European Journal of Operational Research 318 (2024) 740–751

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/eor

Discrete optimization

On the automatic generation of metaheuristic algorithms for combinatorial optimization problems

Raúl Martín-Santamaría [a], Manuel López-Ibáñez [b], Thomas Stützle [c], J. Manuel Colmenar [a,*]

[a] Department of Computer Science and Statistics, Universidad Rey Juan Carlos, Móstoles, Spain
[b] Alliance Manchester Business School, University of Manchester, Manchester, UK
[c] IRIDIA, Université Libre de Bruselles, Brussels, Belgium

ARTICLE INFO

Keywords:
Metaheuristics
Methodology
Reproducibility
Automatic configuration

ABSTRACT

Metaheuristic algorithms have become one of the preferred approaches for solving optimization problems. Finding the best metaheuristic for a given problem is often difficult due to the large number of available approaches and possible algorithmic designs. Moreover, high-performing metaheuristics often combine general-purpose and problem-specific algorithmic components. We propose here an approach for automatically designing metaheuristics using a flexible framework of algorithmic components, from which algorithms are instantiated and evaluated by an automatic configuration method. The rules for composing algorithmic components are defined implicitly by the properties of each algorithmic component, in contrast to previous proposals, which require a handwritten algorithmic template or grammar. As a result, extending our framework with additional components, even problem-specific or user-defined ones, automatically updates the design space. Furthermore, since the generated algorithms are made up of components, they can be easily interpreted. We provide an implementation of our proposal and demonstrate its benefits by outperforming previous research in three distinct problems from completely different families: a facility layout problem, a vehicle routing problem and a clustering problem.

1. Introduction

Stochastic local search algorithms, and specially metaheuristics, are one of the most successful methods for solving optimization problems. Their distinct performance, specially when short computing times are required in practical applications, makes them considerably popular (Gendreau & Potvin, 2019; Hoos & Stützle, 2005). The design of metaheuristics is mostly guided by human expertise, intuition and a great deal of trial-and-error, where the same problem-independent algorithmic components are recombined in various ways together with other problem-specific components. To avoid this human-intensive iterative design process that often fails to fully explore the space of possible metaheuristic designs, there is an increasing interest in automatically designing metaheuristics from a library of algorithmic components given training instances of a problem.

There are two main approaches to make this process more automatic. One approach is hyper-heuristics (Ross, 2005) that select or generate low-level heuristics within a given algorithmic template. Although hyper-heuristics are useful for selecting or generating problem-specific heuristics at a few critical points of a given metaheuristic (Burke et al., 2013), their application for designing complete metaheuristics has been limited. One common characteristic of hyper-heuristic approaches is the tight coupling between the algorithmic framework that defines the design space and the optimizer that searches this design space. As a result of this tight coupling, hyper-heuristics are often problem-specific.

A second approach formulates the automatic design of metaheuristics as an automatic algorithm configuration (AAC) problem, also known as parameter tuning or hyperparameter optimization in machine learning. In such an approach, the design space is described as a meta-algorithm with a large number of numerical, categorical and conditional parameters, which must be set to instantiate a specific metaheuristic. This meta-algorithm is developed independently of the optimizer that searches the design space, which is often an off-the-shelf AAC method, such as `irace` (López-Ibáñez et al., 2016), based on F-race (Birattari et al., 2002). Examples of this approach are SATenstein KhudaBukhsh et al. (2009, 2016), AutoMOACO (López-Ibáñez & Stützle, 2012), AutoMOEA (Bezerra et al., 2016) and PSO-X (Camacho-Villalón et al., 2021). These examples were able to produce novel algorithms, outperforming human-designed state-of-the-art ones. These successes have inspired significant changes in popular metaheuristic frameworks such as ParadisEO (Cahon et al., 2004) and jMetal (Durillo & Nebro, 2011). These frameworks were proposed to facilitate the manual (*i.e.*, non-automatic) design of metaheuristics, but they have been

* Corresponding author.
E-mail addresses: raul.martin@urjc.es (R. Martín-Santamaría), manuel.lopez-ibanez@manchester.ac.uk (M. López-Ibáñez), stuetzle@ulb.ac.be (T. Stützle), josemanuel.colmenar@urjc.es (J.M. Colmenar).

https://doi.org/10.1016/j.ejor.2024.06.001
Received 23 February 2024; Accepted 2 June 2024
Available online 6 June 2024

recently extended with meta-algorithm templates whose components are exposed as parameters, thus enabling automatic design via AAC methods (Dréo et al., 2021; Nebro et al., 2019).

Stützle and López-Ibáñez (2019) categorize these approaches for the automated design of metaheuristics as *top-down*, because their design space is defined as a high-level algorithmic template with a large number of parameters. *Top-down* approaches impose a rather inflexible algorithmic structure that is difficult to extend with new components and to other problems. In contrast, *bottom-up* approaches define algorithmic components and rules for combining them, often in the form of a grammar. Many hyper-heuristics follow a *bottom-up* approach and rely on different algorithms for building heuristics: grammatical evolution (Burke et al., 2012), genetic algorithms (Hassan & Pillay, 2017, 2018, 2019), and genetic programming (Iturra et al., 2021; Silva-Muñoz et al., 2023), to name a few of them. Reinforcement learning has been recently applied in the automatic design of algorithms following the *bottom-up* approach (Meng & Qu, 2021; Yi et al., 2022). AAC methods can also optimize a parameter space defined by a grammar (Mascia et al., 2013). If this grammar defines a metaheuristic design space that combines problem-independent and problem-specific components, the result is a flexible approach for the automatic design of metaheuristics (Mascia et al., 2014). A recent demonstration of this *bottom-up* approach is the `Emili` framework (Pagnozzi & Stützle, 2019), which has been used to generate new state-of-the-art metaheuristics for a number of different problems (Alfaro-Fernández et al., 2020; Pagnozzi & Stützle, 2021).

Despite the obvious benefits of the *bottom-up* approach versus the *top-down* one in terms of flexibility, the design of the grammar requires a careful consideration to make sure that all algorithmic components available are enumerated at the appropriate places, together with their parameters, relationships and possible constraints. This is a process that is both tedious and error-prone. Furthermore, its extension with new algorithmic components requires manually modifying the grammar to insert the new components in the relevant places, which is also error-prone.

In contrast to this tedious and error-prone manual update, Swan et al. (2022) argue that automatic design frameworks should be "truly extensible algorithm templates that support reuse without modification", which is closely related to the Liskov substitution principle (Martin, 1996a), also known as *design by contract*, and the *open-closed principle* (Martin, 1996b), which states that components should be extendable rather than modifiable.

We propose here an automatic design framework that follows the above principles. In particular, the requirements implement the *design by contract* that allows substitution of components without modification. In addition, our framework implements the *open-closed principle* by discouraging modification of algorithmic components. Not allowing users to modify existing framework components may give the impression that the framework is not flexible, which may seem as a downside. On the contrary, instead of modifying existing components, which may break their requirements, we encourage users to split algorithmic components into as many independent components as possible with clear requirements, which allows the user to easily extend and override each component's behavior, while matching the same set of requirements.

The proposed automatic design framework follows a *bottom-up* approach that differs from previous works in two key ideas. First, algorithmic components are annotated with their requirements, such that any component can be swapped with any other component that satisfies the same requirements. For example, all constructive methods can be used in place of each other. Although the idea of annotating source code to allow the automatic discovery of parameters to be tuned is not new (Hoos, 2012), previous proposals require an explicit algorithmic template or grammar that defines the relationships between algorithmic components. Our second key contribution is that extending the framework does not require the modification of existing components, nor of an explicit grammar or algorithmic template. Instead, the requirements associated to the algorithmic components give rise to an implicit grammar that is automatically updated when new algorithmic components are added.

With the help of this framework, we propose in this work a methodology that automates the design of metaheuristic algorithms from a given set of algorithmic components. To this aim, we provide a classification of components that includes all algorithmic elements and their relationships. We also describe a labeling nomenclature that allows the automatic discovery and analysis of components and automatic grammar generation. Besides, we propose a new strategy for automatically transforming the grammar to a parameter space that is searched by an AAC method, using an intermediate language that allows any framework user to easily interpret and analyze the results. The AAC method is treated as a black-box component, with the ability of being swapped by any other AAC method if so desired by the user.

In summary, our work addresses the following challenges:

- Our proposed approach does not require a fixed algorithmic template or an explicit grammar that must be manually updated when extending the library of algorithmic components.
- Despite the above, the algorithmic framework is fully extensible by the user because an implicit grammar is automatically built from the relationships defined between components.
- The definition of the relationships between components in our approach only requires annotation of new algorithmic components added to the framework and does not require modification of existing components, following the *open-closed* principle advocated by Swan et al. (2022). Moreover, the proposed intermediate language allows researchers to easily interpret the generated designs.

To the best of our knowledge, our proposal is the first and only framework for automatic design of metaheuristics with the above features. We also explain how these ideas can be implemented using different programming languages by future or existing frameworks like ParadisEO (Cahon et al., 2004), and jMetal (Durillo & Nebro, 2011).

In addition to the above methodological contributions to automatic metaheuristic design, we demonstrate our approach by producing state-of-the-art metaheuristics for three different combinatorial optimization problems, without providing existing knowledge in the form of initial algorithm designs. Moreover, we provide all information needed to re-implement the automatically-generated algorithms as well as the source code of our own implementation.

