that, for various, rather different combinatorial optimization problems, we can automatically generate high-performance metaheuristic algorithms that reach and in various cases surpass the performance of current state-of-the-art algorithms for the respective problems. Our results also indicate that a paradigm shift in how effective metaheuristic algorithms are designed is possible, which has significant advantages such as reproducibility, unification of existing approaches, and reduction in the development time of high-performing algorithms.

I. Introduction

Metaheuristics are general algorithm templates used to design heuristic search algorithms by organizing diversifying and intensifying features of the search to escape the limitations of local optimality [1]. Over the last few decades, a number of metaheuristics, also called stochastic local search (SLS) methods [2], have been proposed. Some *population-based* metaheuristics, such as evolutionary algorithms [3] and ant colony optimization [4], maintain a set of candidate solutions, while many other well-known metaheuristics manipulate a single current solution at each iteration. Examples of the latter are simulated annealing [5, 6], tabu search [1], iterated local search [7], variable neighborhood search [8], or greedy randomized adaptive search procedures [9]. In this article, we will focus on such single-solution metaheuristics.

M. López-Ibáñez is with Alliance Manchester Business School, University of Manchester, UK. email: manuel.lopez-ibanez@manchester.ac.uk; M.-E. Kessaci is with Université de Lille, France. email: me.kessaci@univ-lille1.fr; T. Stützle is with IRIDIA, Université Libre de Bruxelles (ULB), Belgium. email: stuetzle@ulb.ac.be

Franco Mascia contributed to this work when he was a postdoctoral researcher at IRIDIA between 2012–2014.

When designing a metaheuristic algorithm, reasonable performance can often be obtained by rather straightforward adaptations of the general methods to the specific problem under concern. However, when higher performance is desired, the design of metaheuristic algorithms can require high expertise and a significant algorithm engineering effort that typically consists of a highly manual process guided by trial and error. Apart from being time-consuming, this manual process is highly biased by the expertise of the algorithm developer, often does not consider interdependencies between algorithm components, and leads to a limited exploration of algorithm design alternatives or relevant algorithm components.

As an alternative to this manual process, we show that the automated design of high-performance metaheuristic algorithms is well feasible. In particular, we show that metaheuristic algorithms for rather different combinatorial optimization problems can be automatically designed from a common algorithm framework and that the resulting algorithms are competitive or superior to current state-of-the-art algorithms. Our proposed automated algorithm design process works thanks to three main ingredients. The first is the identification of a unified algorithm template from which we can instantiate specific metaheuristics but also a large set of hybrid algorithms that combine components from various metaheuristics. This template also delivers a unified view on metaheuristics that are generally believed to have little in common. The second ingredient is the algorithm framework that has been created. It separates problem-specific from problem-independent aspects of the code, generating the high-level algorithm control as provided by the metaheuristic aspects in combination with problem-specific components that make the final algorithm efficient for the specific problem being tackled. The third ingredient are modern, automated algorithm configuration techniques, which nowadays can handle efficiently algorithm design tasks that involve hundreds of algorithm parameters [10–12].

Our goal is to automate the design of metaheuristic algorithms by developing a freely configurable framework, which spans a very wide set of metaheuristics and includes important, problem-specific components. In this sense, our work considerably extends over earlier efforts that also try to generate new algorithms out of algorithmic components. Previous work on automatic algorithm design often distinguishes between *top-down* approaches, where the design space is defined by some fixed algorithmic template with parametrized components and algorithms are generated by selecting the parameters of those components [13–15]; and *bottom-up* approaches, where a grammar specifies the rules for composing algorithmic components, thus giving much greater flexibility on the possible algorithm designs [16–18]. Either genetic programming [16, 19] or grammatical evolution [17, 18] are the most popular methods for generating algorithms from grammars. However, recent work has shown that automatic configuration techniques can also exploit flexible representations of algorithm compositions through grammars [20]. While *top-down* approaches are able to generate complete metaheuristics with state-of-the-art performance [13, 15]; *bottom-up* approaches are so far limited to either generating specific algorithmic components [16] or

```
GLS Algorithm
  1: $s_0 \leftarrow$ Initialization()
  2: $s^* \leftarrow$ GLS($s_0$)
  3: return $s^*$
```

```
Function GLS($s_0$)
Require: perturbation, localsearch, acceptance, stop
  1: $s^* \leftarrow$ localsearch($s_0$)
  2: repeat
  3:     $s' \leftarrow$ perturbation($s^*$)
  4:     $s'' \leftarrow$ localsearch($s'$)
  5:     $s^* \leftarrow$ acceptance($s''$, $s^*$)
  6: until termination criterion (stop) is satisfied
  7: return $s^*$
```

Figure 1. The generalized local search (GLS) algorithm. Hybrid GLS methods can be obtained by allowing localsearch to be a different (or similar) GLS.

metaheuristics that are far from the human-designed state-of-the-art methods for the same problem [17–20].

The grammar that underlies our algorithmic template has been carefully designed with the goal of allowing as much flexibility as possible while at the same time being able to generate recognizable metaheuristics, in particular, being able to replicate existing state-of-the-art methods. This is a key difference with previous algorithmic templates that either focused on providing a unified conceptual framework rather than generating high-performing metaheuristics [21]; or focused on replicating existing algorithms, while allowing less flexibility than our proposed design [22].

II. Component-based view of Hybrid Metaheuristics

The first ingredient of our automatic design system is a unified template from which complete algorithms may be instantiated. Our goal is to automatically design new hybrid metaheuristics, perhaps never explored before, and also to replicate state-of-the-art designs. Therefore, our algorithmic template must be flexible enough to instantiate a large number of different algorithms. At the same time, our template should allow *precisely* instantiating high-performing methods, such as the state-of-the-art methods already proposed in the literature.

As a first step, we studied well-known metaheuristics and their algorithmic components, trying to identify commonalities. The metaheuristics included in our study are iterated local search (ILS) [7], which creates a chain of solutions by iterating through the processes of solution perturbation, local search, and an acceptance criterion that decides from which solution to continue the iterative process; simulated annealing (SA) [6], which performs a random move within some neighbourhood and accepts the new solution according a probabilistic acceptance criterion (often the Metropolis condition) and a temperature parameter that is modified at run-time; probabilistic iterative improvement (PII) [2], which can be seen as a variant of simulated annealing with constant temperature; randomized iterative improvement (RII) [2], which applies a random walk step with a probability p or an iterative improvement step with probability $(1-p)$; variable neighborhood search (VNS) [8], which in many of its variants corresponds to an ILS that systematically adapts the perturbation size; iterated greedy (IG) [23], which iterates over constructive algorithms through

Table I

CLASSICAL METAHEURISTICS FORMULATED AS INSTANCES OF THE GLS TEMPLATE.

Name	**Perturbation**	**Local Search**	**Acceptance Criterion**
ILS [7]	*any*	*any*	*any*
SA [6]	one move	none	Metropolis
PII [2]	one move	none	Metropolis (fixed temp.)
RII [2]	one move	none	Probabilistic
VNS [8]	variable move	iterative improvement	Better
IG [23]	destruct-construct	*any*	*any*
TS [24]	none	tabu search	Always accept

a loop of solution destruction and construction, an optional local search and an acceptance criterion; and tabu search (TS) [24], which in its basic form keeps a list of prohibited solution components to avoid reversing recently made moves.

