# Automated Design of Metaheuristic Algorithms: A Survey

**Anonymous authors**
**Paper under double-blind review**

## Abstract

Metaheuristics have gained great success in academia and practice because their search logic can be applied to any problem with available solution representation, solution quality evaluation, and certain notions of locality. Manually designing metaheuristic algorithms for solving a target problem is criticized for being laborious, error-prone, and requiring intensive specialized knowledge. This gives rise to increasing interest in automated design of metaheuristic algorithms. With computing power to fully explore potential design choices, the automated design could reach and even surpass human-level design and could make high-performance algorithms accessible to a much wider range of researchers and practitioners. This paper presents a broad picture of automated design of metaheuristic algorithms, by conducting a survey on the common grounds and representative techniques in terms of design space, design strategies, performance evaluation strategies, and target problems in this field.

## 1 Introduction

Metaheuristic algorithms are stochastic search methods that integrate local improvement with high-level strategies of escaping from local optima (Glover and Kochenberger, 2006). Representatives include genetic algorithm (GA) (Holland, 1973), simulated annealing (SA) (Kirkpatrick et al., 1983), tabu search (Glover, 1989), particle swarm optimization (PSO) (Kennedy and Eberhart, 1995), ant colony optimization (ACO) (Dorigo et al., 1996), and memetic algorithms (Moscato, 1999), etc. In contrast to analytical methods and problem-specific heuristics, metaheuristics can conduct search on any problem with available solution representation, solution quality evaluation, and a certain notion of locality, in which locality denotes the ability to generate neighboring solutions via a heuristically-informed function of one or more incumbent solutions (Swan et al., 2022). Such capability has enabled metaheuristics to gain broad success across various fields.

Usually, human experts are requested to manually tailor algorithms to a target problem to obtain good enough solutions. Although many manually tailored algorithms have been reported to perform remarkably on various problems, manual tailoring suffers from apparent limitations. First, the manual tailoring process could be laborious, which may cost the expert days or weeks conceiving, building up, and verifying the algorithms. Second, manual tailoring could be error-prone due to the high complexity of the target problem and the high degree of freedom in tailoring the algorithm. Third, the manual process is untraceable regarding what motivates certain design decisions, losing insights and principles for future reuse (Swan et al., 2022). Lastly, manual tailoring is unavailable in scenarios with an absence of human experts.

The automated design is a promising alternative to manual tailoring to address the limitations. It leverages today's increasing computing resources to create metaheuristic algorithms to fit for solving a target problem. Herein, the computer (partly) replaces human experts to conceive, build up, and verify design choices. Human experts can be involved but are not necessary during the process. Therefore, the automated design could make high-performance algorithms accessible to a much broader range of researchers and practitioners. This is significant in response to the lack of time and labor resources. Furthermore, by leveraging computing power to fully explore potential design choices, the automated design could be expected to reach or even surpass human-level design. In the long run, it could be a critical tool in the pursuit of autonomous and general artificial intelligence systems.

**Related work:** By automatically tailoring algorithm components, structures, and hyperparameter values, the automated design could find either instantiations/variants of existing algorithms or unseen algorithms with novel structures and component compositions (Stützle and López-Ibáñez, 2019; Qu et al., 2020). In principle, the automated design can be conducted either offline by a target distribution of problem instances or online by the search trajectory (Swan et al., 2022). Most automated design works follow the offline manner, which is significant for scenarios where one can afford a priori computational resources (for design) to subsequently solve many problem instances drawn from the target domain. We distinguish the relationship of automated design to related topics, i.e., automated algorithm selection (Kerschke et al., 2019), automated algorithm configuration (Huang et al., 2020; Schede et al., 2022), adaptive operator selection and hyperheuristics (Burke et al., 2013; Pillay and Qu, 2018), to avoid conceptual confusion:

- Automated algorithm selection (Kerschke et al., 2019) chooses algorithms from a portfolio for solving a specific problem or problem instance. It alleviates the limitations of manual algorithm design by the selection mechanism that allocates suitable algorithms to different problems or problem instances. The output of the selection is one of the existing algorithms, instead of a customized algorithm for the target problem. From this perspective, automated algorithm selection is an independent topic with respect to automated design.

- Automated algorithm configuration appears in two kinds of literature. The first kind is offline tuning or online controlling hyperparameters of an existing algorithm (Eiben et al., 1999; Huang et al., 2020). The output is an instantiation of the existing algorithm. The second kind involves both offline hyperparameter tuning and algorithm component composition via representing components as categorical parameters (Blot et al., 2017; Aydın et al., 2017; Blot et al., 2019; Sae-Dan et al., 2020; Tari et al., 2020a). Despite subjecting to given algorithm templates, the second kind performs the same as a part of automated design, i.e., component composition and parameter configuration. Thus, the second kind of automated algorithm configuration falls into automated design.

- Adaptive operator selection and hyperheuristics use high-level automated methods to select or generate low-level heuristics/metaheuristics (Burke et al., 2019). The high-level methods are usually heuristics/metaheuristics, e.g., genetic programming (GP) (Koza, 1994), as well as machine learning and data mining (e.g., exploratory landscape analysis) methods; the primary goal of hyperheuristics is raising the low-level heuristics' generality (Swan et al., 2017; Burke et al., 2019; Pillay and Qu, 2021). In comparison, methods for automated design include not only GP, but also those from other fields, e.g., from the hyperparameter optimization of automated machine learning (Hutter et al., 2019); goals of automated design include but are not limited to generality (Ye et al., 2022a). In this regard, the hyperheuristics that generate metaheuristic algorithms can be seen as a part of automated design of metaheuristic algorithms.

Many efforts have been devoted to the automated design of metaheuristic algorithms in recent years. There have been several surveys (Kerschke et al., 2019; Huang et al., 2020; Stützle and López-Ibáñez, 2019; Schede et al., 2022) relating to some of the efforts, but a comprehensive compilation is lacking. The surveys in Kerschke et al. (2019); Huang et al. (2020); Schede et al. (2022) are for automated algorithm selection or configuration, the scopes of which differ from or only cover a part of automated design according to the above concept discrimination. The work in Stützle and López-Ibáñez (2019) is for automated design and focuses on the methods of design. However, apart from the methods, different types of design space, ways of algorithm representations, paradigms for searching design choices, and metrics for evaluating the designed algorithms are also essential for conducting automated design. A systematic survey of these essentials is necessary but still lacking.

**Contributions:** This paper presents a broad picture of automated design of metaheuristic algorithms, by conducting a survey on the common grounds and representative techniques in this field. In the survey, we first provide a taxonomy of automated design of metaheuristic algorithms by formalizing the design process into four modules, i.e., design space, design strategies, performance evaluation strategies, and target problems. Unlike related surveys organized from the perspectives of methods for algorithm configuration/design, our taxonomy involves important elements of automated design that have not been involved in related surveys,

such as types of design space, algorithm representations, and applications. Moreover, our taxonomy provides a comprehensive understanding of the four main modules of the automated design process, which would allow interested readers to easily overview existing studies on the modules of interest and make comparisons. Then, we overview the common grounds and representative techniques with regard to the four modules, respectively. We further discuss the strengths, weaknesses, challenges, and usability of these techniques, which would give researchers a comprehensive understanding of the techniques and provide practitioners with guidance on choosing techniques for different algorithm design scenarios. Finally, we list some research trends in the field, which would promote the development of automated design techniques for serving and facilitating metaheuristic algorithms for complicated problem-solving.

The rest of the paper is structured as: Section 2 introduces preliminaries of this survey; Sections 3, 4, 5, and 6 review and discuss the research progress on the four main modules of automated design of metaheuristic algorithms, i.e., design space, design strategies, performance evaluation strategies, and target problems, respectively; Section 7 points out future research directions; finally, Section 8 concludes the paper.

## 2 Preliminaries

### 2.1 Formulation of Metaheuristic Algorithm Design

Given a target problem, through algorithm design, we would like to find an algorithm (or algorithms) to fit for solving the problem (Birattari and Kacprzyk, 2009):

$$\begin{aligned} &\arg\max_{A} \ \mathbb{E}[\mathbb{E}[P(A)|i]|\mathcal{I}], \\ &\text{s.t.} \qquad A \in \mathcal{S}, \end{aligned} \tag{1}$$

where $A$ is the designed algorithm; $\mathcal{S}$ is the design space, from where $A$ can be instantiated; $i \in \mathcal{I}$ is an instance of the target problem domain $\mathcal{I}$; $P : \mathcal{S} \times \mathcal{I} \to \mathbb{R}$ is a performance metric that measures the performance of $A$ in $\mathcal{I}$. The design aims to find an algorithm (or algorithms) with the best expected performance in $\mathcal{I}$.

In reality, the distribution of problem instances in $\mathcal{I}$ is usually unknown; and one cannot exhaust all the instances during the design process. The common practice of settling this is to consider a finite set of instances from $\mathcal{I}$. Consequently, Eq. (1) is reformulated as

$$\begin{aligned} &\arg\max_{A} \ \mathbb{E}[\mathbb{E}[P(A)|i]|I_t], t = 1, 2, \cdots, T \\ &\text{s.t.} \qquad A \in \mathcal{S}, \\ &\qquad\quad\ \ I_t \subseteq \mathcal{I}, \forall t \in \{1, 2, \cdots, T\} \end{aligned} \tag{2}$$

where $I_t$ is the finite set of problem instances that are targeted at time (i.e., iteration[1]) $t$ of the design process. The instances can either be fixed (i.e., $I_1 = I_2 = \cdots = I_T$) or dynamically changed during the design process. The output of solving Eq. (2) is an algorithm (or algorithms) with the best performance on the instances. To avoid the designed algorithms overfitting, the design process should be followed by validation to testify the designed algorithms' generalization to instances from $\mathcal{I} \backslash \{I_1, I_2, \cdots, I_T\}$.

### 2.2 Automated Design of Metaheuristic Algorithms

We abstract the general process of automated design of metaheuristic algorithms into four modules, as shown in Fig. 1. First, the design space collects candidate design choices (e.g., computational primitives, algorithmic operators, and hyperparameters), which regulates what algorithms can be designed in principle. Second, the design strategy provides a principle way to design algorithms by selecting and assembling design choices from the design space. Third, the performance evaluation strategy defines how to measure the performance of the designed algorithms. The measured performance guides the design strategy to search for desired algorithms. Finally, because the design aims to find algorithms to fit for solving a target problem, the

[1]Since solving Equation 2 is black-box, it is often solved in an iterative manner.

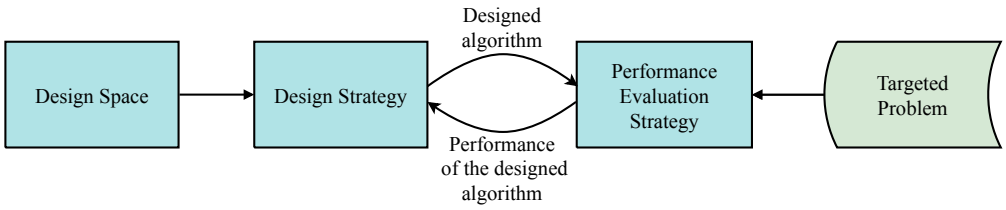


Figure 1: Abstractive process of automated design of metaheuristic algorithms.

target problem acts as external data to support the performance evaluation. The four modules constitute this survey's taxonomy of related research efforts.