The rest of the paper is organized as follows. First, we present the full methodology in Section 2. Afterward, in Section 3, the target problems are defined, and a summary of the state of the art of each problem is presented. Next, we detail the experimentation methodology in Section 4, and compare the performance of the automatically generated algorithms using the proposed methodology against the state-of-the-art metaheuristic. Finally, the conclusions and future line of research are presented in Section 5.

2. A new methodology for automatic design of metaheuristics

In this section, the proposed methodology is introduced. As seen in Fig. 1, the methodology consists of two main phases: (1) component discovery and automatic grammar generation, and (2) design by optimization. Next, we describe the details of both phases of the proposed methodology in a general way.

2.1. Component discovery and grammar processing

The first phase of the methodology is implemented by three different steps: the *scanner*, which scans the annotated source code of all components, including user-provided code, and detects all available algorithmic components; the *recursive grammar walker*; and the *parameter space transformer*, as seen on the left side of Fig. 1. These steps are implemented as independent software modules, each one having a single responsibility. Before detailing their behavior, some definitions are introduced in the next subsection.

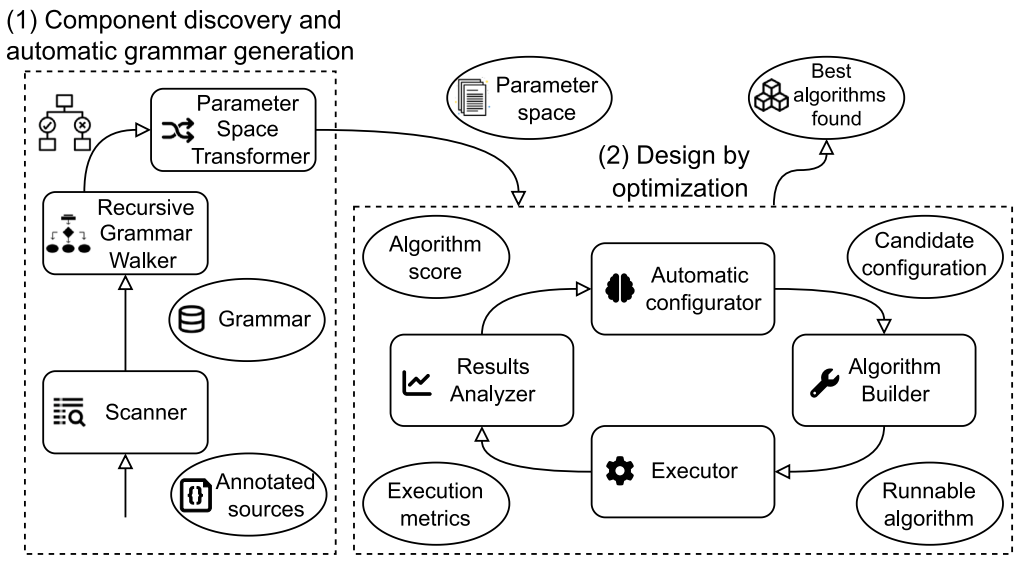


Fig. 1. Complete picture of the proposed automated metaheuristic design framework.

2.1.1. Components and relationships

We define *algorithmic component* as any procedure, method or parameter used or required by any heuristic or metaheuristic method. This definition could be more general, but it is sufficient for our purposes. Algorithmic components may be composed of other simpler algorithmic components. Examples of algorithmic components are constructive methods, stopping conditions, neighborhoods and local search methods. All heuristic and metaheuristic methods are considered as algorithmic components themselves, as they can be used by other metaheuristics.

We define *parameter domain* as the set of valid values that a given parameter can take. Optionally, any component may define *metadata* properties. For example, a numeric parameter may define the minimum and maximum values it can take.

Components can have two types of relationships: they can depend on each other, or they can provide or implement a given functionality. From this point on, we will call them `requires` and `is a` relationships, respectively. Any algorithmic component can `requires` any other component type. This dependency will be automatically resolved to any available component that fulfills the required functionality. As an example, if a component called *SimpleAlgorithm* `requires` a component of type *Constructive* in order to work, this dependency can be satisfied by any component that `is a` *Constructive* method, or in other words, any component that matches the same specification defined by the *Constructive* component by means of inheritance, composition or any other available mechanism of the programming language used. Note that while relationships between components can be as complex as necessary, there need to be simple components that do not depend on any other components; otherwise, no functional algorithm could be constructed.

We have already implemented a set of algorithmic components in a publicly available framework[1] in order to prove our methodology. Users can also provide components that fulfill some requirement of the components already provided by the framework, thus extending the framework.

Fig. 2 shows an example of algorithmic components that could be used to solve the classical Traveling Salesman Problem. The yellow background boxes represent the general-purpose and problem independent components provided by our framework, while the blue background ones are custom components developed by a hypothetical user: a custom algorithm, *Memetic*, using a memetic metaheuristic (Neri et al., 2011); a constructive method, *RouteShuffle*, that shuffles the order in which the destinations are visited; a candidate list manager, *TSPCandList*, for the GRASP constructive procedure (Feo & Resende, 1989); a swap neighborhood, *SwapNeigh*, that changes the order in which two destinations are visited, and a Tabu improvement strategy (Glover & Laguna, 1997), *TSPTabu*. The `requires` and `is a` relationships are denoted with black diamond and white triangle symbols, respectively.

2.1.2. Code analysis and grammar generation

In our proposed methodology, the `requires` and `is a` relationships are discovered by scanning the source code. We rely on both the class hierarchy provided by object-oriented programming languages and source code annotations to describe requirements, domains and other metadata. The details of our implementation in Java are given in Appendix B, where we also provide instructions on how to implement our proposal in other programming languages. The *Scanner* module shown in Fig. 1 finds all available algorithmic components, including those provided by the framework and those implemented by the user. Then, each component and their relationships are analyzed, and a graph similar to Fig. 2 is generated.

As there is no distinction between components provided by our framework and those provided by the user, the user is free to extend any part of the existing component hierarchy or even create a new one. The only component that must always be used or inherited is the *Algorithm* component. See Figure A.10 in Appendix A for a summary of all the components currently available in our framework.

Examining the relationships, the *Scanner* phase generates an implicit grammar as follows: `requires` relationships produce rules that expand to a sequence of one or more symbols without choices, whereas `is a` relationships produce rules that expand to one or more choices of single symbols.

Grammar 1 shows an example. The first grammar rule is the list of available algorithms in the current context, represented by the starting symbol *Algorithm*, with each found algorithm as a derivation. The third rule in the example specifies that the algorithm *Simple* has a dependency on both a *Constructive* and an *Improver* component, such as the *Memetic* component defined by the user, as seen in the next grammar rule. Components are not just labels but also encapsulate behavior, thus an expansion Algorithm→MultiStart→Simple has a different behavior than Algorithm→Simple. *Constructive* can be subsequently replaced with any component that matches its specification: for example, a

[1] https://github.com/mork-optimization/mork

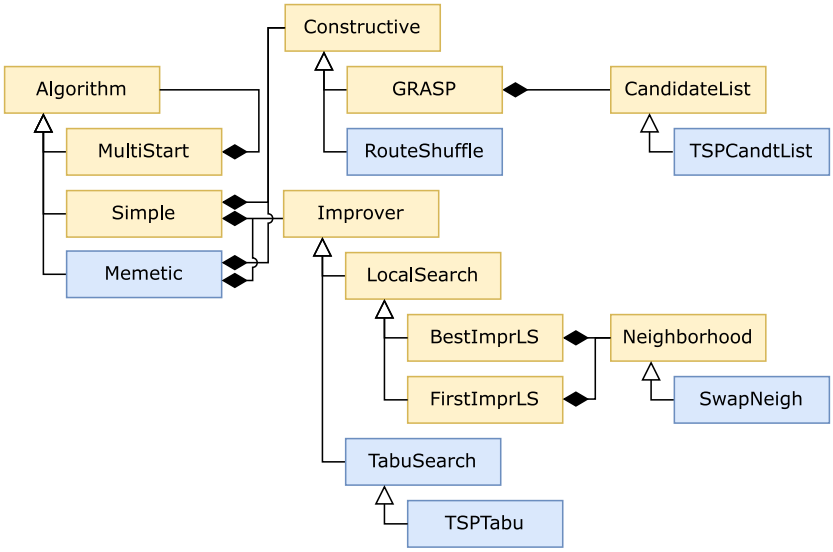


Fig. 2. Example of automatically discovered components and the hierarchy they form. Diamond arrows mean that the component at the side of the diamond is composed by or depends on the component at the other side (`requires` relationship), while the white triangle represents that a given component implements or is of the type pointed by the triangle (`is a` relationship). Parameters of each component are omitted for brevity.

⟨Algorithm⟩ ::= ⟨Simple⟩ | ⟨MultiStart⟩ | ⟨Memetic⟩
⟨MultiStart⟩ ::= ⟨Algorithm⟩
⟨Simple⟩ ::= ⟨Constructive⟩ ⟨Improver⟩
⟨Memetic⟩ ::= ⟨Constructive⟩ ⟨Improver⟩
⟨Constructive⟩ ::= ⟨GRASP⟩ | `RouteShuffle`

Grammar 1: Excerpt of grammar generated from the component relationships seen in Fig. 2.

random shuffle (`RouteShuffle`) of the instance data, or a GRASP with its corresponding candidate list implementation (*GRASP*).