A first step towards a unified view of these rather varied methods is a common algorithmic template that is able to instantiate all of them. The algorithmic template we propose and call generalised local search (GLS) is shown in Fig. 1. At the top-level, a solution is initialised by the Initialization component. This solution is the starting solution of the GLS procedure, which takes this initial solution and applies intensification and diversification mechanisms until a termination criterion, such as maximum computation time, is met. The main loop of the GLS procedure is defined by three main components: perturbation, localsearch, and acceptance criterion. An additional stop criterion implements the termination criterion. The localsearch component is any procedure that takes an initial solution and returns a, hopefully better, solution. It is, therefore, an intensification mechanism aimed at improving its starting solution. The purpose of the perturbation component is to diversify the search by generating a solution that is different from its input. Finally, the acceptance component must decide whether to continue the search either from the result of the intensification, or from the diversification or from some other solution stored in memory.

GLS is heavily inspired by ILS, which can be obtained directly from the GLS procedure by simply setting the perturbation component to a randomly biased or uniform random modification of the current solution, localsearch to any improvement method and the acceptance criterion is any possible choice as discussed in [7]. Other well-known metaheuristics can be instantiated from the GLS template by setting its components to the proper procedures, as shown in Table I. For example, SA may be replicated by using a solution perturbation that performs one random move within a given neighborhood relation and uses the Metropolis acceptance criterion [25], with the temperature value used by the Metropolis criterion modified at run-time according to a cooling schedule. A PII algorithm then would be the same as SA but keep a constant temperature. A basic VNS is obtained by varying the perturbation strength using some schedule [8].[1]

[1]For example, it may initialize the perturbation strength k to a value k_{min} and then increase it by k_{step} until a given k_{max} if the execution of perturbation and localsearch did not lead to an improvement and reset it to k_{min} if it did.

The local search in simple cases of VNS may be a simple iterative improvement algorithm or a variable neighborhood descent in the generalized VNS algorithm, and the acceptance criterion only accepting better quality solutions as in basic VNS or accepting a new solution based on the solution quality and the distance to the current one as in skewed VNS [8]. An IG would be instantiated by making the perturbation consist of a destruction–construction cycle that first removes solution components from a complete candidate solution and, then, completes the resulting partial candidate solution using a constructive heuristic. This new, full candidate solution may be improved again by a localsearch. An acceptance criterion decides from which solution to continue as in ILS.[2] We could continue the description of the way the various metaheuristics are instantiated from GLS by enumeration; but from the few examples above, it should be clear that many variants of the above methods may be obtained by instantiating appropriately the four components Initialization, perturbation, localsearch, and acceptance. Additionally, some methods may offer specific components that have not been considered in the context of other methods but may still be useful to create additional variants.

High-performing metaheuristics not only combine ideas from different metaheuristics, but also hybridize metaheuristics such that one algorithm is executed as one step within another algorithm [26, 27]. Thus, our template should allow such hybridizations. While there may be several possibilities for how to build such hybrids, our framework creates hybrids by embedding one GLS as the localsearch component of another GLS, possibly instantiated with different components than in the upper levels. This results in an elegant way of hybridizing metaheuristics, analogous to executing an ILS algorithm inside another ILS algorithm as shortly discussed in [7]. An example of a hybrid with three levels (Fig. 2) would be an ILS method, whose local search is a VNS, whose local search is a SA. Each level has its own settings of the algorithmic components, including its own termination criterion. In practice, it is often more useful to define the termination criterion of inner levels as some fraction of the termination criterion of the outer level.

III. Automatic Generation of Hybrid Metaheuristics

A. A Grammar for Designing Hybrid Metaheuristics

The flexibility of hybridization afforded by the GLS template is more easily represented by a grammar. Figures 3, 4, and 5 show a subset of the derivation rules of a simplified version of the problem-independent grammar used in our experiments. The derivation of a GLS algorithm starts with the rule `<start>`, at Fig. 3. The start rule expands to an initialisation component and a `<GLS>` rule, which may derive into either one of the well-known metaheuristics discussed above or a hybrid method. In the latter case, the

[2]Various methods proposed in the literature essentially implement the same ideas as IG, among others, ruin-and-recreate and large neighborhood search (as explained in [23]). Thus, our proposed framework also includes these methods.

```
ILS(VNS(SA)):
  InitRandom();
  GLS(PertMoveInsert($k = 6$),                                          % ILS with perturbation strength 6
      GLS(PertMoveExchange(StrengthSchedule($k_{\min} = 1, k_{\max} = 10$)),                % VNS
          GLS(PertMoveSwap($k = 1$),                                    % Simulated Annealing
              LSNone(),
              AcceptMetropolis(T_init = $1$, T_factor = $1.5$, T_final = $10$, REHEAT),
              stop(max_time = $5\%$)),
          AcceptBetter,
          stop(max_time = $25\%$)),
      AcceptBetter,
      stop(maxTime = $60$ seconds))
  return best_found
```

Figure 2. Example of hybrid GLS.

```
      <start> ::= <Initialisation>; <GLS>; return best_found;
        <GLS> ::= <known_MH> | <Hybrid>
   <known_MH> ::= <ILS> | <SA> |<PII> | <RII> | <IG> | <VNS> | <TS>
     <Hybrid> ::= GLS( <perturbation>, <GLS>, <acceptance>, <stop> )
```

Figure 3. Grammar description of the generalised local search procedure.

```
<ILS>  ::=  GLS(<perturbation>, <local_search>, <acceptance>)
<PII>  ::=  GLS(<pert_one_move>, LSNone, AcceptMetropolis(<temperature>))
 <SA>  ::=  GLS(<pert_one_move>, LSNone, <accept_metropolis>)
<RII>  ::=  GLS(<pert_one_move>, LSNone, AcceptProbabilistic( <probability> ))
<VNS>  ::=  GLS(<pert_VNS>, <local_search>, <accept_VNS>)
 <IG>  ::=  GLS(<ps:pert_destruct_construct>, <local_search>, <acceptance>)
 <TS>  ::=  GLS(PertNone, <ls_TS>, AcceptAlways)
```

Figure 4. Known metaheuristic algorithms that can be generated from the grammar. Compare with Table I

```
         <accept_VNS>  ::=  AcceptBetter | <accept_skewed>
           <pert_VNS>  ::=  <ps:pert_repeatable> (<pert_strength_dyn_incr>)
       <perturbation>  ::=  PertNone | PertRestart | <pert_one_move>
                         |  <pert_k_moves> | <pert_variable>
       <pert_k_moves>  ::=  <ps:pert_repeatable> (<pert_strength_value>)
      <pert_variable>  ::=  <ps:pert_repeatable> (<pert_strength_schedule>)
       <local_search>  ::=  LSNone | <ls_first_imp> | <ls_best_imp > | <...>
         <acceptance>  ::=  AcceptAlways | AcceptBetter | AcceptBetterEqual
                         |  <accept_probab> | <accept_threshold> | <accept_metropolis>
  <accept_metropolis>  ::=  AcceptMetropolis(<temp_init>, <temp_factor>, <temp_final>,
                                                                <temp_reheat_mode>)
```

Figure 5. Other significant derivation rules in the grammar. Local searches and perturbations can be extended by defining them in the problem-specific grammar.

hybrid method is defined by a perturbation component, an acceptance criterion and an inner GLS. Terminal symbols, that is, those that cannot be further expanded, are indicated in boldface in the grammar. Each recursion of the <GLS> rule creates a new level of hybridization.