### 2.3 Survey Scope

As stated in Section 1, the automated design, in principle, can be conducted either offline with a target distribution of problem instances or online with the search trajectory. Most automated design works follow the offline manner. In contrast, the online manner is often referred to in hyperheuristic or adaptive operator selection literature (related surveys can be found in Burke et al. (2013) and Pillay and Qu (2018)). The scope of this survey is the research efforts on offline automated design.

## 3 Design Space

The design space defines what metaheuristic algorithms can be found in principle. From the perspective of the elements within the design space, the current design space may be classified into two categories, namely, the design space with computational primitives and the design space with existing algorithmic operators. We first overview the research progress on the design space with computational primitives and discuss its strengths, weaknesses, and challenges in subsection 3.1. We then overview the design space with existing algorithmic operators, with a discussion on its strengths, weaknesses, and challenges in subsection 3.2. Finally, we analyze the usability of the two categories and their corresponding algorithm representations at the end of the section.

### 3.1 Design Space with Computational Primitives

Computational primitives define the elementary actions for computation. They include: arithmetic primitives, e.g., $+$, $-$, $*$, $/$; trigonometric primitives, e.g., $sin()$, $cos()$; probabilistic and statistic primitives, e.g., $max()$, $mean()$; operating instructions, e.g., *swap*, *duplicate*; etc. The design space consisting of these primitives enables designing metaheuristic algorithms by choosing primitives from the space, combining the chosen primitives as algorithmic operators, and composing the operators as an algorithm. The ways of representing algorithms over this design space include tree representation and linear array representation in the literature.

**Tree representation with computational primitives:** The binary GP-style tree (Koza, 1994) is the dominant representation for algorithms designed over computational primitives (Poli et al., 2005a;b; Burke et al., 2010; Vázquez-Rodríguez and Ochoa, 2011; Richter and Tauritz, 2018; Richter et al., 2019). As shown in Fig. 2, terminals of the tree are the inputs, including variables, parameters, and constants; non-terminals are the computational primitives. The operation specified by the computational primitive performs by using the values obtained from its child nodes; this process repeats until the root node is reached.

The tree was usually employed to represent a single algorithmic operator instead of an entire algorithm (Poli et al., 2005a;b; Burke et al., 2010; Vázquez-Rodríguez and Ochoa, 2011; Richter and Tauritz, 2018; Richter et al., 2019), because representing an entire algorithm leads to a bloated tree and could be infeasible to find an appropriate tree over the bloated search space (Woodward and Bai, 2009). A given algorithm template is required to map the tree to its phenotype. The template refers to an algorithm structure that manages the execution order and logic of the involved operators. For example, in Poli et al. (2005a) and (Poli et al.,

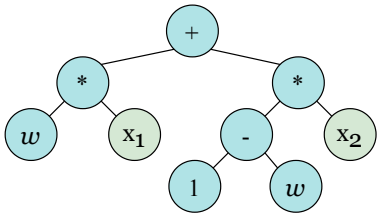


Figure 2: Example of a metaheuristic operator represented by a binary GP-style tree. This example represents the whole arithmetic crossover operator (Eiben et al., 2003), where $x_1$ and $x_2$ are the input variables (i.e., two parent solutions), $w$ is a weight parameter, and $+, *, -$ are primitives. The operator performs $w * x_1 + (1 - w) * x_2$ in mathematical form.

2005b), the particle update operator was represented by a tree and designed; then, the designed operator was inserted into the PSO template (i.e., a recursive process of particle velocity update, particle location update, and best particle update) (Kennedy and Eberhart, 1995). Other examples include representing the mating selection operator of GA (Richter and Tauritz, 2018) and the individual selection operator for mean-update in CMA-ES (Richter et al., 2019).

Due to GP tree's expressiveness in representing programs over the primitives, the tree representation has been widely incorporated into the design space with computational primitives. An open challenge of using the tree representation is avoiding underfitting and overfitting. Current GP-based automated design literature presets the tree's size or structure (Richter et al., 2019; Nguyen et al., 2021), which can be seen as a means of regularization that mitigates over-fitting. More strategies, e.g., model (i.e., tree) selection that determines the optimal amount of model complexity (Hastie et al., 2009), are worth investigating in the automated design field to cope with the challenge. The GP generalization survey in Agapitos et al. (2019) provides a comprehensive analysis of these strategies. Another challenge is that the tree has been limited to represent a single algorithmic operator. The indirect encoding (Eiben and Smith, 2015; Stanley et al., 2019) may alleviate this challenge. Its generative and developmental nature allows learning and reusing building blocks of the tree, which could help improve the search efficiency and scale up the complexity of the algorithm being represented (Eiben and Smith, 2015).

**Linear array representation with computational primitives:** It uses a linear array of computational primitives to represent a metaheuristic operator (Goldman and Tauritz, 2011; Woodward and Swan, 2012). The primitives execute sequentially along with the array; primitives are associated with indexes and parameters to determine which input variable(s) the primitive should execute on. Similar to the tree representation, the linear array was utilized to represent a single algorithmic operator instead of an entire algorithm (Goldman and Tauritz, 2011; Woodward and Swan, 2012) because of the weak expressiveness of the linear array. Related works include representing the crossover (Goldman and Tauritz, 2011) and mutation (Woodward and Swan, 2012) operators of GAs.

A way of augmenting the expressiveness is using the Push language (Spector, 2001) to map the linear array of primitives to an executable algorithm (Lones, 2019; Kamrath et al., 2020; Lones, 2021). Push is a Turing-complete, stack-based typed language used in GP (Spector, 2001). With Push, the primitives are typed, and each primitive executes upon its corresponding type stack (Lones, 2021), rather than executing sequentially along with the array. This stack-based typed system enables the linear array to be more expressive and ensures the represented algorithm to be syntactically valid (Lones, 2021). The linear array representation with Push was used to represent individual-based local search (Lones, 2019; Kamrath et al., 2020) and population-based (Lones, 2021) algorithms.

Overall, using the design space with computational primitives indicates that the design is from scratch with little human bias. Although unbiasedness makes the design space more difficult to search than biased ones, it provides the design space generalization to different target problems. Furthermore, this kind of design space has the potential to produce innovative algorithms that go beyond human experience and surpass the performance of existing algorithms. This potential is appealing and could attract more researchers and

```
N =  initialization, crossover, mutation, selection
T =  number cross, prob reset
S =  <algorithm>
P =  the following production rules:
     <initialization>::=  uniform random | Latin hypercube sample.
          <crossover>::=  one-point | n-point(number cross n).
           <mutation>::=  reset(prob reset).
          <selection>::=  binary tournament | roulette wheel | greedy.
         number cross::=  1 | 2 | 3.
           prob reset::=  0.1 | 0.2 | 0.3.
          <algorithm>::=  <initialization><selection><crossover><mutation><selection> |
                          <initialization><selection><mutation><selection>.
```

Figure 3: Example of grammar for designing genetic algorithms. The production rule `<initialization>` means that the initialization operator can be either `uniform random` or `Latin hypercube sample`. The meaning of other rules is in the same fashion. According to the `<algorithm>` rule, two elitism GA structures can be represented: one searches by crossover and mutation; the other searches by mutation only.

practitioners to study, analyze, interpret, and apply automated design techniques with the advancement of representations and methods for manipulating the representations.

### 3.2 Design Space with Algorithmic Operators

Operators, e.g., the single-point crossover (Eiben et al., 2003), tournament selection (Eiben et al., 2003), greedy randomized construction (Feo and Resende, 1995), and tabu list mechanism (Glover, 1986), are the functional components of an algorithm. They have been used as building blocks to constitute the design space. This design space enables designing metaheuristic algorithms via choosing operators from the space, composing the chosen operators as an algorithm, and configuring endogenous hyperparameters. The ways of representing algorithms over this design space include linear array representation, graph representation, and tree representation in the literature.

**Linear array representation with algorithmic operators:** The linear array representation is the common option in the literature with design space with algorithmic operators. It uses categorical identifiers to index operators and numerals to refer to (conditional) hyperparameter values; an algorithm is subsequently represented by a linear array of the identifiers and numerals (Oltean, 2005; Dioşan and Oltean, 2009; Khudabukhsh et al., 2009; Lopez-Ibanez and Stutzle, 2012; van Rijn et al., 2016). The array is normally with a fixed length for easy manipulation. The common practice of executing the array of operators is predefining an algorithm template that maps the linear array to an executable algorithm. For example, in Blot et al. (2019), a local search template was constructed by four components, i.e., selection, exploration, perturbation, and archive; the template organizes the linear array of operators and parameters into a local search algorithm. in Villalón et al. (2022), the PSO template (Shi and Eberhart, 1998) was adopted to map the linear array to a variant of PSO.

Oltean (2005) and Dioşan and Oltean (2009) provided an alternative to the predefined template, in which each operator was associated with a sequence number; the operators executed according to the sequence. This results in unfolded algorithms with serial executions of operators. The Backus Naur Form grammar (Ryan et al., 1998) was introduced as another alternative to the template (Mascia et al., 2013). It has been widely employed (Tavares and Pereira, 2012; Mascia et al., 2014; Pagnozzi and Stützle, 2019; Miranda and Prudêncio, 2020). As shown in Fig. 3, the grammar is formalized as a tuple $(N, T, S, P)$, where $N$ is the non-terminals (operators); $T$ is the terminals (the algorithm's inputs, e.g., parameters); $S \in N$ is called axiom; $P$ is production rules that manage the operators' execution. This grammar ensures the syntactic validity of the represented algorithm by appropriately setting the production rules (Ryan et al., 1998).

**Graph representation with algorithmic operators:** In contrast to the linear array representation that limits the designed algorithms' structure within the predefined template or production rules, the graph is a well-defined format describing arbitrary orderings of a process, resulting in flexible algorithm structures. Nodes of the graph refer to operators from the design space; directed edges manage the operators' execution flow. Specifically, the directed acyclic graph from Cartesian GP (Miller, 2011) was employed in Kantschik

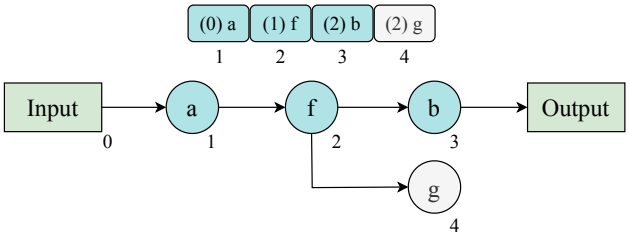


Figure 4: Example of a metaheuristic algorithm represented by a directed acyclic graph. The example is derived from (Ryser-Welch et al., 2016). Nodes $a, f, b, g$ represent different operators; integers $0, 1, 2, 3, 4$ index the nodes. The "(0)a" for node 1 means that node 1 represents operator $a$, and node 1 connects from node 0. The meaning for other nodes is in the same fashion. The exemplified algorithm consists of one input, one output, and three operators $a, f, b$. Operator $g$ in node 4 is inactivated because it is not on the path from the input to the output.

et al. (1999); Shirakawa and Nagao (2009); Ryser-Welch et al. (2016), in which each node is connected from a previous node or the input in a feed-forward manner; the nodes on the path from the input to the output are activated and constitute the represented algorithm. An example is depicted in Fig. 4.

A more flexible directed cyclic graph representation was proposed in Tisdale et al. (2021), in which nodes were allowed to be connected from multiple previous nodes, and edges were associated with weights and orders to determine i) the computational resources allocated to nodes, ii) the inputs and outputs of nodes, and iii) the starting point of the represented algorithm. These features enabled the graph to express various $(\mu/p + \lambda)$ and $(\mu/p, \lambda)$ EA structures.