Recursivity is implicitly allowed in our proposal, and occurs in cases when an algorithmic component has a depency on a component type which it can fulfill itself. Fig. 2, and specifically, the second rule of Grammar 1, shows that the *MultiStart* component requires an *Algorithm* component. This is due to the fact that the *MultiStart* algorithm executes multiple times any component of type *Algorithm*. Therefore, it is possible that the chosen component is a *MultiStart* component with a different configuration. While recursive behavior can be explicitly disabled for any component, as will be detailed in Section 2.3, a global recursion limit is always enforced in order to prevent infinite loops, and its value can be customized by the user. Therefore, derivations are applied until all chosen components have their dependencies satisfied, or the current path in the derivation tree reaches the recursion limit and it must be discarded.

Once all components have been analyzed, and the implicit grammar is generated, the next step consists in building the derivation tree. This task is performed by the *Recursive Grammar Walker* (Fig. 1). The derivation tree defines the space of valid algorithmic component combinations that will be explored during the design by optimization phase (phase 2 in Fig. 1). Fig. 3 shows two algorithms that could be generated from the example presented in Fig. 2. In the first one, Fig. 3(a), the *MultiStart* algorithm provided by the framework has been chosen, using the *Simple* component as its internal algorithm. The *Simple* algorithm consists in sequentially building a solution using a constructive method, using an improvement method over the generated solution afterward. Components that represent simple algorithmic parameters (component parameters), like the size of the tabu list and the number of iterations, are represented as green empty boxes. In Fig. 3(b), a more complex example is built with the user-defined *Memetic* algorithm. This example also shows that the framework-provided and user-defined components are intermingled, without any differentiation, which is one of the novel contributions of our approach. If a given component combination does not generate a valid proposal because a component has a dependency that cannot be satisfied, or the maximum depth is reached for a given component combination, the affected branches of the derivation tree are pruned.

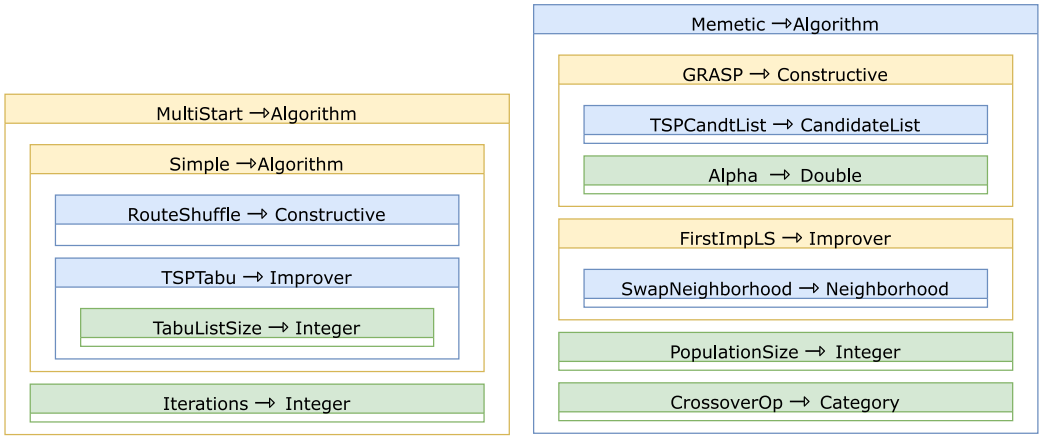
Lastly, the derivation tree is translated into a space of categorical, numerical and conditional parameters (*Parameter Space Transformer* in Fig. 1). The generated parameter space can be optimized by an off-the-shelf AAC method, such as `irace`.[2]

2.2. Parameter space and algorithmic component types

Modern AAC methods are able to handle parameter spaces that contain categorical and numerical parameters, in addition to conditional parameters that are only enabled when other parameters take particular values. We take advantage of this ability to define a number of type annotations that users may apply to algorithmic components. Note that each annotation allows the user to provide specific metadata used when generating the parameter space, such as the minimum and maximum values of a number parameter. In particular, we define six types of annotations, which will map components to the following parameter types:

- *Component* parameter represents any algorithmic component type required as a dependency. By default, any component that matches the given type could be used. For example, a parameter of type *Improver*, could be filled either by any *LocalSearch* component, or any custom *Improver* implemented by the user. An algorithmic component may optionally provide metadata to explicitly declare which components are allowed, or block some

[2] The *Parameter Space Transformer* could generate output in a format suitable for other AAC methods or optimizers, such as Optuna (Akiba et al., 2019), SMAC (Hutter et al., 2011) or ParamILS (Hutter et al., 2009).



(a) Example of a generated multi-start GRASP algorithm.

(b) Example of a generated hybrid evolutionary algorithm, using a memetic approach.

Fig. 3. Examples of automatically generated algorithms. Each component and parameter is represented as a box, which may recursively contain components. Components available in the framework are yellow; blue components are the ones provided by the user, and component parameters (for example, numerical parameters) are green. Note that the component parameters are terminals and can never contain other components themselves. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

specific components, reducing the number of possibilities available by default. For example, users may desire to exclude certain framework-provided components inside a user-implemented component.

- *Context* parameter represents any parameter type whose value is either fixed or calculated at runtime by a user-provided function. Examples can be the algorithm name, or the type of objective function, which is either maximize or minimize.
- *Integer* and *Real* parameters represent an integer value or a real value, respectively, and allow the user to define the range of valid values.
- *Categorical* and *Ordinal* parameters represent a choice from a predefined set of values. An example of a categorical parameter could be choosing either enabling or disabling a component feature using a boolean value. Ordinals may be used in place of categorical parameters when there is an implied order between the parameter possible values.

When using `irace` as the AAC method, the *Parameter Space Transformer* tool maintains the type of categorical, ordinal, integer and real parameters, as those types are directly supported by `irace`. Component parameters are translated as categorical parameters. Conditional parameters control which parameters are active according to the choices made for parameters that appear earlier in the derivation tree. Finally, context parameters are not translated to the parameter space at all. Instead, their value is automatically set by the *Algorithm Builder* (see Fig. 1) when generating an algorithm.

2.3. Automatic design by optimization

The next step employs optimization, in particular an AAC method, to search the generated parameter space for good metaheuristic designs. Each potential configuration of parameter values represents a candidate metaheuristic that can be instantiated from the framework. The quality of a configuration is evaluated by running the generated metaheuristic on a number of (training) problem instances and scoring the result. In this section, we explain the optimization loop in part (2) of Fig. 1 that implements this search, which is formed by four distinct components.

Automatic Configurator is the AAC method that optimizes the parameter space. It generates candidate parameter configurations that the *Algorithm Builder* translates into metaheuristic designs. It also selects problem instances and random seeds to evaluate those configurations by means of the *Executor*. Finally, it receives a score value for each evaluation from the *Results Analyzer*. Using these score values, it generates new candidate configurations to evaluate, until it runs out of the provided evaluation budget.

Algorithm Builder translates each parameter configuration received from the AAC method to an executable algorithm in two steps. First, the parameter configuration received from the AAC method is translated to an intermediate language that describes the corresponding algorithm. An example of a configuration generated by `irace` and its translation to the intermediate language are given in Listings 1 and 2, respectively.

Listing 1: Example parameter configuration generated by `irace` to execute the algorithm presented in Fig. 3(a).

```
--Algorithm=MultiStart --Algorithm_MultiStart.
    ↪ algorithm=Simple --Algorithm_MultiStart
    ↪ .iterations=10 --Algorithm_MultiStart.
    ↪ algorithm_Simple.constructive=
    ↪ RouteShuffle --Algorithm_MultiStart.
    ↪ algorithm_Simple.improver=TSPTabu --
    ↪ Algorithm_MultiStart.algorithm_Simple.
    ↪ improver_TSPTabu.tabuListSize=250
```

Listing 2: Intermediate language representation of the algorithm presented in Fig. 3(a)

```
MultiStart{
  iterations=10,
  algorithm=Simple{
    constructive=RouteShuffle{},
    improver=TSPTabu{
      tabuListSize=250
    }
  }
}
```

Then, the algorithm description is parsed to build an executable version of the requested algorithm. The separation in two steps decouples the parameter description, which is specific to the AAC method, from the intermediate language parser that builds

executable algorithms, which may be used by any AAC method or even invoked manually by the user. In our implementation, building an algorithm requires no compilation and, thus, its overhead is negligible compared to running the algorithm. For a complete technical description of this step, see Appendix B.

Executor launches the executable algorithm generated by the algorithm builder using the chosen instance, in a reproducible environment. Two implementations are provided. The first one executes built algorithms sequentially, while the second one can parallelize a limited set of the runs. All built algorithms have a user-defined maximum runtime, after which their execution is terminated if they did not finish earlier.

Results analyzer collects metrics during the algorithm execution, i.e., the objective value of the best-so-far solution within a single run together with the runtime when it was found.