Well-known metaheuristics are defined in terms of algorithmic components that are also available to the hybrid methods, as illustrated by the grammar in Fig. 4. On the other hand, the alternatives available for each component are restricted to those that characterize the specific metaheuristic. Additional examples of components are shown in Fig. 5. For example, a perturbation may either have no effect (**PertNone**), discard the current solution and call again the initialisation (**PertRestart**), perform one or k moves within some neighbourhood, or perform a number of moves that follow some dynamic schedule. One rule may expand into a number of components, e.g., `<accept_metropolis>`. Terminals in the grammar are either categorical choices or numerical ranges.

Parts of the grammar derive into problem-specific components, e.g., `<ps:pert_repeatable>`. If those components are not implemented for the problem at hand, then those parts of the grammar will be ignored. This behavior allows for a very flexible problem-independent grammar that specifies where and which type of problem-specific components may be useful, but it does not require the implementation of all the problem-specific components in advance. Moreover, by keeping separated the problem-independent and problem-specific parts of the grammar, we are able to re-use the problem-independent part for different problems, even when not all problem-specific components are implemented.

Finally, our grammar for designing hybrid metaheuristics is quite different from the grammars typically used in genetic programming benchmarks. In particular, the only recursive rule is the one defining hybridisation, there is a relatively larger number of rules and alternatives per rule, and numeric terminals (both integer and real-valued) are represented as ranges.

B. Automatic Design of Hybrid Metaheuristic Algorithms

The above grammars are only a very simplified subset of our current grammar for generating hybrid metaheuristic algorithms. Even if we limit the number of hybridization levels to two or three, by arguing that successful hybrid algorithms almost never contain more than three levels, the number of potential algorithms derived from the complete grammar is large. Given the wide range of algorithmic components available in the literature, evaluating all algorithms that may be generated from the grammar is impossible in practice. Thus, we need a way to search the design space for good instances of algorithms given a problem. Moreover, when automatically designing an algorithm for a specific problem, we are often given a training set of problem instances, representative of the problem instances that the algorithm will tackle in practice, and the goal is to generate an algorithm that performs well over all potential instances of the problem.

Automatic algorithm configuration methods are particularly well-suited for finding good parameter settings given a set of training instances of a problem. For example, IRACE [12] is an iterative racing procedure that applies the following main steps. First, some candidate parameter configurations are sampled from a probability distribution. Each candidate configuration corresponds to instantiating a specific algorithm from an appropriately defined framework, which in our case is GLS. The generated candidate configurations are next evaluated by means of racing on a stream of problem instances. In the race, poor-performing candidate configurations are discarded. A race terminates once the allocated computation budget is exhausted or the number of surviving candidate configurations is below some specific number. The best candidate configurations are then used for biasing the probability distribution that generates new candidate configurations for the next iteration or IRACE. These steps are repeated until exhausting an overall computation budget as given, for example, by a fixed budget of experiments (runs of the algorithm being configured). Racing is a well-known method for selecting the best among a set of potential candidates under uncertainty by iteratively increasing the number of samples (runs of a particular configuration on different problem instances) of the best-performing candidates and discarding the worst-performing ones as soon as some statistical evidence becomes available. IRACE has been successfully applied for tuning the parameters of exact solvers, designing multi-objective algorithms, model selection in machine learning, automatic design of control software for robots and many other tasks, often involving very large parameter spaces [12].

In order to use automatic configuration methods to search within the design space defined by a grammar, we have developed a tool (grammar2code) [20] that converts the grammar into a parameter description file, which describes the type, domain and possible constraints and dependencies of each parameter [12]. The grammar2code tool requires specifying a maximum recursive derivation level, which in our grammar controls the level of hybridization, and it supports a number of useful features: (i) grammar annotations produce appropriate numerical parameters from numerical terminals; (ii) separate grammars can be composed into a larger one, which allows us to maintain separately one common problem-independent grammar and multiple problem-specific ones; (iii) derivation rules that ultimately require some other missing rule are ignored, such that the problem-specific part does not need to implement every terminal specified by the problem-independent grammar; and (iv) grammar2code performs simplifications of the grammar that create more balanced derivation trees.

The other role of grammar2code is to produce an algorithm from the grammar given a candidate parameter configuration. In the implementation of the system used here, there is a static C++ source file into which the code generated by grammar2code is included. When given a candidate configuration, the output of grammar2code is C++ source code that (recursively) constructs a GLS object, where each algorithm component is itself another object (the C++ implementation of what is shown in Fig. 2). The top-level object is a function object that, when called, executes the specific algorithm that was constructed. All the code generated is then compiled and linked against a software library that actually implements the components (in our case, an extended version of ParadisEO [22]). As a result, the final executable

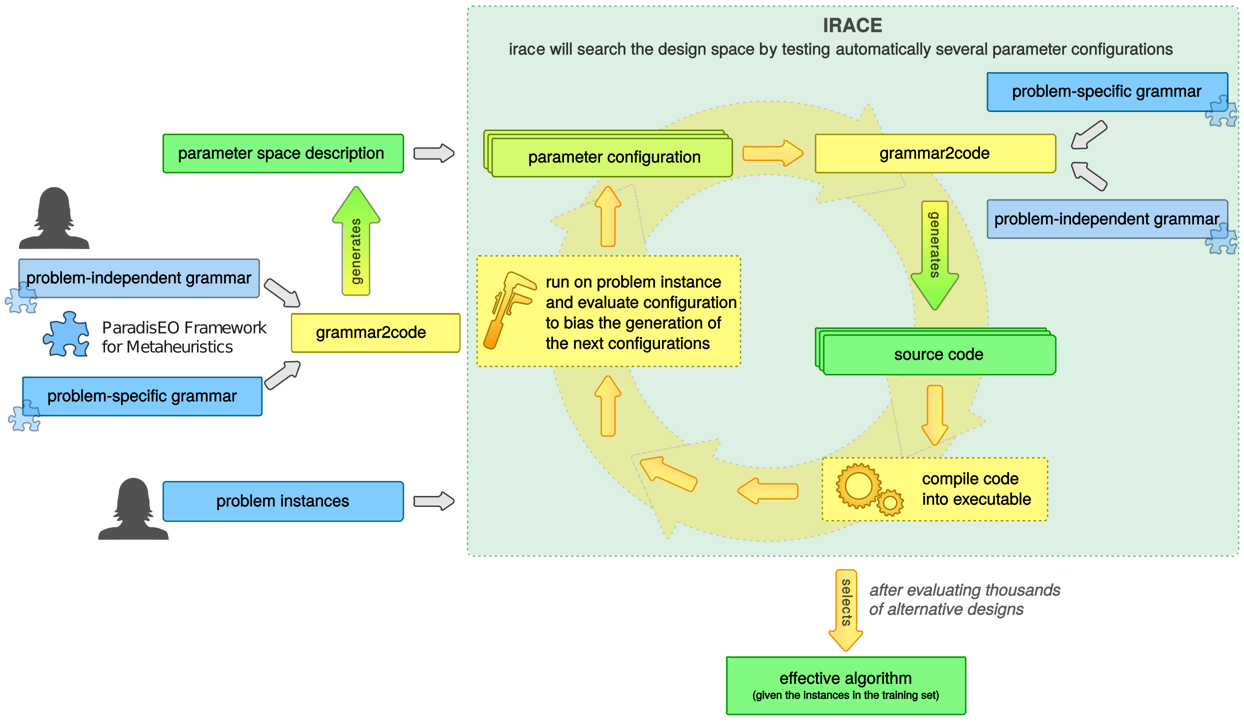
IRACE
irace will search the design space by testing automatically several parameter configurations
problem-specific grammar
parameter space description
parameter configuration
grammar2code
problem-independent grammar
generates
generates
problem-independent grammar
run on problem instance and evaluate configuration to bias the generation of the next configurations
ParadisEO Framework for Metaheuristics
grammar2code
source code
problem-specific grammar
compile code into executable
problem instances
selects
after evaluating thousands of alternative designs
effective algorithm
(given the instances in the training set)