The graph representation would be promising because of its expressiveness in representing diverse algorithm structures. Related research is still in its infancy. An open challenge is that it is difficult to be manipulated. The challenge is severe for the cyclic graph, because cyclic induces a bloated design space with plenty of meaningless cycles that do not connect the output forward to the beginning of the next algorithm iteration. More studies are desired on the encoding and decoding of graph representation to make it easily manipulated and compatible with different design strategies and performance evaluation strategies.

**Tree representation with algorithmic operators:** The binary GP-style tree (Koza, 1994) was also considered to represent an algorithm over operators (Smorodkina and Tauritz, 2007; Rivers and Tauritz, 2013). Terminals of the tree are the algorithm's inputs. Non-terminals of the tree are the algorithm's operators and (conditional) hyperparameters from the design space. This representation is similar to the tree representation with computational primitives in Fig. 2. The difference is that the non-terminals change from computational primitives to operators.

Overall, the design space with existing algorithmic operators is more compact than that with computational primitives. Furthermore, using this design space allows the design inheriting from prior expert knowledge and experimentation. The compact and knowledge-induced space makes the design over the space possibly easier and accessible with fewer computing resources than that over the space with computational primitives. The weakness is that the space may bias the designed algorithms in favor of human-made ones, thereby reducing the flexibility and innovation potential; besides, users may need to select and collect operators to form the space by themselves, which is a new burden.

### 3.3 Usability

The usability of different kinds of design space and representations varies in different algorithm design scenarios. The design space with computational primitives generally requires a large amount of computing resources to fully explore. This design space would be desired for metaheuristic researchers because of its innovation potential. The design space is not the first choice for practitioners and researchers from other communities because it is not easy for these users to interpret the algorithms derived from computational primitives.

The tree representation is preferred when using this design space due to the tree's strong expressiveness and maturity along with the advancement of GP.

The design space with existing algorithmic operators can be dense, which makes the space accessible to scenarios with limited computing resources. Furthermore, the space can be formed with good design choices from prior algorithms and the user's domain expertise. Therefore, this space is recommended if such prior algorithms or expertise are available. If the target problem is not complicated and does not require a complex algorithm, the linear array representation may be considered since it is easy to be implemented and manipulated; otherwise, the graph representation is advised because of its strong expressiveness. In addition, the graph representation is easy to understand, which is important for users outside the metaheuristic community.

Both the design space with computational primitives and that with algorithmic operators are mixed and conditional, no matter what representation is used. That is, the space is a mix of discrete primitives/operators and continuous hyperparameter values[2]; the chosen operator conditions what hyperparameters will be involved. The mixed and conditional nature requires specific design strategies (detail in Section 4) to handle. Model-free heuristic methods may be required for directly manipulating the mixed and conditional space; otherwise, the space should be projected to another space with a single type of entities (e.g., discretizing continuous hyperparameter values) and align the representations with different conditional hyperparameters to a unified prototype, in order to leverage model-based and learning-based methods for manipulation.

## 4 Design Strategies

The design strategy is a principled way to generate metaheuristic algorithms over the design space. Based on whether there is statistical inference that guides the design process, current design strategies may be classified into two categories, i.e., model-free strategies and model-based strategies. We first overview the model-free design strategies and discuss their strengths, weaknesses, and challenges in subsection 4.2. We then overview the model-based design strategies with a discussion on its strengths, weaknesses, and challenges in subsection 4.1. Finally, we analyze the usability of the two categories at the end of the section.

### 4.1 Model-Free Strategies

Model-free strategies search for desired algorithms in a trial-and-error fashion. They include local search, evolutionary search, and reinforcement learning strategies in the literature.

**Local search:** It mainly refers to the iterative local search (Lourenço et al., 2003) utilized in the ParamILS pipeline (Blot et al., 2019; Tari et al., 2020a; KhudaBukhsh et al., 2009). It is usually incorporated with the design space with algorithmic operators. The key steps are local improvement and global perturbation, in which the former changes one dimension of the algorithm representation; the latter reinitializes the current algorithm once reaching a local optima (Hutter et al., 2009). The local search is often conducted over a dense design space with good choices from prior expert knowledge and experimentation. Therefore, the design strategy itself is often deemphasized.

**Evolutionary search:** Evolutionary search (Eiben et al., 2003) is a more popular design strategy and is widely used in the GP-based algorithm design literature. The algorithm representation determines what specified evolutionary operations can be used. For the tree representation, the sub-tree crossover and mutation are common options (Poli et al., 2005a;b; Richter and Tauritz, 2018; Richter et al., 2019; Smorodkina and Tauritz, 2007; Rivers and Tauritz, 2013). That is, for a pair of tree representations, a node is randomly selected from each tree, respectively, and then, the sub-trees starting at the selected nodes are exchanged (crossover); each sub-tree starting at a randomly selected node is reinitialized (mutation) (Eiben et al., 2003). For the linear array representation, various standard evolutionary crossover and mutation are available (Goldman and Tauritz, 2011; Woodward and Swan, 2012; Oltean, 2005; Dioşan and Oltean, 2009; van Rijn et al., 2016; Ryan et al., 1998; Ye et al., 2022a).

[2]Some hyperparameters may be discrete.

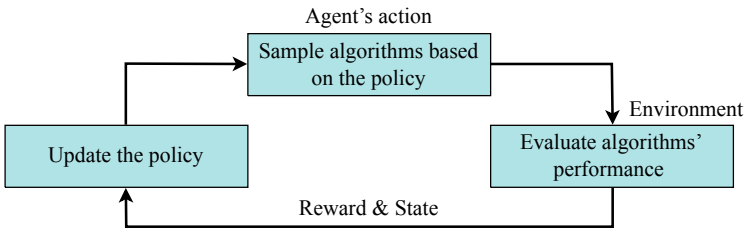


Figure 5: Workflow of reinforcement learning for algorithm design.

Evolutionary search is an adaptive system (Holland, 1962) that provides higher-level adaptation across different target problem instances than local search. The adaption is realized by evolving a population of algorithms fitting across the problem instances. The adaption can be further boosted by evolving from easy to difficult instances (if the difficulty is accountable), in which the knowledge found from easy instances can be anchors to enhance the adaption to difficult instances.

An open challenge behind evolutionary design strategies, as well as local search ones, is the lack of principles for handling the conditional design space, i.e., the chosen algorithmic operator conditions what hyperparameters should be involved. The conditional design space essentially makes the algorithm design task to be bi-level. The upper-level searches for the optimal operator composition, and the nested lower-level tunes the hyperparameters within the operators chosen at the upper level. Principle strategies for managing the bi-level search deserve more investigation.

**Reinforcement learning:** The Markov decision process was introduced to design metaheuristic algorithms in Meng and Qu (2021); Adriaensen and Nowé (2016). It is executed over the design space with algorithmic operators. The state refers to the operators from the design space (Meng and Qu, 2021) or the algorithms constituted by the operators from the design space (Adriaensen and Nowé, 2016). The transition probability between each pair of states is learned according to the constituted algorithms' performance on the target problem instances. Operators or algorithms with better performance have higher transaction probabilities and are more likely to be chosen. The Markov decision process reveals the intrinsic relationships between each pair of design choices and their contributions to the overall performance (Adriaensen and Nowé, 2016).

Tabular reinforcement learning was employed in Meng and Qu (2021); Sakurai et al. (2010); Buzdalova et al. (2014), in which a tabulation records the reward and penalty of each design choice. Modern reinforcement learning approaches, e.g., deep Q-network (Mnih et al., 2015) and proximal policy optimization (Schulman et al., 2017), were introduced recently (Schuchardt et al., 2019; Yi et al., 2022; 2023). The action space is the design space with algorithmic operators. The state refers to the information from evaluating the designed algorithm on certain problem instances, including features of the instances, features of the designed algorithm's solutions to the instances, etc. In particular, the action of (Yi et al., 2022; 2023) is modifying a certain operator of the current algorithm, resulting in algorithms with adaptively changed operators according to the search trajectory (features of solutions in the state). Neural network value functions are equipped to make a long-term planning of design. The general framework of reinforcement learning for metaheuristic algorithm design is given in Fig. 5.

Although reinforcement learning is less widely used than local and evolutionary design strategies, it is promising in automated algorithm design due to three appealing features. The first is the fine-grained design. Reinforcement learning can learn to design at each time step rather than at each episode. This enables building up an algorithm operator-by-operator or primitive-by-primitive, which could be expected to be more efficient than building the whole algorithm at once. The second is the value function. The value function realizes long-term planning of the design with consideration of not only the current target problem instance but also the future ones. The third is the neural network policy model that restores the design knowledge within the neurons, which enables future reuse to related problems or problem domains.

One concern is that the algorithms derived from the policy are often in linear array representations. Current studies limit the algorithms to the sequential execution of the array. It is worth studying encoding-decoding

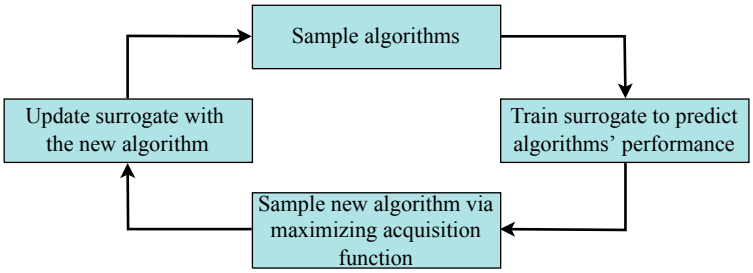


Figure 6: Workflow of Bayesian optimization for algorithm design.

mapping between the representation manipulated by the agent's actions and the phenotype of algorithms to bring innovations in algorithm structures. Another concern is that the reward cannot be calculated in scenarios with fine-grained design from scratch unless all actions have been collected to form a complete algorithm. This arises the sparse reward challenge.

### 4.2 Model-Based Strategies

Model-based strategies use stochastic or statistical models to guide the algorithm design process. They include estimation of distribution and Bayesian optimization in the literature.

**Estimation of distribution:** The "estimation of distribution" was the design strategy in irace (López-Ibáñez et al., 2016) [3]. It is usually incorporated with the design space with algorithmic operators and the linear array algorithm representation. This strategy approximates the distribution of algorithms for the target problem by sampling algorithms over the design space. The distribution is updated by elite algorithms among the sampled ones. in López-Ibáñez et al. (2016), the truncated normal distribution was used for numerical parameters; the multinomial distribution was adopted for categorical parameters/operators; the distribution was partially reinitialized during iteration to introduce exploration.

The "estimation of distribution" design strategy is widely used in the literature (Lopez-Ibanez and Stutzle, 2012; Franzin and Stützle, 2019; Bezerra et al., 2014; 2016; Pagnozzi and Stützle, 2019; Diaz and López-Ibáñez, 2021; Brum et al., 2022) because its fast convergence and the popularity of irace (López-Ibáñez et al., 2016). In its default setting, each entity (an algorithmic operator or hyperparameter) in the array representation was assumed to follow an independent univariate distribution for ease of manipulation. This indicates the orthogonality among the entities, which is usually not the case and may make the design premature. Learning the covariance of distribution of the representation and other advanced methods, e.g., smoothing the update of the distribution (Wierstra et al., 2014), may be investigated to mitigate the premature issue.