Typically, AAC methods expect a single number representing the performance of the given parameter configuration on the given problem instance. Examples of performance metrics are the best objective function value found up to a given termination criterion, or the runtime required to reach some decision or solution quality target. Another possible metric is the Area Under the Curve (AUC) of the decreasing step function defined by pairs *(computation time, objective value)* generated by the algorithm when it finds an improved best-so-far solution within a single run. Lower values of the AUC metric indicate faster convergence, that is, better anytime behavior. More details about using this approach for the automated configuration of anytime algorithms can be found in López-Ibáñez and Stützle (2014).

The AAC method is given a computational budget defined by the user, typically measured in total number of algorithm runs or a maximum running time for the whole process. Once the budget is exhausted, the system returns the best algorithm designs found, along with a report summarizing the findings.

3. Target problems

We tested the proposed methodology for the automated design of metaheuristics on three optimization problems from different problem families. In this section, each problem is introduced along with a summary of its state of the art.

3.1. Space-free multi-row facility layout problem

The Space-Free Multi-Row Facility Layout Problem (SF-MRFLP) belongs to the family of Facility Layout Problems. This family encompasses optimization problems devoted to placing facilities using a given layout as the main constraint, trying to optimize magnitudes that are usually related to handling material among different facilities or operation costs (Anjos & Vieira, 2017). Examples are circular or loop layouts, row based layouts and open-field layouts. The SF-MRFLP generalizes the Corridor Allocation Problem (Amaral, 2012), in which only two rows may be used.

The SF-MRFLP consists of placing a predefined number of facilities n in a layout formed by two or more rows, taking into account that no free space is allowed between adjacent facilities. Each facility i has a particular length l_i and a list of weights w_{ij} specifying the material flow to other facilities j. The position of each facility i in its assigned row determines its loading point x_i, which is situated at the center of the facility. The loading point is calculated as the sum of the lengths of all facilities placed in the same row before itself, plus half its length. If the facility i is the first in the row, the loading point is simply $x_i = \frac{l_i}{2}$. The goal is to minimize the *material handling cost*, which is calculated as the sum of the horizontal distances, assuming the distances between rows and the height of the facilities are negligible, between the loading points of each facility pair, multiplied by their respective material flow, given by:

$$\min \quad \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij} |x_i - x_j| \tag{1}$$

As indicated by the problem name, facilities in the same row cannot have free space between them. Moreover, there are two additional restrictions: all rows must be aligned on the left-hand side, and no overlapping between facilities is allowed.

To the best of our knowledge, the state-of-the-art metaheuristic is a variable neighborhood search (VNS) described by Herrán et al. (2021). Its constructive phase is a GRASP using two different strategies for picking elements from the candidate list, *RandomGreedy* and *GreedyRandom*. During the improvement step of the VNS algorithm, three different neighborhoods are used: removing and adding a facility from one position to another, swapping two facilities, and an extended neighborhood formed by both. Furthermore, the proposed neighborhoods can be explored using three different strategies: first improvement, best improvement and hybrid (pick best move in a smaller section of the neighborhood).

3.2. Balanced minimum sum-of-squares clustering problem

The second problem used to validate the proposal is the Balanced Minimum Sum-of-squares Clustering (BMSSC) problem (Xavier & Xavier, 2011). The BMSSC problem is devoted to assigning a given number n of elements (points) located in a d-dimensional space into a given number k of sets (clusters) of equal size. If the number of points n is not divisible by k, there will be $n \bmod k$ clusters of size $\lceil n/k \rceil$ and $k - (n \bmod k)$ clusters of size $\lfloor n/k \rfloor$. The objective function is given by:

$$\min \sum_{i=1}^{k} \sum_{j=1}^{n} \|p_j - c_i\|^2 \cdot a_{ij} \tag{2}$$

which minimizes the sum-of-squares distance of an assignment of points $P = \{p_1, p_2, \ldots, p_n\}$, to the k clusters, taking into account the distance of each point p_j to the centroid c_i of the cluster it is assigned to. The centroid c_i is the point with the minimum Euclidean distance to all points assigned to the same cluster i; and a_{ij} is a binary variable that takes the value of 1 if point j is assigned to cluster i and 0 otherwise.

The state-of-the-art metaheuristic is by Martín-Santamaría et al. (2022), where the authors propose combining a GRASP approach with the Strategic Oscillation method (Glover & Hao, 2011). Two neighborhoods are proposed: the first one swaps elements between clusters, which maintains solution feasibility; while the second one removes an element from its currently assigned cluster and reassigns it to a different one, which is likely to break the equal-size constraint. The authors propose using the first neighborhood during the local search phase of the GRASP metaheuristic and using the second one during the strategic oscillation step, while relaxing the cluster size constraint. Afterward, the solutions may need to be repaired, which is achieved by using the same neighborhood to reassign elements from overloaded clusters to undersized clusters.

3.3. Vehicle routing problem with occasional drivers

The third and final problem belongs to the family of the Vehicle Routing Problems (VRP). The main objective of the VRP is to create delivery routes, in order to fully satisfy consumer demand, while minimizing operating costs. The Vehicle Routing Problem with Occasional Drivers (VRPOD), firstly introduced in Archetti et al. (2016), is a variant of the VRP that considers the possibility of having occasional drivers (ODs) to attend some customers, therefore reducing the length and number of routes, and the associated operating costs. An OD is not a professional driver, but a customer that goes to a physical shop location, and agrees to deliver a package from an online order for a small

compensation if the package destination is in route to their destination. The objective function takes into account both the cost of the vehicle fleet and the sum of compensations to the occasional drivers. Different compensation schemes may be used to pay the drivers, see Archetti et al. (2016) for the full problem formulation.

The current state-of-the-art metaheuristic for the VRPOD is by Martín-Santamaría et al. (2021), which improves the results obtained by Archetti et al. (2016). The authors propose using a cooperative parallel Iterative Local Search, named ILS_M, using five different neighborhoods for the local search step. Three are commonly used in VRP problems, specifically move one package from one route to another one in any position, swapping two deliveries, and reversing a route fragment, also known as 2-Opt. Moreover, two specific neighborhoods for the VRPOD related to the usage of ODs are used: assigning a delivery to an OD, and therefore removing it from its associated route, or the reverse operation, removing an OD and assigning its delivery to any route in any position. For the perturbation step, three different methods are presented: removing a random route, using a probability distribution which depends on the total cost of the route; randomly deassigning a percentage of all deliveries to be later repaired; and randomly applying movements from any neighborhood.

4. Experimental analysis

The state-of-the-art metaheuristics for the three problems considered were originally implemented in different programming languages, evaluated on different hardware and use different parallelization strategies. To enable a fair comparison, special care must be employed, so performance differences cannot be attributed to a mismatch between either hardware or software. The experimentation is divided in two distinct phases. First, the automatic design by optimization, detailed in Section 2, is used to generate elite algorithm designs. Then, the performance of the automatically generated algorithms is compared against the already published state-of-the-art algorithms, which were designed manually using an incremental iterative methodology.

4.1. Experimental methodology

The proposed automatic-design framework is implemented in Java. The original implementations of the state-of-the-art metaheuristics for the VRPOD and BMSSC are also implemented in Java. The original implementations of state-of-the-art algorithm for SF-MRFLP was in C++. Java algorithms were executed using Java 21. The Java Virtual Machine (JVM) heap size was limited to 24 GB. In the case of C++, GCC 11.4 was used to compile the sources, enabling optimizations (`-O3`).

All experiments were executed in the same virtual machine, running on a host using an AMD EPYC 7643 CPU running at 2.3 GHz. The automatic design process used 32 CPUs to run multiple algorithm designs on different instances in parallel, but each of those runs is limited to a single CPU core. The state-of-the-art algorithms for the VRPOD and the SF-MRFLP use parallelization strategies in their original proposal, using multiple CPUs to speed up execution. To ensure a fair comparison against single threaded proposals, the CPU usage of all algorithms was limited to a single CPU core.

All problem instances used in this paper are taken from the works that proposed the current state-of-the-art algorithms. There is a total of 420 instances for the VRPOD, 25 for the BMSSC, and 356 for the SF-MRFLP.

Similarly, all algorithmic components designed in the state-of-the-art problems are included as user-defined components in the framework.

4.1.1. Automated metaheuristic design

The first step of the experimentation is using the automatic design by optimization procedure to generate elite algorithms.

We have chosen `irace` (López-Ibáñez et al., 2016) as the AAC method that searches for good metaheuristic designs because it is open source, well documented, matches all our requirements, and it is widely used for automatic configuration and design (De Souza & Ritt, 2018; Pagnozzi & Stützle, 2019; Rasku et al., 2019). The computational budget given to `irace` scales with the number of parameters. Specifically, the budget is 500 evaluations per parameter to optimize, with a minimum of 10 000, where each evaluation corresponds to running once a candidate algorithm design on one problem instance. In the case of SF-MRFLP, the design space has a total of 38 parameters, so `irace` will execute a maximum of 19 000 evaluations. In the case of the BMSSC and the VRPOD, the number of parameters is 114 and 54, which correspond to 57 000 and 27 000 evaluations, respectively.

We use the AUC (defined above) as the metric minimized by `irace` when comparing configurations, *i.e.*, potential algorithm designs. When computing the AUC, we discard the area outside the interval $[10, 60]$ in seconds. Any algorithm configuration that does not report an objective function value before the first 10 s is considered invalid and removed. All other settings of `irace` are set to their default values. Moreover, in order to highlight the ability of the automatic design methodology to identify good algorithm designs without prior knowledge, we do not provide any initial configuration to `irace` in terms of good algorithm designs or default parameter values. Therefore, each execution of `irace` starts from algorithm designs and parameter values randomly sampled from the algorithm design space.