Figure 6. Overview of the automatic design process to generate hybrid metaheuristic algorithm. The user only needs to provide the grammar, the code implementing problem-specific algorithmic components and the problem instances.

only contains the code of the components actually used by the algorithm. This is not the only possible implementation and, for example, grammar2code may generate a high-level representation (e.g., exactly as shown in Fig. 2) that another executable interprets dynamically in order to construct and execute the appropriate algorithm. In this latter implementation, the executable would need to contain all possible algorithmic components, however, dynamic linking and compiler optimizations may offset any potential slowdowns.

An overview of the complete system is shown in Fig. 6. From a grammar, which is divided into problem-independent and problem-specific sub-grammars, grammar2code generates a parameter space description. The level of hybridization is controlled by a parameter that limits the maximum number of derivations of recursive rules. This ensures a finite number of parameters generated from the grammar. The parameter space description together with a set of problem instances are the inputs to the automatic design tool, in our case IRACE. From the parameter space, IRACE will instantiate parameter configurations that correspond to potential designs of hybrid metaheuristic algorithms. These parameter configurations are translated into derivations of the grammar by grammar2code, thus generating source code that, in our current implementation, is compiled into an executable. The generated executable is then run on problem instances to evaluate the performance of the corresponding hybrid metaheuristic algorithm. The executable file is cached on disk as long as the corresponding configuration is not eliminated by IRACE. When IRACE consumes a maximum budget of runs, the search stops and the best configuration, i.e., algorithm design, is returned to the user.

IV. Experimental Evaluation

We applied our automatic design method to three combinatorial optimisation problems: the perturbation flowshop problem with weighted tardiness (PFSP-WT), the unconstrained binary quadratic problem (UBQP), and the travelling salesman problem with time windows (TSPTW). The state-of-the-art algorithms for these problems are hybrid single-solution local search metaheuristics. Thus, we are able to exactly replicate them as instances of our GLS template, after implementing the required problem-specific components.

In the most optimistic case, the automatic design method will be able to outperform the state-of-the-art algorithms by identifying a better hybrid algorithm that uses better components. However, the problems chosen here have been thoroughly studied in the literature and the state-of-the-art components are extremely well-performing already. Thus, a more realistic goal may be to match state-of-the-art performance exploiting known state-of-the-art components or generating similar algorithmic structures, but doing this automatically.

V. Permutation FlowShop Problem with Weighted Tardiness Objective

The permutation flowshop scheduling problem (PFSP) encompasses a variety of problems that are typical of industrial production environments. PFSPs involve scheduling n jobs on m machines under the condition that all jobs are processed in the same order and jobs are not allowed to pass each other. Each job i requires, on each machine j, a fixed, non-negative processing time p_{ij}. In the PFSP-WT, the objective function to be minimized is the total weighted tardiness. Each job i has a due date d_i, which denotes the desired completion time of the job on the last machine, and a priority weight w_i, which denotes its importance. The *tardiness* of a job i is defined as $T_i = \max\{C_i - d_i, 0\}$, where C_i is the completion time of job i on the last machine, and the *total weighted tardiness* is given by $\sum_{i=1}^{n} w_i \cdot T_i$. This problem is $\mathcal{NP}$-hard even for a single machine.

1) State-of-the-art for the PFSP-WT: A state-of-the-art (SOA) iterated greedy algorithm for the PFSP-WT was proposed by Dubois-Lacoste et al. [28]. In this algorithm, the initial solution is constructed using a modified version of the well-known NEH algorithm [29] called NEH-WSLACK [30]. The IG uses a first-improvement local search based on a swap neighborhood that stops after a maximum number of $2(n-1)$ swaps. The perturbation operator removes d randomly chosen jobs from the solution and re-inserts them one by one at the point that minimizes the partial objective function. Finally, the acceptance criterion is a Metropolis condition with fixed temperature (T_c); it accepts a solution that is worse than the current one with a probability given by $\exp(100 \cdot (f(s^*) - f(s'')) / (f(s^*) \cdot T_c))$, where $f(s^*)$ is the objective value of the current solution and $f(s'')$ is the objective value of the new one. Dubois-Lacoste et al. [28] suggest the settings $d = 5$ and $T_c = 1.2$.

2) Local search components for the PFSP-WT: The components of the SOA algorithm were included as problem-specific components of the grammar, together with several other problem-specific alternatives. For example, we added two initialization methods, NEH with and without the WSLACK heuristic. In addition to the random destruction-construction perturbation used by the SOA algorithm, we add further problem-specific perturbations based on classical neighborhood move operators (insert, exchange and swap). Given the rules of the problem-independent grammar, these moves may be applied several times per perturbation and the number of repetitions may be either constant or vary during the run of the algorithm.

3) Experimental Setup: For the automatic design phase, we generated a training set of 5 random instances for each number of jobs in $\{50, 60, 70, 80, 90, 100\}$ and with 20 machines, following the procedure described by Minella et al. [31]. Each parameter configuration is run with a CPU time limit of 30 CPU-seconds on the training instances. We run IRACE three times to generate three algorithms with different maximum levels of hybridization, GLS1, GLS2 and GLS3. A single run of IRACE stops after exhausting a given budget of evaluations, concretely, 30 000 evaluations for GLS1, 40 000 for GLS2, and 50 000 for GLS3. As an independent test set, we consider the Taillard instances, extended with due dates by Minella et al. [31], with 50 and 100 jobs and 20 machines, that is, size `50x20` and `100x20`.

4) Experimental Results: We summarise the results from previous work [32], which used a simpler grammar than the one used in the rest of this paper. In this early grammar, even the classical metaheuristics were wrapped within a GLS, therefore, even when specifying a single level of hybridization, the resulting algorithm will always be a hybrid of a metaheuristic within a GLS. The automatic design was performed with the non-elitist variant of IRACE [12].