**Bayesian optimization:** Bayesian optimization is popular in hyperparameter optimization in automated machine learning (Hutter et al., 2011; 2019; Lindauer et al., 2022), which has been introduced in metaheuristic algorithm design (Tang et al., 2021; Ye et al., 2022a). It is used to search the design space with algorithmic operators and manipulate the linear array representations of the operators. Its core idea is learning a surrogate that models the mapping from algorithms to their performance on the target problem. The sampled algorithm for training the surrogate is obtained by maximizing the acquisition function. The random forest can be adopted as the surrogate because algorithms should be represented as a mix of numeral parameters and categorical operators/parameters, and random forest supports the mixed training data. The expected improvement (Jones et al., 1998) was utilized as the acquisition function (Hutter et al., 2011) due to its closed-form solution assuming that the designed algorithm's performance follows a normal distribution. The workflow is depicted in Fig. 6.

Bayesian optimization obeys a similar iteration process (i.e., algorithm sampling and model update) with estimation of distribution but with two unique features. First, Bayesian optimization leverages a low-

[3]Note that irace did not claim its design strategy as estimation of distribution (López-Ibáñez et al., 2016). We abstract the strategy as an "estimation of distribution" for ease of description.

Table 1: Summary of design strategies.

| Design Strategies | | Key features |
|---|---|---|
| *Model-Free* | Local Search[1,2]<br>Evolutionary Search[1,2,3]<br>Reinforcement Learning[1,4,5] | [1] With little assumption on the algorithm being designed.<br>[2] Flexible to be integrated with various algorithm representations.<br>[3] Exploration over the design space and adaption across problem instances.<br>[4] Reveal intrinsic relationships and contributions of design choices.<br>[5] Fine-grained design, long-term planning, and online design. |
| *Model-Based* | Estimation of Distribution [6,7,8]<br>Bayesian Optimization [6,7,8,9] | [6] Statistical inference provides interpretation.<br>[7] Designed algorithms are subject to predefined structure.<br>[8] Simplified statistics may lead to fast but premature convergence.<br>[9] Surrogate saves computational effort incurred by algorithm evaluation. |

complexity surrogate model to predict the algorithm's performance, which saves computational effort at the cost of potential prediction error. Second, the acquisition function balances the exploration and exploitation in sampling over the design space. The balance is realized by preferring a new sample (algorithm) with high predicted performance (exploitation) and uncertainty (exploration).

Bayesian optimization is acknowledged to perform well on low-dimensional problems. There would be room for improving its scalability to high-dimensional design scenarios. The choice of surrogate models is another open issue. Metaheuristic algorithm design is black-box, in which the landscape of algorithm performance prediction varies when facing different target problems. Apart from the random forest, it is worth having more surrogate choices, e.g., neural networks, to improve the surrogate's accuracy and cope with different design scenarios (regarding target problem types, data richness, computing resources, etc.).

Overall, the main advantage of model-based design strategies is that the statistical models could make the design process interpretable, which will attract more researchers and practitioners to develop and apply the automated design techniques. In this regard, studies on the interpretation of model-based metaheuristic algorithm design are desired. The drawback of the statistical models is that they can only manipulate fixed-length linear array representations of algorithms. The fixed-length array limits the designed algorithm's structures, subsequently weakening the innovation that can be discovered and leading the model-based strategies only to approach a sub-region of the design space.

### 4.3 Usability

In principle, any search method might be employed to search over the design space to produce algorithms. As algorithm design is black-box, there is not yet a consensus and dominant strategy for designing algorithms. The usability of the design strategies depends on the target problem-solving scenarios. If the design space is formed with a few existing algorithmic operators and there is a predefined algorithm template (e.g., designing a variable neighborhood search algorithm from several neighborhood search operators), the local search, estimation of distribution, and Bayesian optimization could be recommended. Among them, using Bayesian optimization as the design strategy is particularly preferred when targeting expensive problems with computational cost dominated by function evaluations, because the low-complexity surrogate model can save function evaluations.

When manipulating the design space with computational primitives, evolutionary search, e.g., GP, would be the first choice. Due to the ability to maintain a population of diverse algorithms, evolutionary search is also suitable for scenarios with demands for algorithm portfolios. Reinforcement learning is appealing for scenarios with a large amount of problems from the same or similar domains, in which the policy model enables the reuse and transfer of the knowledge of prior algorithm design to new problems. It is also appealing for online scenarios. That is, learning the design policy offline and then deploying the policy online to generate algorithms during continual problem-solving. A summary of the design strategies is given in Table 1.

## 5 Performance Evaluation Strategies

The performance evaluation strategy defines how to measure the performance of the designed algorithms. To conduct performance evaluation, one should first choose a performance metric, then evaluate the performance according to the metric and identify desired algorithms by performance comparison. In particular, reducing the time cost of performance evaluation is vital because performance evaluation normally occupies most of the computational cost of the design process. Hence, we first summarize the performance metrics employed in the literature and discuss their usability in subsection 5.1. Then, we review the performance evaluation and comparison strategies in subsection 5.2. Finally, we report the strategies for reducing the time cost of performance evaluation and discuss their strengths and weakness in subsection 5.3.

### 5.1 Performance Metrics

The metrics include the algorithm's solution quality, running time, anytime performance, and jointly considering multiple metrics in the literature.

**Solution quality:** A commonly used performance metric is the algorithm's solution quality within a fixed computational budget. The solution quality is usually a summarizing value based on the objective function value and constraint satisfaction on the target problem. The black-box optimization benchmarking study in Hansen et al. (2022) constructed solution quality metrics for different types of problems, e.g., single-/multi-objective, constrained, and noisy problems.

**Running time:** It measures the algorithm's running time until the algorithm reaches a performance threshold. The threshold can be the solution quality or a particular solution. The metric is formulated as (Ye et al., 2022a):

$$P(A)|i = \min\{r(A, i, \epsilon), b(A, i)\}, \tag{3}$$

where $r(A, i, \epsilon)$ is the running time of algorithm $A$ on target problem instance $i$ till reaching the threshold $\epsilon$, and $b(A, i)$ is the time budget of performance evaluation. The running time often refers to the number of function evaluations. The wall clock time and CPU time may also be considered but may bring about unreproducible results due to the difference in hardware, programming language, and coding style.

**Anytime performance:** It measures the algorithm's ability to return high-quality solutions within any computational budget. The common practice to measure the anytime performance is calculating the area under the curve (AUC) of the empirical cumulative distribution function of running time. Given a set of running time points $\{\tau_u | u \in U\}$ and a set of solution quality thresholds $\{\epsilon_v | v \in V\}$, the AUC is calculated as (Ye et al., 2022a):

$$P(A)|i = \frac{\sum_{u \in U} \sum_{v \in V} \langle s(A, i, \tau_u) \geq \epsilon_v \rangle}{|U| * |V|}, \tag{4}$$

where $s(A, i, \tau_u)$ is algorithm $A$'s solution quality on target problem instance $i$ within $\tau_u$ running time; $\langle \cdot \rangle$ returns 1 if $s(A, i, \tau_u) \geq \epsilon_v$, and return 0 otherwise; $|U|$ and $|V|$ return the numbers of elements in sets $U$ and $V$, respectively. A larger AUC value indicates a better algorithm performance.

An alternative way to measure the anytime performance is by the hypervolume indicator (López-Ibánez and Stützle, 2014). That is, the solution quality and running time can be seen as two conflicting objectives, i.e., better solution quality often demands longer running time. Thus, the elements of an algorithm's performance profile (solution quality versus running time) can be seen as a set of Pareto non-dominate points regarding the two objectives. Therefore, the hypervolume value (Zitzler and Thiele, 1999) of the Pareto non-dominate points with respect to a reference point measures the anytime performance. Higher hypervolume values indicate better anytime performance. This method does not require setting a priori solution quality thresholds.

**Multiple-metric:** Simultaneously considering multiple performance metrics is of interest in certain scenarios. in Blot et al. (2016), the solution quality and memory usage were jointly considered for scenarios with memory limitations. in Blot et al. (2017); Tari et al. (2020b); Bezerra et al. (2020), the solution convergence and solution diversity were considered for designing multi-objective algorithms, i.e., multi-objective design

of multi-objective algorithms; they used the hypervolume (Zitzler and Thiele, 1999) to measure the algorithm's solution convergence and used the $\Delta$ spread to measure the solution diversity (Deb et al., 2002). By considering multiple performance metrics, the designed algorithm would be more generalized to the target problem-solving scenario than only pursuing a single metric.

**Usability of the metrics:** The solution quality has been employed in most literature. The reason is possibe because presetting the computational budget is easy when targeting the widely-used numerical benchmark problems (where much prior experimentation can be referenced for presetting the budget). The running time and AUC provide a performance guarantee that the designed algorithm could reach the predefined performance threshold within the time budget. The guarantee is significant to practical scenarios. Therefore, the running time and AUC are recommended if the threshold can be defined according to domain expertise or preliminary experimentation. Function evaluations should be the first choice for quantifying the time budget if function evaluations dominate the computational cost. The reason is that it is fair and can provide reproducible results regardless of hardware, programming language, and coding style (Hooker, 1995). The anytime performance is more informative than running time since it comprehensively reveals the performance across the algorithm execution (Hansen et al., 2022). It is preferred in scenarios with demands for anytime behavior, e.g., the designed algorithm deployed in scenarios with variable computational budgets.

### 5.2 Performance Evaluation and Comparison

Due to the stochastic nature, different runs (with different seeds) of a metaheuristic algorithm on the same target problem instance result in inconsistent performance. During the algorithm design process, simply evaluating and comparing candidate algorithms by the performance of a single run is not meaningful. Current strategies of performance evaluation and comparison during the algorithm design process are as follows.

**Conducting descriptive statistics on the performance:** It is straightforward to use descriptive statistics, e.g., mean or median, to evaluate and compare candidate algorithms (Ye et al., 2022a; Junior et al., 2020; Vermetten et al., 2022), but they contain issues. When targeting multiple problem instances, the mean requires the performance values on different instances within the same scale, which is hard to achieve. The median is robust to outliers but will ignore the majority of the data, e.g., an algorithm may get unqualified performance in 49% of the evaluations, but the median would not capture this.

**Conducting statistical hypothesis test on the performance:** Hypothesis test provides more statistical evidence of to what extent an algorithm performs significantly better than others. The Friedman test and $t$-test have been employed in the literature (López-Ibáñez et al., 2016). The $t$-test assumes samples (i.e., performance metric values) following a normal distribution, while the Friedman test is nonparametric and may be more preferred. In particular, the $p$-value correction in multiple comparisons (i.e., simultaneously comparing more than two algorithms) was suggested to be abandoned in López-Ibáñez et al. (2016), considering that $p$-value correction makes the test conservative, subsequently lowering the discrimination among algorithms.

### 5.3 Reducing Time Cost of Performance Evaluation

The repetitive performance evaluation leads the algorithm design process to be computationally expensive. A number of strategies have been introduced to reduce the time cost of performance evaluation. The time cost of performance evaluation can be determined by:

$$time_cost = r(C) * N = r(C * N), \tag{5}$$

where $r(C)$ is the average time cost of one performance evaluation, and $N$ is the number of performance evaluations. Reducing the time cost indicates either reducing the number of performance evaluations (i.e., reducing $N$) or estimating the performance without a full evaluation (i.e., reducing $r(C)$).