In order to avoid possible overfitting issues during the automatic metaheuristic design phase, a representative set of instances is selected for each optimization problem, using the methodology proposed in Martín-Santamaría et al. (2023). The size of this training set for the SF-MRFLP, the BMSSC and the VRPOD is 54, 11, and 63, respectively. In the case of the SF-MRFLP and the VRPOD, these numbers correspond with 15% of the size of the instance set, while for the BMSSC, we have used almost 50% due to the reduced size of the original set.

4.1.2. Comparison with the state-of-the-art designs

Once an algorithmic design has been generated for each problem, using the proposed automated design methodology, we compare their performance against the state-of-the-art methods. As mentioned above, the automatically-generated algorithms are implemented in Java, hence, we can compare them directly with the original state-of-the-art implementations for the VRPOD and BMSSC, which were also implemented in Java. In the case of the SF-MRFLP, to compare with the original state-of-the-art implementation in C++, we rewrite the automatically-generated algorithm also in C++ and compile both algorithms with the same compiler and libraries to perform a fair comparison.

For each problem, the automatically generated algorithm ($Auto_{Problem}$) and the state-of-the-art one ($SOTA_{Problem}$) will be executed using all the available instances, repeating the experiment 30 times, changing the random seed in each repetition. A hard limit is imposed on the maximum execution time of each $(algorithm, instance, repetition)$ triplet, using twice the value of the execution time reported in the literature. Instances used while tuning the approaches are removed from this testing set. Full results for the full set of instances are available in Appendix C.

The results for each problem will be summarized in tables using four metrics: number of executions that reach the best-known value of an instance (#Times reaches *bkv*, where *bkv* means best-known value); number of instances for which the algorithm obtains the best-known value at least once (#Instances finds *bkv*); percentage gap between the best value found in all repetitions for a given instance and the best-known value, averaged over all instances (%Gap Min); and lastly, averaged percentage gap of all executions to the best known value (%Gap Avg). Moreover, for each problem, we will generate plots describing the performance profile of both algorithms, following the methodology proposed in López-Ibáñez and Stützle (2014), where the

Table 1
Links to both artifacts archived in Zenodo and source code available publicly in GitHub repositories.

Problem	Source code	Archived artifacts DOI
SF-MRFLP	rmartinsanta/ac-SFMRFLP	10.5281/zenodo.7774832
BMSSC	rmartinsanta/ac-BMSSC	10.5281/zenodo.7774637
VRPOD	rmartinsanta/ac-VRPOD	10.5281/zenodo.7774830

Listing 3: Intermediate language representation of the best algorithm found for the SF-MRFLP problem ($Auto_{SF\text{-}MRFLP}$). For a detailed description of each algorithm component, see Appendix A.1.

```
IteratedGreedy{
  constructive=CAPGRASPConstructive{
    type="greedyB1",
    alpha=0.13
  },
  improver=CAPFirstHybridIncLS{
    neighborhood=ExtendedNeighborhood{}
  },
  destructionReconstruction=CAPShake{
    type="Insert",
    mode="6"
  },
  maxIterations=18236,
  stopIfNotImprovedIn=786175
}
```

Table 2
Comparison between the automatically generated algorithm and the state-of-the-art method for the SF-MRFLP.

	$Auto_{SF\text{-}MRFLP}$	$SOTA_{SF\text{-}MRFLP}$
#Times reaches *bkv*	3049 (33.65%)	2623 (28.95%)
#Instances finds *bkv*	239 (79.14%)	261 (86.42%)
%Gap Min	0.0039	0.0030
%Gap Avg	0.0570	0.0966

Listing 4: Intermediate language representation of the best algorithm found for the BMSSC problem ($Auto_{BMSSC}$). For a detailed description of each algorithm component, see Appendix A.2.

```
VNS{
  shake=StrategicOscillation{
    increment=0.75
  },
  maxK=2,
  improver=FirstImpLS{
    neighborhood=BMSSCSwapMove{}
  },
  constructive=GreedyRandomGRASPConstructive{
    alpha=0.68,
    candidateListManager=BMSSCListManager{}
  }
}
```

evolution of the average objective function gap to the best known value (o.f. gap) is plotted along the execution time considering all studied instances.

In addition, we assess the statistical significance of the observed differences between the state-of-the-art method and the automatically generated algorithms by means of Bayesian statistical analysis (Calvo et al., 2019). In this methodology, the algorithms under comparison are ranked per instance according to a given metric, and the expected probability of each algorithm winning (*i.e.*, returning a better objective value than the other in a single run) is computed. Since the final experiment comparing the automatically generated algorithms and the state-of-the-art ones is repeated 30 times, we compare the results of both algorithms for each instance using the %Gap Avg. value. The uncertainty around the winning probability is given by credible intervals, which are the Bayesian equivalent of confidence intervals. A credible interval estimates the range of the true winning probability with a particular probability.

In the next subsections, we summarize the results obtained for each problem, including descriptions of the generated algorithms. Raw results, solution data and, in general, all artifacts generated during the experimentation, are archived in Zenodo. The complete source code of both the state-of-the-art metaheuristic and the automatically designed metaheuristics is available on GitHub. Table 1 provides links to all this material.

4.2. Results for the SF-MRFLP

The automatic design process took approximately 52 h and generated a design based on the Iterated Greedy metaheuristic, as seen in Listing 3, where the algorithm is represented using the intermediate language described in previous sections. An interesting feature of the proposal is the presence of a shake component instead of the traditional destruction-reconstruction component. In the state-of-the-art paper, a parallel Variable Neighborhood Search method was designed with a cooperative scheme.

Table 2 compares the best algorithm found for this problem by the automated design process ($Auto_{SF\text{-}MRFLP}$) with the state-of-the-art algorithm ($SOTA_{SF\text{-}MRFLP}$). We can observe that $SOTA_{SF\text{-}MRFLP}$ obtains the best known value in 28.95% of runs, while $Auto_{SF\text{-}MRFLP}$ obtains 33.65%. However, the $Auto_{SF\text{-}MRFLP}$ algorithm reaches the best-known value for 239 out of a total of 302 instances in the testing set, while the state-of-the-art algorithm reaches it for 261 instances. This difference in results can be explained by the fact that the $SOTA_{SF\text{-}MRFLP}$ is less stable, and only reaches the *bkv* once or a few times over the 30 repetitions.The gaps show that on average, the $Auto_{SF\text{-}MRFLP}$ obtains better results, while being slightly behind to the original proposal if taking into account the minimum value found over the 30 repetitions.

The results of the Bayesian statistical analysis are shown in Fig. 4, where the dot represents the probability of winning estimated by the Bayesian approach previously described. We can observe in the plot that $Auto_{SF\text{-}MRFLP}$ has a higher probability of winning than the state-of-the-art algorithm. Since the intervals do not overlap, the superiority of the $Auto_{SF\text{-}MRFLP}$ is statistically significant. Finally, in Fig. 5, we can observe the anytime behavior of both algorithms. In very short computing times, the state-of-the-art algorithm may have a slight advantage, but if the execution time is greater than ten seconds, the automatically generated design obtains better results on average. The fact that the $Auto_{SF\text{-}MRFLP}$ proposal is worse in short computing times can be explained because, during the tuning phase, the interval used to calculate the algorithm performance ignores precisely the results obtained during the first 10 seconds.

4.3. Results for the BMSSC

After 50 h, the automatic design procedure generated an algorithm based on the VNS metaheuristic, described in Listing 4. Notably, the algorithm combines VNS with a strategic oscillation method to increase the maximum allowed cluster sizes, increasing the limit a maximum of two times. The state-of-the-art paper proposed a GRASP algorithm also with strategic oscillation.

Table 3 compares the best automatically generated algorithm found for this problem ($Auto_{BMSSC}$), and the state-of-the-art algorithm ($SOTA_{BMSSC}$). We observe that $Auto_{BMSSC}$ reaches the best-known value more often and in more instances than the state-of-the-art algorithm. Moreover, $Auto_{BMSSC}$ also obtains lower minimum and average gaps.

The Bayesian statistical analysis (Fig. 6) estimates that $Auto_{BMSSC}$ has greater winning probability. The credible intervals are wider than

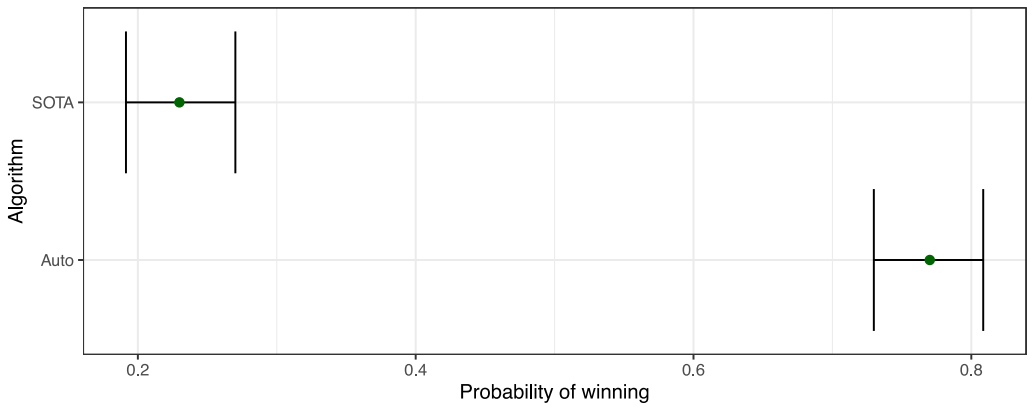


Fig. 4. Expected winning probability for both the SF-MRFLP state-of-the-art metaheuristic and the best automatically generated algorithm found.