The three algorithms generated by IRACE can be briefly described as:

- **GLS1** is an IG algorithm within an ILS. It uses the NEH-WSLACK initialization, then executes a classical ILS with a *k*-insert move as perturbation, IG as the subsidiary local search, and an **AcceptBetterEqual** criterion. The IG uses the perturbation operator of the SOA algorithm, a first-improvement local search and the SOA acceptance criterion.
- **GLS2** uses NEH initialization instead of NEH-WSLACK, then executes a GLS without perturbation, an **AcceptBetterEqual** acceptance criterion and another GLS as localsearch. The inner GLS again has no perturbation, but uses a Metropolis acceptance criterion and a VNS algorithm as local search. The VNS uses a *variable* insert move perturbation, a first-improvement descent, and the **AcceptBetter** acceptance criterion.
- **GLS3** is similar to GLS2, despite being allowed one more level of hybridization. In this case, initialization is done by NEH-WSLACK; the outermost GLS uses a a *k*-exchange move as perturbation and an acceptance criterion that always accepts a new solution. The inner GLS uses the destruct-

Table II

COMPARISON OF THE METHODS THROUGH THE FRIEDMAN TEST BLOCKING ON THE TEST INSTANCES OF THE PFSP-WT. SEE TEXT FOR DETAILS. $\Delta R_{\alpha=0.05}$ GIVES THE MINIMUM DIFFERENCE IN THE SUM OF RANKS BETWEEN TWO METHODS THAT IS STATISTICALLY SIGNIFICANT.

Benchmark set	$\Delta R_{\alpha=0.05}$	Method (ΔR relative to the best ranked)
`50x20`	10.65	**GLS1** (0), **GLS3** (6), **GLS2** (10), SOA (16)
`100x20`	5.18	**GLS3** (0), GLS2 (9), GLS1 (15), SOA (28)

construct perturbation operator of the SOA and a Metropolis acceptance. Finally, the inner-most VNS is identical to GLS2.

Table II compares the results obtained by the automatically designed GLS algorithms against the SOA on the test instances. We ran each algorithm on each instance 15 times with different random seeds and computed the mean objective value per instance. Then, we computed the rank of each algorithm on each instance, summed the ranks of each algorithm for each benchmark set and computed the rank difference relative to the best ranked algorithm (shown in parenthesis). The critical rank-sum difference (ΔR at a significance level of $\alpha = 0.05$) was computed by means of a Friedman test blocking on the instances. Algorithms with a rank difference larger than the critical value perform statistically worse; otherwise, best ranked methods are highlighted in bold-face. For both benchmark sets, the algorithms that have been selected by IRACE are significantly better than SOA. GLS3, in particular, showed a strong improvement over SOA, with the main difference between the two being that GLS3 performs an inner VNS within the IG. This is a slightly unusual design, which may have been difficult to design effectively by a human.

VI. UNCONSTRAINED BINARY QUADRATIC PROGRAMMING PROBLEM

The unconstrained binary quadratic programming problem (UBQP) is a non-linear combinatorial optimization problem that unifies a wide range of important problems including graph coloring, set partitioning, and maximum cut problems [33]. Given a symmetric square matrix Q of size $n \times n$, the objective is to maximize the function $x^T Qx = \sum_{i=1}^{n} \sum_{j=1}^{n} q_{ij} x_i x_j$, where x is a binary vector of size n ($x_i \in \{0,1\}$, $i = 1,\dots,n$).

1) State-of-the-art for the UBQP: In the literature, various algorithms have been applied to solve the UBQP. Exact methods become prohibitively expensive when solving large instances [34]. Hence, metaheuristics have been used to reach acceptable solutions in a reasonable time [35, 36]. Wang et al. [37] proposed a state-of-the-art (SOA) algorithm based on GRASP. Each iteration of the GRASP algorithm starts with a binary vector where all variables are set to 0, and the construction procedure iteratively selects some binary variables to switch their values from 0 to 1 according to some rules (see details in the original paper) with an *rcl* parameter acting like a tabu list size. Then, a tabu search is applied to the constructed solution using a one-flip move neighborhood that switches the value of one variable at each step. When a move is applied, the reverse move becomes tabu for a given number of iterations parametrized by a *TabuTenure* calculated from a parameter *ttc* and a number uniformly generated between 0 to 10 ($\mathit{TabuTenure}(i) = \mathit{ttc} + \mathrm{rand}(10)$, indicating that the variable i of solution x can not be flipped for $\mathit{TabuTenure}(i)$ iterations). An aspiration criterion is added to allow a tabu flip when it leads to a better solution than the current best solution. The tabu search stops after a maximum number of moves without improvement μ. The authors suggest to set $\mathit{rcl} = 50$, $\mathit{ttc} = n/100$ and $\mu = 5n$, where n is the problem size.

2) Local search components for the UBQP: We implemented several greedy and randomized greedy heuristics for the construction of initial solutions [35, 37], which may also be used as constructive heuristics within a deconstruct/reconstruct perturbation. Other problem-specific components are tabu search [36, 37], and 1-opt and k-opt neighborhoods [35]. The rules of the problem-independent grammar of the PFSP-WT are kept, hence, the perturbations may be composed of a (constant or varying) number of neighborhood moves.

3) Experimental Setup: The most popular UBQP instances can be separated into two classes, depending on the characteristics of the Q matrix. Several instances with sparse matrices of different sizes are available from OR-lib (http://people.brunel.ac.uk/~mastjjb/jeb/info.html),while Gintaras Palubeckis (https://www.personalas.ktu.lt/~ginpalu/ubqop_its.html) proposed instances with dense matrices. These instances have been used in the literature to evaluate the SOA algorithm [37] and we use them as test set.

For the automatic design phase, we generated two different training sets: 30 *sparse* instances with sizes 1000 and 2500, stopping each algorithm run after 120 seconds; and 30 *dense* instances with sizes 3000 and 4000, stopping each algorithm run after 600 seconds. We ran IRACE separately on each training set with three different maximum levels of hybridization. Each run of IRACE executes a maximum of 100 000 runs and returns one GLS design.

The generated GLS algorithms are then evaluated on the test instances. The GLS algorithms generated using the sparse instances are evaluated on the 10 OR-lib instances of size 1000 and 2500, while the algorithms generated using the dense instances are evaluated on the 5 Gintaras instances of size 3000, 4000, 5000, 6000 and 7000. On the test set, the maximum execution time of the algorithms was 2, 2, 10, 20, 60, 60, and 120 minutes, respectively. These values correspond to the time that our implementation of the SOA, running in our hardware, requires to match the results reported in the original paper [37].

4) Experimental Results: Following the setup above, IRACE generated a GLS variant for each maximum level of recursion (1, 2 or 3) and each type of instance (*s*parse or *d*ense). The GLS algorithms generated for *sparse* instances are variants of the SOA where the main differences are the settings of two numerical parameters (*rcl* and μ). The initialisation from Wang et al. [37] is always selected along with a restart

Table III

COMPARISON OF THE METHODS THROUGH THE FRIEDMAN TEST BLOCKING ON THE TEST INSTANCES OF THE UBQP.