#### 5.3.1 Reducing the Number of Performance Evaluations

This strategy refers to evaluating on a few target problem instances instead of on all instances. Intensification (Hutter et al., 2009; Blot et al., 2016) is a method following this strategy. It evaluates the candidate

algorithms' performance instance by instance. The evaluation terminates, and the candidate algorithm is discarded if the algorithm's performance is worse than the incumbent[4] on the current instance. Otherwise, the evaluation continues till it has been evaluated in all instances.

Racing (Maron and Moore, 1997) is also a method that evaluates the performance instance by instance (López-Ibáñez et al., 2016). The difference of racing from intensification (Hutter et al., 2009; Blot et al., 2016) is that there is no incumbent, but all candidate algorithms are evaluated together. In detail, the candidate algorithms' performance is evaluated in the same instances; the candidates whose performance is statistically worse than at least another candidate are discarded. The evaluation continues with the survival candidates on the remaining instances until the number of survivors reaches the minimum, the computational budget is exhausted, or all instances have been evaluated.

Another strategy for saving performance evaluations is only evaluating desired algorithms instead of all candidate algorithms. The dual population of gender-based GA (Ansótegui et al., 2009) follows this strategy. It maintains multiple algorithms during the design process. The algorithms are separated into a competitive and non-competitive population. The competitive population is evaluated on the target problem instances and works on guiding the design towards finding promising algorithms. The non-competitive population works on introducing diversity without evaluation. The two populations interact through genetic recombination.

#### 5.3.2 Estimating the Performance without a Full Evaluation

One such strategy is using a low-complexity surrogate model to estimate the performance instead of exact evaluation (run the algorithm on the target problem). The random forest acted as a surrogate in Bartz-Beielstein et al. (2005); Hutter et al. (2011); Wang et al. (2017); Lindauer et al. (2022). Its input is vectorized representations of the candidate algorithms; the output is the estimated performance values. Thus, it is an end-to-end performance estimation method and is convenient for use.

Another strategy for estimating performance is early stopping the performance evaluation before the computational budget is exhausted. Capping (Hutter et al., 2009; De Souza et al., 2022) is a method following this strategy. There are two capping versions in the literature. The first version was proposed in Hutter et al. (2009) for scenarios with running time as the performance metric. It uses the best algorithm's [5] running time as a bound. An algorithm's performance evaluation stops once its running time reaches the bound. Reaching the bound indicates that the algorithm fails to perform better than the best algorithm until the time bound, which means that the algorithm can be discarded.

The second version of capping was proposed in De Souza et al. (2022) and is for scenarios with anytime performance metrics. It also uses a bound to limit the performance evaluation. But the bound is a step function curve that describes previously evaluated algorithms' aggregated performance profile across multiple running time points. For a new algorithm, its performance evaluation stops once its performance profile exceeds the bound at some running time points.

#### 5.3.3 Usability

Intensification (Hutter et al., 2009; Blot et al., 2016) and racing (López-Ibáñez et al., 2016) evaluate performance instance by instance. They are available if there are a relatively large number of instances to discriminate algorithms. While intensification and racing can be compatible with any performance metrics, capping (Hutter et al., 2009; De Souza et al., 2022) is specified to scenarios with running time and anytime performance metrics.

Using a surrogate to partially replace performance evaluations is promising due to its end-to-end nature. However, common surrogates (e.g., random forest) require a fixed-length vectorized representation of the algorithm as input. Subsequently, a predefined algorithm template is needed to avoid unnecessarily lengthy vectors. The template limits the designed algorithms to have the same structure. In this regard, it is worth developing algorithm embedding methods to transform algorithms with various structures into compact

[4]Incumbent is the best algorithm found so far in the design process. Its performance in all instances has been evaluated.

[5]The best algorithm can be either the best one found at the current iteration of the design process (trajectory-preserving capping) or the best one found so far of the design process (aggressive capping) (Hutter et al., 2009).

Table 2: Summary of strategies for reducing time cost of algorithm evaluation.

| Methods | | Main ideas |
|---|---|---|
| *Reducing the number of evaluations* | Intensification | Do not evaluate on the next problem instance if performance is worse than the incumbent. |
| | Racing | Do not evaluate on the next problem instance if performance is statistically worse than at least another algorithm. |
| | Dual population | Only evaluate desired algorithms. |
| *Estimating without a full evaluation* | Surrogate | Use low complexity surrogate to approximate performance. |
| | Hutter et al. (2009)'s Capping | Stop evaluating on the current problem instance once running time meets budget. |
| | De Souza et al. (2022)'s Capping | Stop evaluating on the current problem instance once performance profile exceeds bound. |

representations with the same form. Furthermore, currently employed surrogates are regression models estimating algorithms' performance values, which are not always reasonable. For example, suppose there are two algorithms $\{A_1, A_2\}$ with ground truth performance values $\{0.9, 0.91\}$, where 0.9 and 0.91 stand for $A_1$ and $A_2$'s performance, respectively; and there are two estimations $\{0.89, 0.92\}$ and $\{0.91, 0.9\}$. $\{0.91, 0.9\}$ has a smaller mean square error loss than $\{0.89, 0.92\}$ but results in a wrong performance rank. Considering such cases, surrogates that can directly estimate algorithms' performance rank are advised. A summary of the strategies is given in Table 2.

## 6 Target Problems

This section reviews the problems that have been targeted in the literature. Related software is also reported.

### 6.1 Numerical Benchmark Problems

Many works, especially the early ones, employ numerical benchmarks as target problems. Most of them considered continuous optimization problems. The CEC 2005 real-valued parameter optimization benchmarks (Suganthan et al., 2005) were widely used (Poli et al., 2005a;b; Shirakawa and Nagao, 2009; Miranda and Prudêncio, 2015; Lones, 2019; Bogdanova et al., 2019; Kamrath et al., 2020; Miranda and Prudêncio, 2020; Cruz-Duarte et al., 2020; Lones, 2021; Tian et al., 2023). The CEC 2014 real-valued parameter optimization benchmarks (Liang et al., 2013), the black-box optimization benchmark suite (Hansen et al., 2009), and the comparing continuous optimizers platform used for the GECCO Workshops on Real-Parameter Black-Box Optimization Benchmarking were adopted in Aydın et al. (2017); Villalón et al. (2022); van Rijn et al. (2016); Tisdale et al. (2021), and (Ansótegui et al., 2015; Richter et al., 2019), respectively.

The DTLZ (Deb et al., 2005) and WFG (Huband et al., 2006) test suites acted as target problems in the automated design of multi-objective metaheuristic algorithms (Bezerra et al., 2016; 2020; de Lima and Pozo, 2017). Binary optimization benchmarks were also considered. The $NK$-Landscape benchmarks (Kauffman et al., 1993) were utilized in Goldman and Tauritz (2011); Richter and Tauritz (2018) because the fitness landscape properties can be easily controlled by the $N$ and $K$ parameters. The D-Trap suite (Deb and Goldberg, 1993) was employed in Smorodkina and Tauritz (2007); Goldman and Tauritz (2011); Richter and Tauritz (2018); Rivers and Tauritz (2013).

A common experimental observation of these works is that automated design techniques are able to recur existing algorithms and produce new algorithms that perform better than the existing ones. Due to the expressiveness of various problem landscape features, numerical benchmarks are also important for verifying new automated design techniques and empirically analyzing the techniques, e.g., analyzing the efficiency of different performance metrics (Ye et al., 2022a). A common concern on the numerical benchmarks is that it needs to be clarified to what extent these artificial problems reflect the practice. In this regard, benchmark problems derived from the real world could be considered to stimulate the practicality of automated design techniques.

### 6.2 Practical Problems

Automated design of metaheuristic algorithms has been applied to some practical problem domains, although most of the problems come from experimental simulations. Job shop scheduling (JSS) is a representative practical problem considered in the literature. In Mascia et al. (2013; 2014); Vázquez-Rodríguez and Ochoa (2011); Pagnozzi and Stützle (2019); Bezerra et al. (2014); Franzin and Stützle (2019); Alfaro-Fernández et al. (2020); Sae-Dan et al. (2020); Brum et al. (2022), metaheuristic solvers were automatically designed for JSS with different objectives, e.g., makespan, flowtime, and total tardiness. The bi-objective JSS that simultaneously minimizes makespan and flowtime, the dynamic bi-objective JSS, and the resource-constrained JSS were further considered in Blot et al. (2017; 2019); Nguyen et al. (2014; 2021), respectively.

In Sae-Dan et al. (2020); Tavares and Pereira (2012); Ryser-Welch et al. (2016); Lopez-Ibanez and Stutzle (2012), the automated design was conducted within the ACO template for the travelling salesman problem (TSP). The vehicle routing and nurse rostering problems were adopted as case studies in Qu et al. (2020); Meng and Qu (2021). The propositional satisfiability (SAT) problem with instances from various distributions was targeted in Smorodkina and Tauritz (2007); KhudaBukhsh et al. (2009). The quadratic assignment, bin packing, and imbalanced data classification were considered in Franzin and Stützle (2019); Sae-Dan et al. (2020); Mascia et al. (2014), and (Tari et al., 2020b;a), respectively. The automated design techniques were applied to problems with real-world data in Zhao et al. (2022), in which solvers were automatically designed for the raw material stacking and rack placement problems from the warehouse management department of a biomedical electronics company and were reported to be satisfactory.

The aforementioned works show great potential for automated metaheuristic algorithm design for practical problems. For example, for the JSS problem, new stochastic local search algorithms were designed by the EMILI framework and demonstrated to be superior to the state-of-the-art iterative greedy and local search algorithms when pursuing the makespan, flowtime, and total tardiness objectives (Pagnozzi and Stützle, 2019; Alfaro-Fernández et al., 2020). Multipass heuristics were evolved by GP and outperformed the popular heuristics FIFD, WSPT, WEDD, etc. (Thiruvady et al., 2013; Pinedo, 2012) in resource constraint job scheduling (Nguyen et al., 2021). For the TSP problem, new metaheuristic solvers were discovered by Cartesian GP from small-scale TSP instances and reported to generalize well to much larger instances (Ryser-Welch et al., 2016). For the SAT problem, stochastic local search algorithms were derived from SATenstein and significantly outperformed previous state-of-the-art algorithms on various benchmark distributions (KhudaBukhsh et al., 2009).

### 6.3 Software

Generally, one can choose a design space, design strategy, and performance evaluation strategy from these reviewed in Sections 3, 4, and 5, respectively, to build a concrete pipeline for automated design, following the workflow in Fig. 1. irace (López-Ibáñez et al., 2016), ParamILS (Hutter et al., 2009), and SMAC (Hutter et al., 2011; Lindauer et al., 2022) are established pipelines with open-source code that have been or can be introduced in designing metaheuristic algorithms. They are reported below.

**irace:** irace (López-Ibáñez et al., 2016) is an iterative version of the racing method (Maron and Moore, 1997). irace for metaheuristic algorithm design uses (i) the design space with existing algorithmic operators and the linear array representation, (ii) "estimation of distribution" as the design strategy, and (iii) racing as the performance evaluation strategy. irace has been applied to design metaheuristic algorithms with given templates (e.g., with templates of ACO (Lopez-Ibanez and Stutzle, 2012), artificial bee colony (Aydın et al., 2017), SA (Franzin and Stützle, 2019), PSO (Villalón et al., 2022)) and design general metaheuristics (Bezerra et al., 2014; 2016; Pagnozzi and Stützle, 2019; Alfaro-Fernández et al., 2020; Bezerra et al., 2020; Sae-Dan et al., 2020; Diaz and López-Ibáñez, 2021; Brum et al., 2022). The `R` implementation of irace is available on its website[6]. A `C++` interface of irace is given in the Paradiseo software (Dreo et al., 2021; Aziz-Alaoui et al., 2021) and is available on Github[7].