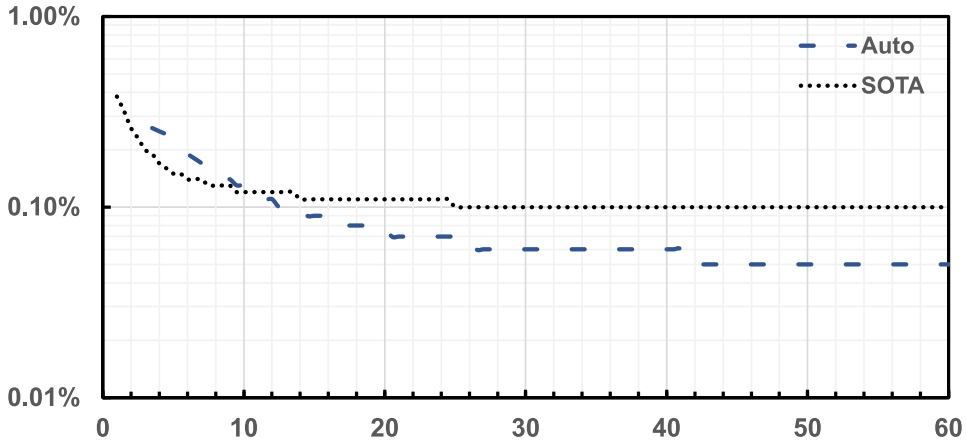


Fig. 5. Performance profile of both the state-of-the-art metaheuristic and the best automatically generated algorithm, for the SF-MRFLP. The x-axis represents the elapsed execution time, in seconds, while the y-axis represents how far away from the best value is the algorithm at any given point in time, using a logarithmic scale.

Table 3
Comparison between the automatically generated algorithm and the state-of-the-art metaheuristic for the BMSSC.

	$Auto_{BMSSC}$	$SOTA_{BMSSC}$
#Times reaches *bkv*	171 (40.71%)	137 (32.62%)
#Instances finds *bkv*	11 (78.57%)	9 (64.29%)
%Gap Min	0.0142	0.2930
%Gap Avg	0.1281	0.4739

in the previous comparison for the SF-MRFLP, which means that the uncertainty is more pronounced. This fact can be explained because $SOTA_{BMSSC}$ obtains better results than $Auto_{BMSSC}$ when certain structures are present in the instance, while the $Auto_{BMSSC}$ approach performs better on average. Finally, Fig. 7 summarizes the performance profile of both algorithms. We can observe that the automatically designed approach converges much faster than the state-of-the-art metaheuristic, having a much better anytime behavior.

4.4. Results for the VRPOD

For the VRPOD, the automatic design procedure required only 17 h for generating a new algorithm based on the Iterated Greedy metaheuristic. The complete algorithm description is shown in Listing 5. The state-of-the-art proposal was an Iterated Local Search with a cooperative scheme.

Table 4 compares the automatically generated algorithm for the VRPOD ($Auto_{VRPOD}$) and the state-of-the-art algorithm ($SOTA_{VRPOD}$). In this case, the $Auto_{VRPOD}$ reaches the best-known value 32.50% of the repetitions, obtaining the best value at least once for 99% of the instances. On average, $Auto_{VRPOD}$ achieves a much lower gap, that is, better solutions, than $SOTA_{VRPOD}$.

The Bayesian statistical analysis, shown in Fig. 8, estimates that $Auto_{VRPOD}$ has a significantly higher winning probability than the state-of-the-art method, with no overlap between the credible intervals.

Listing 5: Intermediate language representation of the best algorithm found for the VRPOD problem ($Auto_{VRPOD}$). For a detailed description of each algorithm component, see Appendix A.3.

```
IteratedGreedy{
  constructive=VRPODGRASPConstructive{
    alpha=0.47
  },
  improver=VND{
    improver1=BestImprovementLS{
      neighborhood=RouteToODNeigh
    },
    improver2=BestImprovementLS{
      neighborhood=ExtendedNeighborhood
    }
  },
  destructionReconstruction=RandomMovement{
    multiplier=36
  },
  maxIterations=771701,
  stopIfNotImprovedIn=683424
}
```

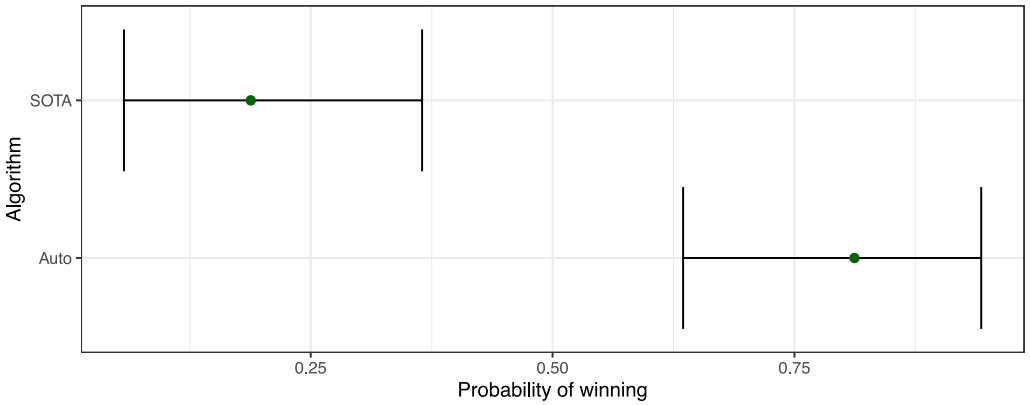


Fig. 6. Expected winning probability for both the BMSSC state-of-the-art metaheuristic and the best automatically generated algorithm found.

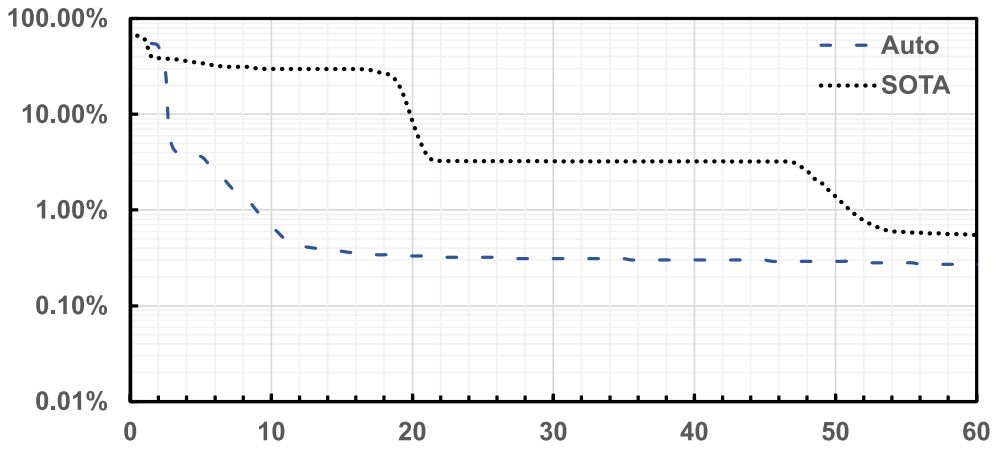


Fig. 7. Performance profile of both the state-of-the-art metaheuristic and the best automatically generated algorithm, for the BMSSC. The x-axis represents the elapsed execution time, in seconds, while the y-axis represents how far away from the best value is the algorithm at any given point in time, using a logarithmic scale.

Table 4
Comparison between the automatically generated algorithm and the state-of-the-art metaheuristic for the VRPOD.

VRPOD	$Auto_{VRPOD}$	$SOTA_{VRPOD}$
#Times reaches *bkv*	3135 (29.27%)	1925 (17.97%)
#Instances finds *bkv*	354 (99.16%)	133 (37.25%)
%Gap Min	0.0023	1.0216
%Gap Avg	1.9638	4.3331

Therefore, $Auto_{VRPOD}$ has a statistically significantly higher chance of finding better solutions than $SOTA_{VRPOD}$. The performance profiles of both algorithms (Fig. 9) show that, on average, they perform similarly, but the automatically designed approach has a slightly better anytime behavior.

5. Conclusions and future work

In this work, we have proposed a methodology to automatically generate metaheuristics for optimization problems. Our proposal overcomes some of the obstacles found in previous automatic design frameworks by following the *design-by-contract* and *open-closed* principles recently advocated in this journal (Swan et al., 2022). Our methodology eliminates the need for an explicit, carefully designed, meta-algorithmic template or grammar that must be manually adjusted when extending the framework. Instead, each algorithmic component is annotated with properties that define its relationship to other algorithmic components. These properties are easily implemented via mechanisms provided by most programming languages. As a result, extending the framework only requires appropriately annotating the new algorithmic components.