Benchmark set		$\Delta R_{\alpha=0.05}$	Method (ΔR relative to the best ranked)
1000	*sparse*	∞	**SOA** (0), **GLS$_{1s}$** (0), **GLS$_{2s}$** (0), **GLS$_{3s}$** (0)
2500	*sparse*	∞	**SOA** (0), **GLS$_{2s}$** (5), **GLS$_{3s}$** (6), **GLS$_{1s}$** (7)
3000	*dense*	4.12	**GLS$_{3d}$** (0), **SOA** (0.5), **GLS$_{2d}$** (2.5), GLS$_{1d}$ (11)
4000	*dense*	∞	**SOA** (0), **GLS$_{2d}$** (0), **GLS$_{3d}$** (0), **GLS$_{1d}$** (8)
5000	*dense*	5.34	**GLS$_{3d}$** (0), **GLS$_{2d}$** (1), **SOA** (5), GLS$_{1d}$ (12)
6000	*dense*	0	**GLS$_{3d}$** (0), GLS$_{2d}$ (3), SOA (6), GLS$_{1d}$ (9)
7000	*dense*	0	**GLS$_{3d}$** (0), GLS$_{2d}$ (3), SOA (6), GLS$_{1d}$ (9)

perturbation, which creates a GLS similar to a GRASP algorithm that uses a tabu search as local search. Moreover, the three GLSs have a double stopping condition of an iteration based on both a maximum number of moves without improvement μ and a maximum time allocated to the tabu search (*tts*). In addition, the acceptance criterion accepts better or equal quality solutions for both GLS$_{2s}$ and GLS$_{3s}$. The differences are: GLS$_{1s}$ uses $rcl = 73$, $\mu = 10n$ (instead of $rcl = 50$ and $\mu = 5n$ for SOA) and $tts = 10\%$, *i.e.*, each application of the tabu search cannot exceed 10% of the time allocated to the whole run; GLS$_{2s}$ uses $rcl = 87$, $\mu = 7n$ and $tts = 10\%$; and GLS$_{3s}$ uses $rcl = 53$, $\mu = 9n$ and $tts = 50\%$.

Looking at the GLS variants generated for the *dense* instances, it appears that the simple GRASP-like structure that was preferred for the sparse instances is not effective anymore, and that more complex hybrids are generated:

- GLS$_{1d}$ is a classical tabu search algorithm. It uses the random variant of the greedy initialisation proposed by Merz and Freisleben [35] and then executes a tabu search. The tabu list is the same as in the SOA with the same value of parameter *ttc*.
- GLS$_{2d}$ uses the construction of the SOA algorithm as initialisation (with $rcl = 53$), then executes a GLS without perturbation, an **AcceptBetter** acceptance criterion, and another GLS as local search. The inner GLS is a variant of SOA with $\mu = 8n$ and $tts = 40\%$.
- GLS$_{3d}$ uses a random initialisation then executes a hybrid GLS with three levels. The outer-most level uses a one-flip perturbation of strength randomly chosen between 21 and 32 and an acceptance criterion that accepts a worse solution if its relative error w.r.t. the current one is lower than 0.8535. Its local search is another GLS with a one-flip perturbation of strength randomly chosen between 29 and 38, a probabilistic acceptance criterion (with probability of 0.5), and another GLS as local search. This inner-most local search is the tabu search of the SOA algorithm with $\mu = 9n$ and $tts = 40\%$.

For comparing the generated algorithms, we ran each method on the test instance set, repeating each run 15 times with different random seeds. We compare the algorithms using the Friedman test as explained for the PFSP-WT (Sec. V-4 on page 11) and the results of this analysis are summarised in Table III. On the sparse instances with sizes 1000 and 2500, it was not possible to reject the null hypothesis and find a statistically significant difference between the algorithms. This is not surprising since the generated GLSs are nearly identical to the SOA algorithm. On the dense instances, the generated algorithms are always as good as the SOA method, but the advantage in favour of the generated methods increases with the instance size. On the largest instances, GLS_{3d} is significantly better than SOA.

VII. Travelling Salesman Problem with Time Windows (TSPTW)

The TSPTW models the problem of visiting a number of customers, starting and ending from the same point, where the travel time between customers is known and each customer must be visited within a given time window with the goal of minimizing total travel time. The TSPTW has been a target of metaheuristics research since several decades, because of its difficulty —it is NP-hard and even finding a feasible solution is NP-complete— and its connection with other important problems—the well-known travelling salesman problem (TSP) is a special case of the TSPTW. The TSPTW itself is the single vehicle case of the vehicle routing problem with time windows.

1) State-of-the-art for the TSPTW: Several high-quality studies have been published in recent years analysing a large number of metaheuristics for the TSPTW [38, 39]. Based on these studies, Ferreira da Silva and Urrutia [40] proposed a hybrid VNS that, to the best of our knowledge, is the most effective metaheuristic algorithm available for several classes of instances. This hybrid VNS, called SOA henceforth, is a hybrid with two levels. The outer level creates a new initial solution whenever the inner level terminates naturally, while keeping track of the best solution found so far. The inner level is a VNS strategy that combines a variable perturbation component performing k moves within a problem-specific insertion neighborhood that maintains feasibility and a local search based on variable neighborhood descent (VND). The value of parameter k starts at 1, it is incremented by one every eight iterations up to a maximum value of 31 (the original paper says 30, yet the code provided by the original authors uses 31) and it is set back to 1 if a new solution improves over the current one. The VND first applies a 1-shift local search and next a 2-opt local search until no improvement is found. Both local search steps maintain feasibility. Finally, the initialization procedure is also a VNS method that starts from a random solution, uses a 1-shift neighborhood as subsidiary local search, and a perturbation based on the insertion neighborhood with a perturbation strength that varies between 1 and $n/2$, where n is the instance size. Both neighborhoods allow infeasible solutions. The algorithm described in [40] differs in some details from the description above, but a close examination of the original source code kindly provided by the authors revealed all the details of the actual implementation and allowed us to reproduce the results reported in [40].

2) Local search components for the TSPTW: In addition to the components of the SOA algorithm, we added other problem-specific components to the grammar. As alternative initialization methods, we added:

a purely random initialization; a constructive heuristic [39]; and improving this initial solution by means of local search or VND using any of the neighborhoods available that handle infeasible solutions. There were subtle differences between the neighborhoods described in the original paper and the ones implemented in the code, thus we added to the grammar all variants, including variants that handle infeasible solutions and variants that only work with feasible ones. The VND may use any sequence of neighborhoods. We also implemented a destruct-construct perturbation that randomly removes a number of nodes from the tour and attempts to re-insert them in the position that optimizes the current partial solution. Finally, we added two tabu list operators [41].

3) Experimental Setup: Following the paper that proposed the SOA, we combined four different sets of benchmark instances: *Dumas*, 135 instances grouped into 27 classes with sizes between 20 and 200 nodes [42], *GendreauDumasExtended*, 120 instances grouped into 26 classes obtained by extending the time windows of Dumas instances [43]; *OhlmannThomas*, 25 instances grouped into five classes, derived from the instances with 150 and 200 nodes proposed by Dumas [38]; and *DaSilvaUrrutia*, 125 instances grouped in 25 classes, with 200 to 400 nodes generated by a different method [40] than Dumas et al. [42].