[6]https://iridia.ulb.ac.be/irace
[7]https://github.com/jdreo/paradiseo

**ParamILS:** ParamILS (Hutter et al., 2009), as its name indicated, uses the iterative local search (Lourenço et al., 2003) for parameter configuration. It was employed in algorithm design by representing algorithmic operators as categorical parameters (Blot et al., 2019; Tari et al., 2020a; KhudaBukhsh et al., 2009). Apart from using existing algorithmic operators as design space and iterative local search as design strategy, ParamILS proposes capping as the performance evaluation strategy. A multi-objective ParamILS was developed in Blot et al. (2016) to jointly consider multiple performance metrics. The `Ruby` implementation of ParamILS is available on its website[8].

**SMAC:** SMAC (Hutter et al., 2011; Lindauer et al., 2022) was originated to hyperparameter optimization in automated machine learning and was then employed to configure metaheuristic algorithms (Tang et al., 2021). Its difference from ParamILS is that it uses Bayesian optimization as the design strategy. The intensification and capping can be the performance evaluation strategy alternatively. The `Python` implementation of SMAC is available on Github[9].

**Sparkle:** Sparkle is a recently released `Python` platform for selecting and configuring algorithms (van der Blom et al., 2022). Its current version uses SMAC as the configurator and could be extended to algorithm design. The platform provides easy use for non-experts and supports benchmarking and ablation analysis. The source code is available on its website[10].

## 7 Research Trends

The development of automated metaheuristic algorithm design continues along a number of research threads. Some of the research trends are discussed in this section.

### 7.1 Design Space

**Constructing design space with less user effort:** The design space is critical since it determines what algorithms can be found in principle. Till now, the design space still requires users to manually construct. Guidelines or strategies for constructing the design space are important. One way to provide such guidelines or strategies may be benchmarking, i.e., collecting general design choices for different types of problems, i.e., continuous, discrete, and permutation, and offering maintainable and extendable platforms for users to use and analyze the design choices on particular problems. Another way is automatically constructing the design space. This way is emerging in the neural architecture search field (Radosavovic et al., 2020), in which the design space was evolved according to the performance of neural networks sampled from it (Ci et al., 2021) or co-evolved along with the architecture search process (Chen et al., 2021). These works pave a way for design space construction.

**Novel representations:** The GP parse tree and linear array are dominant algorithm representations for design space with computational primitives and design space with algorithmic operators, respectively. For the GP tree, it has been limited to represent a single algorithmic operator. The idea of indirect encoding (Eiben and Smith, 2015; Stanley et al., 2019) may break the limitation. Its generative and developmental nature allows learning and reusing building blocks of the tree, which could improve the search efficiency and scale up the complexity of the algorithm being represented (Eiben and Smith, 2015). For the linear array, it requires a given algorithm template, limiting the novelty in algorithm structures. The graph representation (Tisdale et al., 2021) would be promising because of its expressiveness in representing algorithm flows. An open challenge is that the graph is difficult to be directly manipulated. More studies are desired on the encoding and decoding of graph representation to make it easily manipulated and compatible with different design strategies and performance evaluation strategies.

[8]https://www.cs.ubc.ca/labs/algorithms/Projects/ParamILS
[9]https://github.com/automl/SMAC3
[10]https://bitbucket.org/sparkle-ai/sparkle/src/main/

### 7.2 Design Strategies

**Designing diversified algorithms in parallel and distributed manners:** Designing multiple algorithms in parallel facilitates exploration over the design space (Miikkulainen and Forrest, 2021; Ye et al., 2022b). This arises an issue of how to measure the diversity (similarity) of the algorithms. Ideally, the similarity between algorithms is determined by not only the distance between their representations but also their similarity in terms of algorithmic structures and exhibited behaviors (Xu et al., 2016; Nikfarjam et al., 2022; Xiang et al., 2022). Quantifying algorithms' similarities and subsequently identifying unexplored design space deserves more attention. Moreover, the advances of distributed computing platforms (e.g., (Hennessy and Patterson, 2011; 2019)) provide innovative opportunities for scaling automated design in hosting more challenging and expensive problems, which also deserves attention.

**Learning-based design strategies:** Learning-based design strategies, e.g., reinforcement learning, offer appealing features, including fine-grained design, long-term planning of the design with the value function, and knowledge reuse with the policy model. These features promise reinforcement learning to generate algorithms to fit-for-purpose in online or life-long problem-solving scenarios with changing problem characteristics, user requirements, and operating environments (Hart, 2017; Weyns et al., 2021). It is desirable to have more studies on developing advanced policies and value functions for learning-based design strategies.

### 7.3 Performance Evaluation Strategies

**Novel performance comparison methods:** While null hypothesis statistical tests have been the common way of performance comparison during the algorithm design process, recent alternatives have shown great potential in complex scenarios. For example, the Bayesian analysis of (Rojas-Delgado et al., 2022) provided an inference of algorithm ranking and quantification of the uncertainty involved in the ranking. Quantifying the uncertainty is important, considering the stochastic nature of metaheuristic algorithms. The study of (Yan et al., 2022) proposed cumulative distribution functions with an algorithm-independent binary filter condition to address the "cycle ranking" paradox, i.e., inconsistent ranking of algorithms in multiple comparisons. These techniques are worth integrating within the algorithm design process for a more reliable performance comparison.

**Surrogate-based performance estimators:** The automated design would be time costly when targeting expensive problems. The cost can be saved by evaluating on a part of instances or early stopping the evaluation. The former requires adequate instances to discriminate the performance of different algorithms, while the latter is available when using running time or anytime performance metrics. An alternative is to build a low-complexity surrogate model to estimate the performance. This surrogate estimator is end-to-end, which can be integrated with any performance metrics and is independent of the characteristics of the target problem. There are some challenges to building an accurate surrogate. For example, how to embed the algorithms with various structures to the input of the surrogate, how to estimate algorithms' performance rank rather than performance metric values, and how different surrogate models are workable for different design scenarios (regarding sample richness, computing resources, etc.).

### 7.4 Experimental and Theoretical Analysis, Practical Applications

Although many works have reported remarkable efficiency of automated design, it is unclear what design space, design strategy, and performance evaluation strategy contribute to that efficiency. Further experimental and theoretical analysis is required to reveal the impact of different kinds of design space and design strategies on different problems. Besides, a majority of existing works apply automated metaheuristic algorithm design to numerical benchmarks and classical simulated problems, e.g., JSS, TSP, and SAT. It is important to go beyond these numerical and simulated problems and contribute more to real-world problem-solving, e.g., incorporating problem-specific elements with the design process, enabling the designed algorithms to handle constraints and to decouple large-scale problem space. Benchmarking, platforms, and user interfaces are also appreciated to promote practical applications.

### 7.5 Intersection with Related Research Fields

Automated design of metaheuristic algorithms is closely related to automated machine learning and meta-learning (Hutter et al., 2019). Advances from the related fields would fertilize the development of the automated design of metaheuristic algorithms, such as multi-fidelity performance evaluation to speed up the design process, model learning to generate surrogates from previous experience and learn policies for reinforcement learning-based design strategies. On the other hand, the automated design of metaheuristic algorithms presents key techniques for automated machine learning and meta-learning. For example, metaheuristic algorithms could be high-level methods for conducting automated machine learning and meta-learning; the GP tree-based representation with computational primitives paves a potential way for automatically programming machine learning algorithms. The interaction and integration of these fields may be a promising thread for boosting innovations.

## 8 Conclusion

This paper surveyed the automated design of metaheuristic algorithms. In the survey, we have abstracted the concept and proposed a taxonomy of automated design of metaheuristic algorithms by formalizing the design process into four parts, i.e., design space, design strategies, performance evaluation strategies, and target problems. We have then reviewed the advances regarding the four parts, respectively. The strengths, weakness, challenges, and usability of these works have been discussed. Finally, some of the research trends have been pointed out. Hopefully, this survey can provide a comprehensive and meticulous understanding of the automated design of metaheuristic algorithms, inspire interest in leveraging automated design techniques to make high-performance algorithms accessible to a broader range of researchers and practitioners, and boost automated design innovations to fuel the pursuit of autonomous and general artificial intelligence.

## References

Steven Adriaensen and Ann Nowé. 2016. Towards a white box approach to automated algorithm design. In *International Joint Conferences on Artificial Intelligence*. IJCAI, New York, NY, USA, 554–560.

Alexandros Agapitos, Roisin Loughran, Miguel Nicolau, Simon Lucas, Michael O'Neill, and Anthony Brabazon. 2019. A survey of statistical machine learning elements in genetic programming. *IEEE Transactions on Evolutionary Computation* 23, 6 (2019), 1029–1048.

Pedro Alfaro-Fernández, Rubén Ruiz, Federico Pagnozzi, and Thomas Stützle. 2020. Automatic algorithm design for hybrid flowshop scheduling problems. *European Journal of Operational Research* 282, 3 (May 2020), 835–845.

Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney. 2015. Model-based genetic algorithms for algorithm configuration. In *International Joint Conference on Artificial Intelligence*. IJCAI, Buenos Aires, Argentina, 733–739.

Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In *International Conference on Principles and Practice of Constraint Programming*. Springer, Lisbon, Portugal, 142–157.

Doğan Aydın, Gürcan Yavuz, and Thomas Stützle. 2017. ABC-X: A generalized, automatically configurable artificial bee colony framework. *Swarm Intelligence* 11, 1 (Feb. 2017), 1–38.

Amine Aziz-Alaoui, Carola Doerr, and Johann Dreo. 2021. Towards large scale automated algorithm design by integrating modular benchmarking frameworks. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Lille, France, 1365–1374.

Thomas Bartz-Beielstein, Christian WG Lasarczyk, and Mike Preuß. 2005. Sequential parameter optimization. In *2005 IEEE Congress on Evolutionary Computation*, Vol. 1. IEEE, Edinburgh, UK, 773–780.

Leonardo CT Bezerra, Manuel López-Ibánez, and Thomas Stützle. 2014. Automatic design of evolutionary algorithms for multi-objective combinatorial optimization. In *International Conference on Parallel Problem Solving from Nature*. Springer, 508–517.

Leonardo CT Bezerra, Manuel López-Ibánez, and Thomas Stützle. 2016. Automatic component-wise design of multiobjective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 20, 3 (Jun. 2016), 403–417.

Leonardo CT Bezerra, Manuel López-Ibáñez, and Thomas Stützle. 2020. Automatically designing state-of-the-art multi-and many-objective evolutionary algorithms. *Evolutionary Computation* 28, 2 (Jun. 2020), 195–226.

Mauro Birattari and Janusz Kacprzyk. 2009. *Tuning metaheuristics: A machine learning perspective*. Springer, New York, NY, USA.

Aymeric Blot, Holger H Hoos, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Heike Trautmann. 2016. MO-ParamILS: A multi-objective automatic algorithm configuration framework. In *International Conference on Learning and Intelligent Optimization*. Springer, Ischia, Italy, 32–47.

Aymeric Blot, Marie-Éléonore Kessaci, Laetitia Jourdan, and Holger H Hoos. 2019. Automatic configuration of multi-objective local search algorithms for permutation problems. *Evolutionary Computation* 27, 1 (Mar. 2019), 147–171.