We have demonstrated the practical benefit of our proposed methodology by automatically designing metaheuristics for three substantially different combinatorial optimization problems, namely, a facility layout problem, a vehicle routing problem and a clustering problem. We have compared these automatic designs with previous state-of-the-art metaheuristics for each problem, designed by human experts based on intuition and limited experimentation. Our results show that the automatic designs match or improve the performance of the state-of-the-art approaches for all the proposed optimization problems. Although the automatic design process may consume several days of computation, the outcome is a very competitive algorithm. On the contrary, it is more than likely that a researcher will spend weeks on preliminary experimentation in the process of algorithm development, testing and parameter tuning. Therefore, we believe that our proposal would remain useful even if it required weeks of computation time.

In our experiments, the automatic design process begins without any knowledge regarding the state-of-the-art designs, the relative performance of algorithmic components or default values for any algorithmic parameters. An idea worth exploring in future works may be to analyze the effect of starting the design process from known good-performing designs, such as those existing in the state of the art, and measure how they affect the performance and convergence of the AAC method that searches the design space. In addition, it could be interesting to allow the researcher provide a hand-crafted grammar and compare the results against the one generated by our method.

Since the process is based on component composition by means of the automatically generated grammar, the resulting algorithms can be

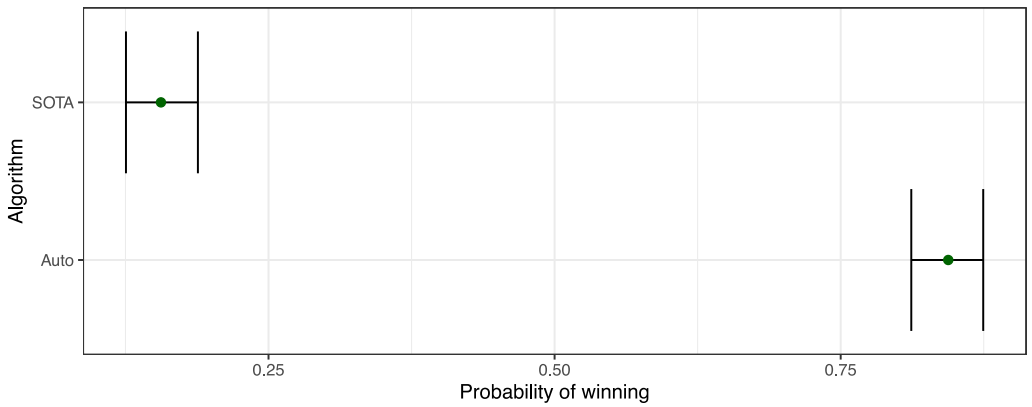


Fig. 8. Expected winning probability for both the VRPOD state-of-the-art metaheuristic and the best automatically generated algorithm found.

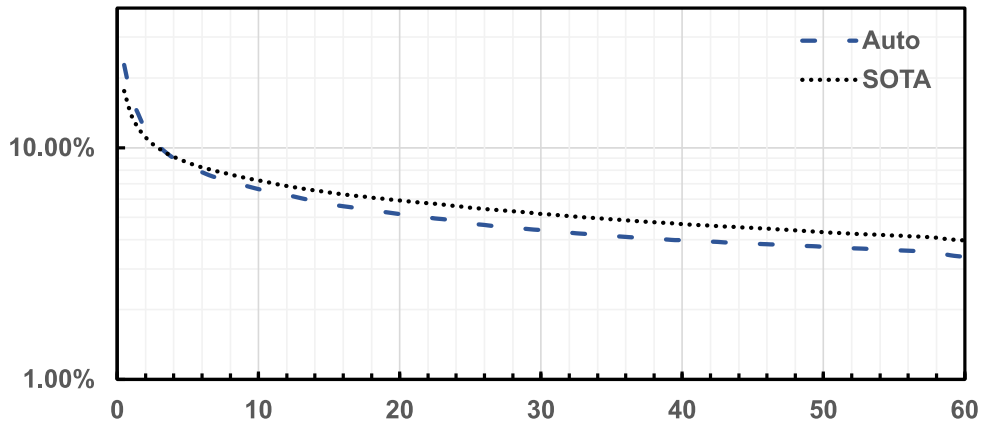


Fig. 9. Performance profile of both the state-of-the-art metaheuristic and the best automatically generated algorithm, for the VRPOD. The x-axis represents the elapsed execution time, in seconds, while the y-axis represents how far away from the best value is the algorithm at any given point in time, using a logarithmic scale.

easily interpretable. For instance, different algorithm designs that are functionally equivalent could be simplified if studied by a human. An example is a *VND* executing the same local search method multiple times, where removing the *VND* component and calling the local search method once should obtain the same results. It is an open research question to what extent such equivalencies could be detected automatically and either reported to the user in order to revise the components provided or used to automatically simplify the generated algorithm designs, so that the automatic design process generates the simplest configurations possible.

Another line of future work consists in extending the framework to incorporate more advanced algorithmic components, such as components that enable parallelization and components from mathematical programming that enable the design of matheuristics. Towards this goal, we make our current implementation of the framework publicly available at https://github.com/mork-optimization.

CRediT authorship contribution statement

Raúl Martín-Santamaría: Conceptualization, Formal analysis, Investigation, Software, Validation, Writing – original draft, Writing – review & editing. **Manuel López-Ibáñez:** Conceptualization, Formal analysis, Methodology, Supervision, Writing – original draft, Writing – review & editing. **Thomas Stützle:** Conceptualization, Formal analysis, Methodology, Supervision, Writing – original draft, Writing – review & editing. **J. Manuel Colmenar:** Conceptualization, Funding acquisition, Investigation, Methodology, Supervision, Validation, Writing – original draft, Writing – review & editing.

Acknowledgments

This work has been partially supported by the Spanish Ministerio de Ciencia e Innovación (MCIN/AEI/10.13039/501100011033) under grant ref. PID2021-126605NB-I00 and by ERDF A way of making Europe; and Generalitat Valenciana with grant ref. CIAICO/2021/224. Thomas Stützle acknowledges support from the Belgian F.R.S.-FNRS of which he is a Research Director.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.ejor.2024.06.001.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In Teredesai , & et al. (Eds.), *25th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 2623–2631). New York, NY: ACM Press, http://dx.doi.org/10.1145/3292500.3330701.

Alfaro-Fernández, P., Ruiz, R., Pagnozzi, F., & Stützle, T. (2020). Automatic algorithm design for hybrid flowshop scheduling problems. *European Journal of Operational Research*, *282*(3), 835–845. http://dx.doi.org/10.1016/j.ejor.2019.10.004.

Amaral, A. R. S. (2012). The corridor allocation problem. *Computers & Operations Research*, *39*(12), 3325–3330. http://dx.doi.org/10.1016/j.cor.2012.04.016.

Anjos, M. F., & Vieira, M. V. C. (2017). Mathematical optimization approaches for facility layout problems: The state-of-the-art and future research directions. *European Journal of Operational Research*, *261*(1), 1–16.

Archetti, C., Savelsbergh, M., & Speranza, M. G. (2016). The vehicle routing problem with occasional drivers. *European Journal of Operational Research*, *254*(2), 472–480. http://dx.doi.org/10.1016/j.ejor.2016.03.049.

Bezerra, L. C. T., López-Ibáñez, M., & Stützle, T. (2016). Automatic component-wise design of multi-objective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, *20*(3), 403–417. http://dx.doi.org/10.1109/TEVC.2015.2474158.

Birattari, M., Stützle, T., Paquete, L., & Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. In W. B. Langdon, & et al. (Eds.), *Proceedings of the genetic and evolutionary computation conference* (pp. 11–18). San Francisco, CA: Morgan Kaufmann Publishers.

Burke, E. K., Gendreau, M., Hyde, M. R., Kendall, G., Ochoa, G., Özcan, E., & Qu, R. (2013). Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, *64*(12), 1695–1724. http://dx.doi.org/10.1057/jors.2013.71.

Burke, E. K., Hyde, M. R., & Kendall, G. (2012). Grammatical evolution of local search heuristics. *IEEE Transactions on Evolutionary Computation*, *16*(7), 406–417. http://dx.doi.org/10.1109/TEVC.2011.2160401.

Cahon, S., Melab, N., & Talbi, E. G. (2004). ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, *10*(3), 357–380. http://dx.doi.org/10.1023/B:HEUR.0000026900.92269.ec.

Calvo, B., Shir, O. M., Ceberio, J., Doerr, C., Wang, H., Bäck, T., & Lozano, J. A. (2019). Bayesian performance analysis for black-box optimization benchmarking. In M. López-Ibáñez, A. Auger, & T. Stützle (Eds.), *Proceedings of the genetic and evolutionary computation conference* (pp. 1789–1797). New York, NY: ACM Press, http://dx.doi.org/10.1145/3319619.

Camacho-Villalón, C. L., Stützle, T., & Dorigo, M. (2021). PSO-X: A component-based framework for the automatic design of particle swarm optimization algorithms. *IEEE Transactions on Evolutionary Computation*, *26*(3), 402–416. http://dx.doi.org/10.1109/TEVC.2021.3102863.