We split the above instances into a training set used during the automatic design phase and a test set used for comparing the generated metaheuristics with the SOA. We used the first two instances of each class in the training set and the remaining three instances of each class in the test set. The configuration for the TSPTW used two special considerations. To avoid a bias towards a specific instance set, we used the instance blocking possibility available in IRACE [12]. As the termination criterion, we use a CPU-time limit that is determined per instance by running the SOA with the same termination criterion as the original paper (30 restarts) and computing the mean CPU-time over 15 runs with values rounded up to the nearest integer second. As above, we performed three runs of IRACE with different hybridization limits that resulted in three metaheuristics, namely, GLS1, GLS2 and GLS3. The budget for each run of IRACE was $100,000$ runs of the hybrid designs.

4) Experimental Results: The three algorithm (GLS1, GLS2 and GLS3) generated by IRACE are VNS methods using VND as a local search. Surprisingly, in all of them, the VND examines first the 2-opt neighborhood, then the insertion one, which is the opposite of what the SOA does. Their main differences are:

- In GLS1, the initialization uses VNS with a perturbation in the insertion neighborhood. The outer VNS behaves like an ILS, with a perturbation strength that goes from 27 to 32, every 30 iterations.
- In GLS2, initialization is a random solution improved by a local search in the insertion neighborhood. The outer VNS uses a uniformly random perturbation strength in the interval $[12, 16]$, which again behaves like an ILS.

Table IV
COMPARISON OF THE METHODS THROUGH THE FRIEDMAN TEST BLOCKING ON THE TEST INSTANCES OF THE TSPTW.

Benchmark set	$\Delta R_{\alpha=0.05}$	Method (ΔR)
Dumas	∞	**SOA** (0), **GLS3** (7.5), **GLS1** (10), **GLS2** (12.5)
GendreauDumasExtended	∞	**GLS3** (0), **SOA** (1.5), **GLS1** (12.5), **GLS2** (18)
OhlmannThomas	10.93	**GLS3** (0), **SOA** (4), GLS1 (15), GLS2 (23)
DaSilvaUrrutia	27.81	**SOA** (0), **GLS3** (12), **GLS2** (26.5), GLS1 (45.5)

- GLS3 uses the same initialization as the SOA. The outer VNS uses a fixed perturbation strength of 16, hence, it is actually an ILS.

The three generated metaheuristics (GLS1, GLS2 and GLS3) and the SOA algorithm were run 15 times on each instance of the test set and the mean travel cost per instance was computed. Infeasible solutions are penalised by multiplying the number of constraint violations by a constant larger than any travel cost. After running the SOA algorithm and the three GLSs described above on the test set, we analyze them according to rank differences by means of the Friedman test, as explained for the PFSP-WT (Sec. V-4 on page 11).

Table IV summarizes the results. Both, the SOA method and GLS3, are always ranked among the best ones. The strongest differences are observed on the *OhlmannThomas* and *DaSilvaUrrutia* sets, which are the largest and hardest ones. The fact that all GLSs generated are very similar to each other and to the SOA algorithm, and their results are similar as well, suggests that there are no algorithmic components in our grammar that outperform the ones from the SOA algorithm. This is not surprising as, beyond basic problem-independent components, we implemented only few problem-specific alternatives that were probably already considered by the designers of the SOA algorithm. A somewhat surprising finding is that reversing the order of the neighborhoods in the VND may perform better. Moreover, the fact that all generated algorithms use VND as local search indicates that this is a critical component. These insights could guide a deeper analysis of algorithmic components and possible extensions of the grammar.

VIII. CONCLUSIONS

In this paper, we presented a system for automatically designing hybrid metaheuristic algorithms. Our system differs from previous proposals in three key insights. First, instead of generating algorithms from very simple operators, we assume that a library of effective algorithmic components is available. This assumption does not preclude the possibility that the components are themselves generated automatically [16, 19], but allows known human-designed components to be readily used. The latter are often extremely efficient and effective, which is essential to match or surpass the performance of the best known algorithms in practice. Second, instead of creating a problem-specific algorithmic template and trying to evolve its components [17, 19] or a very general template that does not directly match most state-of-the-art methods [21], we devised a problem-independent template that is still able to replicate many standard algorithms from the literature and hybrids thereof. A clear benefit is that existing algorithms can be exactly replicated, while allowing re-using the template and any problem-independent components for a wide range of problems. Third, the method actually searching the space of potential designs is an off-the-shelf automatic configuration method (IRACE) instead of an application-specific method. Automatic configuration methods are very general and widely applicable, designed for computer-intensive experiments and able to optimize not only the design, but also any numerical parameters of the algorithms. The latter is more difficult when using other methods [17, 18]. Moreover, improvements in automatic configuration methods directly lead to more effective automatic design.

Given the proposed system, we were able to automatically design, for three very different problems, various hybrid metaheuristic algorithms that match, and sometimes outperform, human-designed state-of-the-art algorithms from the literature. As a side benefit, the system becomes a repository of algorithmic components (and their potential compositions) that accumulates the knowledge present in the literature (both problem-independent and problem-specific). The mere act of extending the system implies the identification of the actual commonalities and differences among algorithmic components beyond any superficial differences in terms of nomenclature, thus, leading to a standardization of the design and development of metaheuristic algorithms. Moreover, the automatic design process is a partial empirical comparison of components and the data generated should be used to further analyze the impact and usefulness of the components. Thus, it leaves human algorithm designers more time to focus on devising better components and analysing their usefulness.

The system presented here is a proof-of-concept and future work could greatly extend its usefulness. We limited ourselves to a single run for generating each algorithm, and we did not examine how difficult is to generate effective algorithms and the variance across design runs. One practical limitation is that our system is build on top of an existing algorithmic framework [22] that was not designed for our grammar. Ongoing work shows that a more efficient and simpler implementation is possible by building the algorithmic framework from scratch. A more general limitation is the exclusion of population-based algorithms from our template. Future research should also aim at devising more general grammars that still satisfy the trade-offs achieved by our grammar as discussed above. Finally, in some sense, the grammars presented here are limited by the state-of-the-art algorithmic components, since we did not attempt to devise better components nor implement all components proposed in the literature for each problem. Future work would incorporate any promising algorithmic components into the system, therefore monotonically increasing the power of the system and keeping a record of the progress in heuristic optimization.

REFERENCES

[1] F. Glover and M. Laguna, *Tabu Search*. Boston, MA, USA: Kluwer Academic Publishers, 1997.

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

[3] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley, 1989.

[4] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Cambridge, MA: MIT Press, 2004.

[5] V. Černý, "A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," *Journal of Optimization Theory and Applications*, vol. 45, no. 1, pp. 41–51, 1985.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[7] H. Ramalhinho Lourenço, O. Martin, and T. Stützle, "Iterated local search," in *Handbook of Metaheuristics*, F. Glover *et al.*, Eds. Kluwer Academic Publishers, Norwell, MA, 2002, pp. 321–353.