Aymeric Blot, Alexis Pernet, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Holger H Hoos. 2017. Automatically configuring multi-objective local search using multi-objective optimisation. In *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, Münster, Germany, 61–76.

Anna Bogdanova, Jair Pereira Junior, and Claus Aranha. 2019. Franken-swarm: Grammatical evolution for the automatic generation of swarm-like meta-heuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Prague, Czech Republic, 411–412.

Artur Brum, Rubén Ruiz, and Marcus Ritt. 2022. Automatic generation of iterated greedy algorithms for the non-permutation flow shop scheduling problem with total completion time minimization. *Computers & Industrial Engineering* 163 (Jan. 2022), 107843.

Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society* 64, 12 (2013), 1695–1724.

Edmund K Burke, Matthew Hyde, Graham Kendall, and John Woodward. 2010. A genetic programming hyper-heuristic approach for evolving 2-D strip packing heuristics. *IEEE Transactions on Evolutionary Computation* 14, 6 (2010), 942–958.

Edmund K Burke, Matthew R Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward. 2019. A classification of hyper-heuristic approaches: revisited. In *Handbook of Metaheuristics*. Springer, 453–477.

Arina Buzdalova, Vladislav Kononov, and Maxim Buzdalov. 2014. Selecting evolutionary operators using reinforcement learning: Initial explorations. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 1033–1036.

Minghao Chen, Kan Wu, Bolin Ni, Houwen Peng, Bei Liu, Jianlong Fu, Hongyang Chao, and Haibin Ling. 2021. Searching the search space of vision transformer. *Advances in Neural Information Processing Systems* 34 (2021), 8714–8726.

Yuanzheng Ci, Chen Lin, Ming Sun, Boyu Chen, Hongwen Zhang, and Wanli Ouyang. 2021. Evolving search space for neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. Montreal, Canada, 6659–6669.

Jorge M Cruz-Duarte, Ivan Amaya, José Carlos Ortiz-Bayliss, Santiago Enrique Conant-Pablos, and Hugo Terashima-Marín. 2020. A primary study on hyper-heuristics to customise metaheuristics for continuous optimisation. In *2020 IEEE Congress on Evolutionary Computation*. IEEE, Glasgow, UK, 1–8.

Ricardo Henrique Remes de Lima and Aurora Trinidad Ramirez Pozo. 2017. A study on auto-configuration of multi-objective particle swarm optimization algorithm. In *2017 IEEE Congress on Evolutionary Computation*. IEEE, Donostia, Spain, 718–725.

Marcelo De Souza, Marcus Ritt, and Manuel López-Ibáñez. 2022. Capping methods for the automatic configuration of optimization algorithms. *Computers & Operations Research* 139 (Mar. 2022), 105615.

Kalyanmoy Deb and David E Goldberg. 1993. Analyzing deception in trap functions. In *Foundations of Genetic Algorithms*. Vol. 2. Elsevier, 93–108.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (Apr. 2002), 182–197.

Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. 2005. Scalable test problems for evolutionary multiobjective optimization. In *Evolutionary Multiobjective Optimization*. Springer, New York, NY, USA, 105–145.

Juan Esteban Diaz and Manuel López-Ibáñez. 2021. Incorporating decision-maker's preferences into the automatic configuration of bi-objective optimisation algorithms. *European Journal of Operational Research* 289, 3 (Mar. 2021), 1209–1222.

Laura Dioşan and Mihai Oltean. 2009. Evolutionary design of evolutionary algorithms. *Genetic Programming and Evolvable Machines* 10, 3 (Mar. 2009), 263–306.

Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. 1996. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26, 1 (1996), 29–41.

Johann Dreo, Arnaud Liefooghe, Sébastien Verel, Marc Schoenauer, Juan J Merelo, Alexandre Quemy, Benjamin Bouvier, and Jan Gmys. 2021. Paradiseo: from a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of Paradiseo. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Lille, France, 1522–1530.

Ágoston E Eiben, Robert Hinterding, and Zbigniew Michalewicz. 1999. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 3, 2 (1999), 124–141.

Agoston E Eiben and Jim Smith. 2015. From evolutionary computation to the evolution of things. *Nature* 521, 7553 (May 2015), 476–482.

Agoston E Eiben, James E Smith, et al. 2003. *Introduction to evolutionary computing*. Vol. 53. Springer, New York, NY, USA.

Thomas A Feo and Mauricio GC Resende. 1995. Greedy randomized adaptive search procedures. *Journal of Global Optimization* 6, 2 (Mar. 1995), 109–133.

Alberto Franzin and Thomas Stützle. 2019. Revisiting simulated annealing: A component-based analysis. *Computers & Operations Research* 104 (Apr. 2019), 191–206.

Fred Glover. 1986. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research* 13, 5 (1986), 533–549.

Fred Glover. 1989. Tabu search—part I. *ORSA Journal on Computing* 1, 3 (Aug. 1989), 190–206.

Fred W Glover and Gary A Kochenberger. 2006. *Handbook of metaheuristics*. Vol. 57. Springer Science & Business Media.

Brian W Goldman and Daniel R Tauritz. 2011. Self-configuring crossover. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. Dublin, Ireland, 575–582.

Nikolaus Hansen, Anne Auger, Dimo Brockhoff, and Tea Tušar. 2022. Anytime performance assessment in blackbox optimization benchmarking. *IEEE Transactions on Evolutionary Computation* 26, 6 (Dec. 2022), 1293–1305.

Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. 2009. *Real-parameter black-box optimization benchmarking 2009: Noiseless function definitions*. Technical Report. INRIA.

Emma Hart. 2017. Towards lifelong learning in optimisation algorithms. In *Proceedings of the 9th International Joint Conference on Computational Intelligence*. Madeira, Portugal, 7–9.

Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer.

John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

John H Holland. 1962. Outline for a logical theory of adaptive systems. *J. ACM* 9, 3 (1962), 297–314.

John H Holland. 1973. Genetic algorithms and the optimal allocation of trials. *SIAM J. Comput.* 2, 2 (Jun. 1973), 88–105.

John N Hooker. 1995. Testing heuristics: We have it all wrong. *Journal of Heuristics* 1, 1 (1995), 33–42.

C. Huang, Y. Li, and X. Yao. 2020. A survey of automatic parameter tuning methods for metaheuristics. *IEEE Transactions on Evolutionary Computation* 24, 2 (Apr. 2020), 201–216.

Simon Huband, Philip Hingston, Luigi Barone, and Lyndon While. 2006. A review of multiobjective test problems and a scalable test problem toolkit. *IEEE Transactions on Evolutionary Computation* 10, 5 (Oct. 2006), 477–506.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, Rome, Italy, 507–523.

Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36 (Oct. 2009), 267–306.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated machine learning: Methods, systems, challenges*. Springer Nature, New York, NY, USA.

Donald R Jones, Matthias Schonlau, and William J Welch. 1998. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization* 13, 4 (Dec. 1998), 455–492.

Jair Pereira Junior, Claus Aranha, and Tetsuya Sakurai. 2020. A training difficulty schedule for effective search of meta-heuristic design. In *2020 IEEE Congress on Evolutionary Computation*. IEEE, Glasgow, UK, 1–8.

Nathaniel R Kamrath, Aaron Scott Pope, and Daniel R Tauritz. 2020. The automated design of local optimizers for memetic algorithms employing supportive coevolution. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. Cancún, Mexico, 1889–1897.

Wolfgang Kantschik, Peter Dittrich, Markus Brameier, and Wolfgang Banzhaf. 1999. Meta-evolution in graph GP. In *European Conference on Genetic Programming*. Springer, Göteborg, Sweden, 15–28.

Stuart A Kauffman et al. 1993. *The origins of order: Self-organization and selection in evolution*. Oxford University Press, Oxford, UK.

James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*. IEEE, Perth, WA, Australia, 1942–1948.

P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann. 2019. Automated algorithm selection: Survey and perspectives. *Evolutionary Computation* 21, 1 (Mar. 2019), 1–47.

Ashiqur R KhudaBukhsh, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. 2009. SATenstein: Automatically building local search SAT solvers from components. In *International Joint Conference on Artificial Intelligence*. Pasadena, CA, USA, 517–524.

Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (May 1983), 671–680.

John R Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing* 4, 2 (Jun. 1994), 87–112.

Jing J Liang, Bo Y Qu, and Ponnuthurai N Suganthan. 2013. Problem definitions and evaluation criteria for the CEC 2014 special session and competition on single objective real-parameter numerical optimization. *Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou China and Technical Report, Nanyang Technological University, Singapore* 635 (2013), 490.

Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A versatile Bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research* 23, 54 (Feb. 2022), 1–9.

Michael A Lones. 2019. Instruction-level design of local optimisers using push GP. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Prague, Czech Republic, 1487–1494.

Michael A Lones. 2021. Evolving continuous optimisers from scratch. *Genetic Programming and Evolvable Machines* 22, 4 (Oct. 2021), 395–428.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3 (Sept. 2016), 43–58.

Manuel Lopez-Ibanez and Thomas Stutzle. 2012. The automatic design of multiobjective ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation* 16, 6 (Dec. 2012), 861–875.

Manuel López-Ibánez and Thomas Stützle. 2014. Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research* 235, 3 (2014), 569–582.

Helena R Lourenço, Olivier C Martin, and Thomas Stützle. 2003. Iterated local search. In *Handbook of Metaheuristics*. Springer, New York, NY, USA, 320–353.

Oden Maron and Andrew W Moore. 1997. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review* 11, 1 (Feb. 1997), 193–225.

Franco Mascia, Manuel López-Ibánez, Jérémie Dubois-Lacoste, and Thomas Stützle. 2013. From grammars to parameters: Automatic iterated greedy design for the permutation flow-shop problem with weighted tardiness. In *International Conference on Learning and Intelligent Optimization*. Springer, Catania, Italy, 321–334.

Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle. 2014. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research* 51 (Nov. 2014), 190–199.

Weiyao Meng and Rong Qu. 2021. Automated design of search algorithms: Learning on algorithmic components. *Expert Systems with Applications* 185 (Dec. 2021), 115493.

Risto Miikkulainen and Stephanie Forrest. 2021. A biological perspective on evolutionary computation. *Nature Machine Intelligence* 3, 1 (Jan. 2021), 9–15.

Julian F Miller. 2011. Cartesian genetic programming. In *Cartesian Genetic Programming*. Springer, New York, NY, USA, 17–34.

Péricles Barbosa Miranda and Ricardo Bastos Prudêncio. 2015. Gefpso: A framework for pso optimization based on grammatical evolution. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. Madrid, Spain, 1087–1094.

Péricles BC Miranda and Ricardo BC Prudêncio. 2020. A novel context-free grammar for the generation of pso algorithms. *Natural Computing* 19, 3 (2020), 495–513.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

Pablo Moscato. 1999. Memetic algorithms: A short introduction. In *New Ideas in Optimization*. McGraw-Hill, 219–234.

Su Nguyen, Dhananjay Thiruvady, Mengjie Zhang, and Damminda Alahakoon. 2021. Automated design of multipass heuristics for resource-constrained job scheduling with self-competitive genetic programming. *IEEE transactions on cybernetics* 52, 9 (2021), 8603–8616.

Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. 2014. Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming. *IEEE Transactions on Evolutionary Computation* 18, 2 (Apr. 2014), 193–208.