De Souza, M., & Ritt, M. (2018). Automatic grammar-based design of heuristic algorithms for unconstrained binary quadratic programming. In A. Liefooghe, & M. López-Ibáñez (Eds.), *Lecture notes in computer science*: *vol. 10782*, *Proceedings of evoCOP 2018 – 18th European conference on evolutionary computation in combinatorial optimization* (pp. 67–84). Heidelberg, Germany: Springer, http://dx.doi.org/10.1007/978-3-319-77449-7_5.

Dréo, J., Liefooghe, A., Verel, S., Schoenauer, M., Merelo, J. J., Quemy, A., Bouvier, B., & Gmys, J. (2021). Paradiseo: from a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of Paradiseo. In F. Chicano, & K. Krawiec (Eds.), *Proceedings of the genetic and evolutionary computation conference* (pp. 1522–1530). New York, NY: ACM Press, http://dx.doi.org/10.1145/3449726.3463276.

Durillo, J. J., & Nebro, A. J. (2011). jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, *42*(10), 760–771. http://dx.doi.org/10.1016/j.advengsoft.2011.05.014.

Feo, T. A., & Resende, M. G. C. (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, *8*(2), 67–71.

Gendreau, M., & Potvin, J. Y. (Eds.), (2019). *International series in operations research & management science*: *vol. 272*, *Handbook of metaheuristics*. Springer.

Glover, F., & Hao, J. K. (2011). The case for strategic oscillation. *Annals of Operations Research*, *183*(1), 163–173.

Glover, F., & Laguna, M. (1997). *Tabu search*. Boston, MA, USA: Kluwer Academic Publishers.

Hassan, A., & Pillay, N. (2017). A meta-genetic algorithm for hybridizing metaheuristics. In *Progress in artificial intelligence: 18th EPIA conference on artificial intelligence, EPIA 2017, Porto, Portugal, September 5-8, 2017, proceedings 18* (pp. 369–381). Springer.

Hassan, A., & Pillay, N. (2018). An improved meta-genetic algorithm for hybridizing metaheuristics. In *2018 IEEE congress on evolutionary computation* (pp. 1–8). IEEE.

Hassan, A., & Pillay, N. (2019). Hybrid metaheuristics: An automated approach. *Expert Systems with Applications*, *130*, 132–144.

Herrán, A., Colmenar, J. M., & Duarte, A. (2021). An efficient variable neighborhood search for the space-free multi-row facility layout problem. *European Journal of Operational Research*, *295*(3), 893–907. http://dx.doi.org/10.1016/j.ejor.2021.03.027.

Hoos, H. H. (2012). Programming by optimization. *Communications of the ACM*, *55*(2), 70–80. http://dx.doi.org/10.1145/2076450.2076469.

Hoos, H. H., & Stützle, T. (2005). *Stochastic local search—Foundations and applications*. San Francisco, CA: Morgan Kaufmann Publishers.

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In C. A. Coello Coello (Ed.), *Lecture notes in computer science*: *vol. 6683*, *Learning and intelligent optimization, 5th international conference* (pp. 507–523). Heidelberg, Germany: Springer, http://dx.doi.org/10.1007/978-3-642-25566-3_40.

Hutter, F., Hoos, H. H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*, 267–306. http://dx.doi.org/10.1613/jair.2861.

Iturra, S., Contreras-Bolton, C., & Parada, V. (2021). Automatic generation of metaheuristic algorithms. In *International conference on metaheuristics and nature inspired computing* (pp. 48–58). Springer.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., & Leyton-Brown, K. (2009). SATenstein: Automatically building local search SAT solvers from components. In C. Boutilier (Ed.), *Proceedings of the 21st international joint conference on artificial intelligence* (pp. 517–524). Menlo Park, CA: AAAI Press.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., & Leyton-Brown, K. (2016). SATenstein: Automatically building local search SAT Solvers from Components. *Artificial Intelligence*, *232*, 20–42. http://dx.doi.org/10.1016/j.artint.2015.11.002.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., & Birattari, M. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, *3*, 43–58. http://dx.doi.org/10.1016/j.orp.2016.09.002.

López-Ibáñez, M., & Stützle, T. (2012). The automatic design of multi-objective ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation*, *16*(6), 861–875. http://dx.doi.org/10.1109/TEVC.2011.2182651.

López-Ibáñez, M., & Stützle, T. (2014). Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research*, *235*(3), 569–582. http://dx.doi.org/10.1016/j.ejor.2013.10.043.

Martin, R. C. (1996a). The Liskov substitution principle. *C++ Report*, *8*(3), 14.

Martin, R. C. (1996b). The open-closed principle. *More C++ Gems*, *19*(96), 9.

Martín-Santamaría, R., Cavero, S., Herrán, A., Duarte, A., & Colmenar, J. M. (2023). A practical methodology for reproducible experimentation: An application to the double-row facility layout problem. *Evolutionary Computation*, 1–35. http://dx.doi.org/10.1162/evco_a_00317.

Martín-Santamaría, R., López-Sánchez, A. D., Delgado-Jalón, M. L., & Colmenar, J. M. (2021). An efficient algorithm for crowd logistics optimization. *Mathematics*, *9*(5), http://dx.doi.org/10.3390/math9050509.

Martín-Santamaría, R., Sánchez-Oro, J., Pérez-Peló, S., & Duarte, A. (2022). Strategic oscillation for the balanced minimum sum-of-squares clustering problem. *Information Sciences*, *585*, 529–542. http://dx.doi.org/10.1016/j.ins.2021.11.048.

Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J., & Stützle, T. (2013). From grammars to parameters: Automatic iterated greedy design for the permutation flow-shop problem with weighted tardiness. In P. M. Pardalos, & G. Nicosia (Eds.), *Lecture notes in computer science*: *vol. 7997*, *Learning and intelligent optimization, 7th international conference* (pp. 321–334). Heidelberg, Germany: Springer, http://dx.doi.org/10.1007/978-3-642-44973-4_36.

Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J., & Stützle, T. (2014). Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research*, *51*, 190–199. http://dx.doi.org/10.1016/j.cor.2014.05.020.

Meng, W., & Qu, R. (2021). Automated design of search algorithms: Learning on algorithmic components. *Expert Systems with Applications*, *185*, Article 115493.

Nebro, A. J., López-Ibáñez, M., Barba-González, C., & García-Nieto, J. (2019). Automatic configuration of NSGA-II with jMetal and irace. In M. López-Ibáñez, A. Auger, & T. Stützle (Eds.), *Proceedings of the genetic and evolutionary computation conference* (pp. 1374–1381). New York, NY: ACM Press, http://dx.doi.org/10.1145/3319619.3326832.

Neri, F., Cotta, C., & Moscato, P. (Eds.), (2011). Handbook of memetic algorithms. In *Studies in computational intelligence*: *vol. 379*, *Handbook of memetic algorithms*. Springer.

Pagnozzi, F., & Stützle, T. (2019). Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. *European Journal of Operational Research*, *276*, 409–421. http://dx.doi.org/10.1016/j.ejor.2019.01.018.

Pagnozzi, F., & Stützle, T. (2021). Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems with additional constraints. *Operations Research Perspectives*, *8*, Article 100180. http://dx.doi.org/10.1016/j.orp.2021.100180.

Rasku, J., Musliu, N., & Kärkkäinen, T. (2019). On automatic algorithm configuration of vehicle routing problem solvers. *Journal on Vehicle Routing Algorithms*, *2*(1–4), 1–22. http://dx.doi.org/10.1007/s41604-019-00010-9.

Ross, P. (2005). Hyper-heuristics. In E. K. Burke, & G. Kendall (Eds.), *Search methodologies* (pp. 529–556). Boston, MA: Springer, http://dx.doi.org/10.1007/0-387-28356-0_17.

Silva-Muñoz, M., Contreras-Bolton, C., Rey, C., & Parada, V. (2023). Automatic generation of a hybrid algorithm for the maximum independent set problem using genetic programming. *Applied Soft Computing*, Article 110474. http://dx.doi.org/10.1016/j.asoc.2023.110474.

Stützle, T., & López-Ibáñez, M. (2019). Automated design of metaheuristic algorithms. In M. Gendreau, & J. Y. Potvin (Eds.), *International series in operations research & management science*: *vol. 272*, *Handbook of metaheuristics* (pp. 541–579). Springer, http://dx.doi.org/10.1007/978-3-319-91086-4_17.

Swan, J., Adriaensen, S., Brownlee, A. E. I., Hammond, K., Johnson, C. G., Kheiri, A., Krawiec, F., Merelo, J. J., Minku, L. L., Özcan, E., Pappa, G., García-Sánchez, P., Sörensen, K., Voß, S., Wagner, M., & White, D. R. (2022). Metaheuristics "in the large". *European Journal of Operational Research*, *297*(2), 393–406. http://dx.doi.org/10.1016/j.ejor.2021.05.042.

Xavier, A. E., & Xavier, V. L. (2011). Solving the minimum sum-of-squares clustering problem by hyperbolic smoothing and partition into boundary and gravitational regions. *Pattern Recognition*, *44*(1), 70–77. http://dx.doi.org/10.1016/j.patcog.2010.07.004.

Yi, W., Qu, R., Jiao, L., & Niu, B. (2022). Automated design of metaheuristics using reinforcement learning within a novel general search framework. *IEEE Transactions on Evolutionary Computation*.