[8] P. Hansen and N. Mladenović, "Variable neighborhood search: Principles and applications," *Eur. J. Oper. Res.*, vol. 130, no. 3, pp. 449–467, 2001.

[9] M. G. C. Resende and C. C. Ribeiro, "Greedy randomized adaptive search procedures," in *Handbook of Metaheuristics*, F. Glover *et al.*, Eds. Kluwer Academic Publishers, Norwell, MA, 2002, pp. 219–249.

[10] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: an automatic algorithm configuration framework," *J. Artif. Intell. Res.*, vol. 36, pp. 267–306, Oct. 2009.

[11] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization, 5th International Conference, LION 5*, ser. LNCS, C. A. Coello Coello, Ed. Springer, 2011, vol. 6683, pp. 507–523.

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

[13] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "SATenstein: Automatically building local search SAT Solvers from Components," *Artificial Intelligence*, vol. 232, pp. 20–42, 2016.

[14] M. López-Ibáñez and T. Stützle, "The automatic design of multi-objective ant colony optimization algorithms," *IEEE Trans. Evol. Comput.*, vol. 16, no. 6, pp. 861–875, 2012.

[15] L. C. T. Bezerra, M. López-Ibáñez, and T. Stützle, "Automatic component-wise design of multi-objective evolutionary algorithms," *IEEE Trans. Evol. Comput.*, vol. 20, no. 3, pp. 403–417, 2016.

[16] J. A. Vázquez-Rodríguez and G. Ochoa, "On the automatic discovery of variants of the NEH procedure for flow shop scheduling using genetic programming," *J. Oper. Res. Soc.*, vol. 62, no. 2, pp. 381–396, 2010.

[17] E. K. Burke, M. R. Hyde, and G. Kendall, "Grammatical evolution of local search heuristics," *IEEE Trans. Evol. Comput.*, vol. 16, no. 7, pp. 406–417, 2012.

[18] J. Tavares and F. B. Pereira, "Automatic design of ant algorithms with grammatical evolution," in *EuroGP 2012*, ser. LNCS, A. Moraglio *et al.*, Eds. Springer, 2012, vol. 7244, pp. 206–217.

[19] A. S. Fukunaga, "Automated discovery of local search heuristics for satisfiability testing," *Evol. Comput.*, vol. 16, no. 1, pp. 31–61, Mar. 2008.

[20] F. Mascia, M. López-Ibáñez, J. Dubois-Lacoste, and T. Stützle, "Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools," *Comput. Oper. Res.*, vol. 51, pp. 190–199, 2014.

[21] R. J. M. Vaessens, E. H. L. Aarts, and J. K. Lenstra, "A local search template," *Comput. Oper. Res.*, vol. 25, no. 11, pp. 969–979, 1998.

[22] J. Humeau, A. Liefooghe, E.-G. Talbi, and S. Verel, "ParadisEO-MO: From fitness landscape analysis to efficient local search algorithms," *J. Heuristics*, vol. 19, no. 6, pp. 881–915, Jun. 2013.

[23] R. Ruiz and T. Stützle, "A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem," *Eur. J. Oper. Res.*, vol. 177, no. 3, pp. 2033–2049, 2007.

[24] F. Glover, "Tabu search – Part I," *INFORMS Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[25] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp. 1087–1092, 1953.

[26] E.-G. Talbi, "A taxonomy of hybrid metaheuristics," *J. Heuristics*, vol. 8, no. 5, pp. 541–564, 2002.

[27] C. Blum, J. Puchinger, G. R. Raidl, and A. Roli, "Hybrid metaheuristics in combinatorial optimization: A survey," *Applied Soft Computing*, vol. 11, no. 6, pp. 4135–4151, 2011.

[28] J. Dubois-Lacoste, M. López-Ibáñez, and T. Stützle, "A hybrid TP+PLS algorithm for bi-objective flow-shop scheduling problems," *Comput. Oper. Res.*, vol. 38, no. 8, pp. 1219–1236, 2011.

[29] M. Nawaz, E. Enscore, Jr, and I. Ham, "A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem," *Omega*, vol. 11, no. 1, pp. 91–95, 1983.

[30] J. Dubois-Lacoste, "A study of Pareto and two-phase local search algorithms for biobjective permutation flowshop scheduling," Master's thesis, IRIDIA, Université Libre de Bruxelles, Belgium, 2009.

[31] G. Minella, R. Ruiz, and M. Ciavotta, "A review and evaluation of multiobjective algorithms for the flowshop scheduling problem," *INFORMS Journal on Computing*, vol. 20, no. 3, pp. 451–471, 2008.

[32] M.-E. Marmion, F. Mascia, M. López-Ibáñez, and T. Stützle, "Automatic design of hybrid stochastic local search algorithms," in *Hybrid Metaheuristics*, ser. LNCS, M. J. Blesa *et al.*, Eds. Springer, 2013, vol. 7919, pp. 144–158.

[33] G. A. Kochenberger, F. Glover, B. Alidaee, and C. Rego, "A unified modeling and solution framework for combinatorial optimization problems," *OR Spektrum*, vol. 26, no. 2, pp. 237–250, 2004.

[34] E. Boros, P. L. Hammer, and G. Tavares, "Local search heuristics for quadratic unconstrained binary optimization (QUBO)," *J. Heuristics*, vol. 13, no. 2, pp. 99–132, 2007.

[35] P. Merz and B. Freisleben, "Greedy and local search heuristics for unconstrained binary quadratic programming," *J. Heuristics*, vol. 8, no. 2, pp. 197–213, 2002.

[36] Z. Lü, F. Glover, and J.-K. Hao, "A hybrid metaheuristic approach to solving the UBQP problem," *Eur. J. Oper. Res.*, vol. 207, no. 3, pp. 1254–1262, 2010.

[37] Y. Wang, Z. Lü, F. Glover, and J.-K. Hao, "Probabilistic GRASP-tabu search algorithms for the UBQP problem," *Comput. Oper. Res.*, vol. 40, no. 12, pp. 3100–3107, 2013.

[38] J. W. Ohlmann and B. W. Thomas, "A compressed-annealing heuristic for the traveling salesman problem with time windows," *INFORMS Journal on Computing*, vol. 19, no. 1, pp. 80–90, 2007.

[39] M. López-Ibáñez and C. Blum, "Beam-ACO for the travelling salesman problem with time windows," *Comput. Oper. Res.*, vol. 37, no. 9, pp. 1570–1583, 2010.

[40] R. Ferreira da Silva and S. Urrutia, "A general VNS heuristic for the traveling salesman problem with time windows," *Discrete Optimization*, vol. 7, no. 4, pp. 203–211, 2010.

[41] W. B. Carlton and J. W. Barnes, "Solving the traveling-salesman problem with time windows using tabu search," *IIE Transactions*, vol. 28, pp. 617–629, 1996.

[42] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon, "An optimal algorithm for the traveling salesman problem with time windows," *Operations Research*, vol. 43, no. 2, pp. 367–371, 1995.

[43] M. Gendreau, A. Hertz, G. Laporte, and M. Stan, "A generalized insertion heuristic for the traveling salesman problem with time windows," *Operations Research*, vol. 46, pp. 330–335, 1998.