Adel Nikfarjam, Aneta Neumann, and Frank Neumann. 2022. Evolutionary diversity optimisation for the traveling thief problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 749–756.

Mihai Oltean. 2005. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation* 13, 3 (Sept. 2005), 387–410.

Federico Pagnozzi and Thomas Stützle. 2019. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. *European Journal of Operational Research* 276, 2 (Jul. 2019), 409–421.

Nelishia Pillay and Rong Qu. 2018. *Hyper-heuristics: Theory and applications*. Springer, New York, NY, USA.

Nelishia Pillay and Rong Qu. 2021. Assessing hyper-heuristic performance. *Journal of the Operational Research Society* 72, 11 (2021), 2503–2516.

Michael L Pinedo. 2012. *Scheduling*. Vol. 29. Springer.

Riccardo Poli, Cecilia Di Chio, and William B Langdon. 2005a. Exploring extended particle swarms: A genetic programming approach. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. Washington DC, USA, 169–176.

Riccardo Poli, William B Langdon, and Owen Holland. 2005b. Extending particle swarm optimisation via genetic programming. In *European Conference on Genetic Programming*. Springer, Lausanne, Switzerland, 291–300.

R. Qu, G. Kendall, and N. Pillay. 2020. The general combinatorial optimization problem: Towards automated algorithm design. *IEEE Computational Intelligence Magazine* 15, 2 (May 2020), 14–23.

Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. 2020. Designing network design spaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Seattle, WA, USA, 10428–10436.

Samuel N Richter, Michael G Schoen, and Daniel R Tauritz. 2019. Evolving mean-update selection methods for CMA-ES. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Prague, Czech Republic, 1513–1517.

Samuel N Richter and Daniel R Tauritz. 2018. The automated design of probabilistic selection methods for evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Kyoto, Japan, 1545–1552.

Rebecca Rivers and Daniel R Tauritz. 2013. Evolving black-box search algorithms employing genetic programming. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*. Amsterdam, The Netherlands, 1497–1504.

Jairo Rojas-Delgado, Josu Ceberio, Borja Calvo, and Jose A Lozano. 2022. Bayesian Performance Analysis for Algorithm Ranking Comparison. *IEEE Transactions on Evolutionary Computation* to be published, DOI: 10.1109/TEVC.2022.3208110 (2022).

Conor Ryan, John James Collins, and Michael O Neill. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*. Springer, Paris, France, 83–96.

Patricia Ryser-Welch, Julian F Miller, Jerry Swan, and Martin A Trefzer. 2016. Iterative Cartesian genetic programming: Creating general algorithms for solving travelling salesman problems. In *European Conference on Genetic Programming*. Springer, Porto, Portugal, 294–310.

Weerapan Sae-Dan, Marie-Eléonore Kessaci, Nadarajen Veerapen, and Laetitia Jourdan. 2020. Time-dependent automatic parameter configuration of a local search algorithm. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. Cancún, Mexico, 1898–1905.

Yoshitaka Sakurai, Kouhei Takada, Takashi Kawabe, and Setsuo Tsuruta. 2010. A method to control parameters of evolutionary algorithms by using reinforcement learning. In *2010 Sixth International Conference on Signal-Image Technology and Internet Based Systems*. IEEE, 74–79.

Elias Schede, Jasmin Brandt, Alexander Tornede, Marcel Wever, Viktor Bengs, Eyke Hüllermeier, and Kevin Tierney. 2022. A survey of methods for automated algorithm configuration. *Journal of Artificial Intelligence Research* 75 (2022), 425–487.

Jan Schuchardt, Vladimir Golkov, and Daniel Cremers. 2019. Learning to evolve. *arXiv preprint arXiv:1905.03389* (2019).

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

Yuhui Shi and Russell Eberhart. 1998. A modified particle swarm optimizer. In *IEEE International Conference on Evolutionary Computation*. IEEE, Anchorage, AK, USA, 69–73.

Shinichi Shirakawa and Tomoharu Nagao. 2009. Evolution of search algorithms using graph structured program evolution. In *European Conference on Genetic Programming*. Springer, T'ubingen, Germany, 109–120.

Ekaterina Smorodkina and Daniel Tauritz. 2007. Toward automating EA configuration: The parent selection stage. In *2007 IEEE Congress on Evolutionary Computation*. IEEE, Singapore, 63–70.

Lee Spector. 2001. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference*. San Francisco, CA, USA.

Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. 2019. Designing neural networks through neuroevolution. *Nature Machine Intelligence* 1, 1 (2019), 24–35.

Thomas Stützle and Manuel López-Ibáñez. 2019. Automated design of metaheuristic algorithms. In *Handbook of Metaheuristics*. Springer, New York, NY, USA, 541–579.

Ponnuthurai N Suganthan, Nikolaus Hansen, Jing J Liang, Kalyanmoy Deb, Ying-Ping Chen, Anne Auger, and Santosh Tiwari. 2005. Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. *KanGAL Report* 2005005, 2005 (2005), 2005.

Jerry Swan, Steven Adriaensen, Alexander EI Brownlee, Kevin Hammond, Colin G Johnson, Ahmed Kheiri, Faustyna Krawiec, Juan Julián Merelo, Leandro L Minku, Ender Özcan, et al. 2022. Metaheuristics "in the large". *European Journal of Operational Research* 297, 2 (2022), 393–406.

Jerry Swan, Patrick De Causmaecker, Simon Martin, and Ender Özcan. 2017. A re-characterization of hyper-heuristics. In *Recent Developments of Metaheuristics*. Springer, 75–89.

Ke Tang, Shengcai Liu, Peng Yang, and Xin Yao. 2021. Few-shots parallel algorithm portfolio construction via co-evolution. *IEEE Transactions on Evolutionary Computation* 25, 3 (Jun. 2021), 595–607.

Sara Tari, Holger Hoos, Julie Jacques, Marie-Eléonore Kessaci, and Laetitia Jourdan. 2020a. Automatic configuration of a multi-objective local search for imbalanced classification. In *International Conference on Parallel Problem Solving from Nature*. Springer, Leiden, The Netherlands, 65–77.

Sara Tari, Nicolas Szczepanski, Lucien Mousin, Julie Jacques, Marie-Eléonore Kessaci, and Laetitia Jourdan. 2020b. Multi-objective automatic algorithm configuration for the classification problem of imbalanced data. In *2020 IEEE Congress on Evolutionary Computation*. IEEE, Glasgow, UK, 1–8.

Jorge Tavares and Francisco B Pereira. 2012. Automatic design of ant algorithms with grammatical evolution. In *European Conference on Genetic Programming*. Málaga, Spain, 206–217.

Dhananjay Thiruvady, Gaurav Singh, Andreas T Ernst, and Bernd Meyer. 2013. Constraint-based ACO for a shared resource constrained scheduling problem. *International Journal of Production Economics* 141, 1 (2013), 230–242.

Ye Tian, Xingyi Zhang, Cheng He, Kay Chen Tan, and Yaochu Jin. 2023. Principled design of translation, scale, and rotation invariant variation operators for metaheuristics. *Chinese Journal of Electronics* 32, 1 (2023), 111–129.

Braden Tisdale, Deacon Seals, Aaron Scott Pope, and Daniel R Tauritz. 2021. Directing evolution: The automated design of evolutionary pathways using directed graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Lille, France, 732–740.

Koen van der Blom, Holger H Hoos, Chuan Luo, and Jeroen G Rook. 2022. Sparkle: Towards Accessible Meta-Algorithmics for Improving the State of the Art in Solving Challenging Problems. *IEEE Transactions on Evolutionary Computation* 26, 6 (2022), 1351–1364.

Sander van Rijn, Hao Wang, Matthijs van Leeuwen, and Thomas Bäck. 2016. Evolving the structure of evolution strategies. In *2016 IEEE Symposium Series on Computational Intelligence*. IEEE, Athens, Greece, 1–8.

José Antonio Vázquez-Rodríguez and Gabriela Ochoa. 2011. On the automatic discovery of variants of the NEH procedure for flow shop scheduling using genetic programming. *Journal of the Operational Research Society* 62, 2 (2011), 381–396.

Diederick Vermetten, Hao Wang, Manuel López-Ibañez, Carola Doerr, and Thomas Bäck. 2022. Analyzing the impact of undersampling on the benchmarking and configuration of evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 867–875.

CL Camacho Villalón, Marco Dorigo, and Thomas Stützle. 2022. PSO-X: A component-based framework for the automatic design of particle swarm optimization algorithms. *IEEE Transactions on Evolutionary Computation* 26, 2 (Jun. 2022), 402–416.

Hao Wang, Bas van Stein, Michael Emmerich, and Thomas Back. 2017. A new acquisition function for Bayesian optimization based on the moment-generating function. In *2017 IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 507–512.

Danny Weyns, Thomas Bäck, Renè Vidal, Xin Yao, and Ahmed Nabil Belbachir. 2021. Lifelong computing. *arXiv preprint arXiv:2108.08802* (2021).

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. 2014. Natural evolution strategies. *The Journal of Machine Learning Research* 15, 1 (2014), 949–980.

John R Woodward and Ruibin Bai. 2009. Why evolution is not a good paradigm for program induction: A critique of genetic programming. In *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation*. Shanghai, China, 593–600.

John R Woodward and Jerry Swan. 2012. The automatic generation of mutation operators for genetic algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*. Philadelphia, PA, USA, 67–74.

Yi Xiang, Han Huang, Yuren Zhou, Sizhe Li, Chuan Luo, Qingwei Lin, Miqing Li, and Xiaowei Yang. 2022. Search-based diverse sampling from real-world software product lines. In *44th IEEE/ACM International Conference on Software Engineering*. Pittsburgh, PA, USA, 1945–1957.

Lin Xu, Ashiqur R KhudaBukhsh, Holger H Hoos, and Kevin Leyton-Brown. 2016. Quantifying the similarity of algorithm configurations. In *International Conference on Learning and Intelligent Optimization*. Springer, Ischia, Italy, 203–217.

Yuan Yan, Qunfeng Liu, and Yun Li. 2022. Paradox-free analysis for comparing the performance of optimization algorithms. *IEEE Transactions on Evolutionary Computation* to be published, DOI: 10.1109/TEVC.2022.3199647 (2022).

Furong Ye, Carola Doerr, Hao Wang, and Thomas Bäck. 2022a. Automated configuration of genetic algorithms by tuning for anytime performance. *IEEE Transactions on Evolutionary Computation* to be published, DOI: 10.1109/TEVC.2022.3159087 (2022).

Furong Ye, Diederick L Vermetten, Carola Doerr, and Thomas Bäck. 2022b. Non-elitist selection among survivor configurations can improve the performance of irace. *arXiv preprint arXiv:2203.09227* (2022).

Wenjie Yi, Rong Qu, and Licheng Jiao. 2023. Automated algorithm design using proximal policy optimisation with identified features. *Expert Systems with Applications* 216 (2023), 119461.

Wenjie Yi, Rong Qu, Licheng Jiao, and Ben Niu. 2022. Automated design of metaheuristics using reinforcement learning within a novel general search framework. *IEEE Transactions on Evolutionary Computation* 27, 4 (2022), 1072–1084.

Qi Zhao, Bai Yan, and Yuhui Shi. 2022. AutoOpt: A general framework for automatically designing metaheuristic optimization algorithms with diverse structures. *arXiv preprint arXiv:2204.00998* (2022).

Eckart Zitzler and Lothar Thiele. 1999. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation* 3, 4 (Nov. 1999), 257–271.