Operations Research Perspectives 3 (2016) 43–58

Contents lists available at ScienceDirect

Operations Research Perspectives

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

The irace package: Iterated racing for automatic algorithm configuration

Manuel López-Ibáñez[a,*], Jérémie Dubois-Lacoste[b], Leslie Pérez Cáceres[b], Mauro Birattari[b], Thomas Stützle[b]

[a] Alliance Manchester Business School, University of Manchester, UK
[b] IRIDIA, Université Libre de Bruxelles (ULB), CP 194/6, Av. F. Roosevelt 50 – B-1050 Brussels, Belgium

ARTICLE INFO

Article history:
Available online 21 September 2016

Keywords:
Automatic algorithm configuration
Racing
Parameter tuning

ABSTRACT

Modern optimization algorithms typically require the setting of a large number of parameters to optimize their performance. The immediate goal of automatic algorithm configuration is to find, automatically, the best parameter settings of an optimizer. Ultimately, automatic algorithm configuration has the potential to lead to new design paradigms for optimization software. The irace package is a software package that implements a number of automatic configuration procedures. In particular, it offers *iterated racing* procedures, which have been used successfully to automatically configure various state-of-the-art algorithms. The iterated racing procedures implemented in irace include the iterated F-race algorithm and several extensions and improvements over it. In this paper, we describe the rationale underlying the iterated racing procedures and introduce a number of recent extensions. Among these, we introduce a restart mechanism to avoid premature convergence, the use of truncated sampling distributions to handle correctly parameter bounds, and an elitist racing procedure for ensuring that the best configurations returned are also those evaluated in the highest number of training instances. We experimentally evaluate the most recent version of irace and demonstrate with a number of example applications the use and potential of irace, in particular, and automatic algorithm configuration, in general.

1. Introduction

Many algorithms for solving optimization problems involve a large number of design choices and algorithm-specific parameters that need to be carefully set to reach their best performance. This is the case for many types of algorithms ranging from exact methods, such as branch-and-bound and the techniques implemented in modern integer programming solvers, to heuristic methods, such as local search or metaheuristics. Maximizing the performance of these algorithms may involve the proper setting of tens to hundreds of parameters [42,44,59,89]. Even if default parameter settings for the algorithms are available, these have often been determined with other problems or application contexts in mind. Hence, when facing a particular problem, for example, the daily routing of delivery trucks, a non-default, problem-specific setting of algorithm parameters can result in a much higher-performing optimization algorithm.

For many years, the design and parameter tuning of optimization algorithms has been done in an ad-hoc fashion. Typically, the algorithm developer first chooses a few parameter configurations, that is, complete assignments of values to parameters, and executes experiments for testing them; next, she examines the results and decides whether to test different configurations, to modify the algorithm or to stop the process. Although this *manual* tuning approach is better than no tuning at all, and it has led to high-performing algorithms, it also has a number of disadvantages: (i) it is time-intensive in terms of human effort; (ii) it is often guided by personal experience and intuition and, therefore, biased and not reproducible; (iii) algorithms are typically tested only on a rather limited set of instances; (iv) few design alternatives and parameter settings are explored; and (v) often the same instances that are used during the design and parameter tuning phase are also used for evaluating the final algorithm, leading to a biased assessment of performance.

Because of these disadvantages, this ad-hoc, manual process has been sidelined by increasingly automatized and principled meth-

* Corresponding author.
E-mail addresses: manuel.lopez-ibanez@manchester.ac.uk (M. López-Ibáñez), jeremie.dubois-lacoste@ulb.ac.be (J. Dubois-Lacoste), leslie.perez.caceres@ulb.ac.be (L. Pérez Cáceres), mbiro@ulb.ac.be (M. Birattari), stuetzle@ulb.ac.be (T. Stützle).

http://dx.doi.org/10.1016/j.orp.2016.09.002

ods for algorithm development. The methods used in this context include experimental design techniques [2,29], racing approaches [20], and algorithmic methods for parameter configuration, such as heuristic search techniques [3,10,41,73,81], and statistical modeling approaches [11,43]. These methods have led to an increasing automatization of the algorithm design and parameter setting process.

Automatic algorithm configuration can be described, from a machine learning perspective, as the problem of finding good parameter settings for solving unseen problem instances by learning on a set of training problem instances [19]. Thus, there are two clearly delimited phases. In a primary tuning phase, an algorithm configuration is chosen, given a set of training instances representative of a particular problem. In a secondary production (or testing) phase, the chosen algorithm configuration is used to solve unseen instances of the same problem. The goal in automatic algorithm configuration is to find, during the tuning phase, an algorithm configuration that minimizes some cost measure over the set of instances that will be seen during the production phase. In other words, the final goal is that the configuration of the algorithm found during the tuning phase generalizes to similar but unseen instances. The tuning phase may also use automatic configuration methods repeatedly while engineering an algorithm [71]. Due to the separation between a tuning and a production phase, automatic algorithm configuration is also known as *offline* parameter tuning to differentiate it from *online* approaches that adapt or control parameter settings while solving an instance [13,50]. Nevertheless, online approaches also contain parameters that need to be defined offline, for example, which and how parameters are adapted at run-time; such parameters and design choices can be configured by an offline tuning method [59].

In our research on making the algorithm configuration process more *automatic*, we have focused on racing approaches. Birattari et al. [19,20] proposed an automatic configuration approach, F-Race, based on *racing* [64] and Friedman's non-parametric two-way analysis of variance by ranks. This proposal was later improved by sampling configurations from the parameter space, and refining the sampling distribution by means of repeated applications of F-Race. The resulting automatic configuration approach was called Iterated F-race (I/F-Race) [10,21]. Although a formal description of the I/F-Race procedure is given in those publications, an implementation was not made publicly available. The irace package implements a general *iterated racing* procedure, which includes I/F-Race as a special case. It also implements several extensions already described by Birattari [19], such as the use of the paired *t*-test instead of Friedman's test. Finally, irace incorporates several improvements not published before, such as sampling from a truncated normal distribution, a parallel implementation, a restart strategy that avoids premature convergence, and an elitist racing procedure to ensure that the best parameter configurations found are also evaluated on the highest number of training instances.

The paper is structured as follows. Section 2 introduces the algorithm configuration problem and gives an overview of approaches to automatic algorithm configuration. Section 3 describes the iterated racing procedure as implemented in the irace package as well as several further extensions including the elitist irace. Section 4 illustrates the steps followed to apply irace to two configuration scenarios and compares experimentally the elitist and non-elitist variants. In Section 5, we give an overview of articles that have used irace for configuration tasks and we conclude in Section 6. For completeness, we include in Appendix A a brief description of the irace package itself, its components and its main options.

2. Automatic configuration

2.1. Configurable algorithms

Many algorithms for computationally hard optimization problems are configurable, that is, they have a number of parameters that may be set by the user and affect their results. As an example, evolutionary algorithms (EAs) [36] often require the user to specify settings such as the mutation rate, the recombination operator and the population size. Another example is CPLEX [45], a mixed-integer programming solver, that has dozens of configurable parameters that affect the optimization process, for instance, different branching strategies. The reason these parameters are configurable is that there is no single optimal setting for every possible application of the algorithm and, in fact, the optimal setting of these parameters depends on the problem being tackled [2,19,42].

There are three main classes of algorithm parameters: *categorical*, *numerical* and *ordinal* parameters. Categorical parameters represent discrete values without any implicit order or sensible distance measure. An example is the different recombination operators in EAs. Ordinal parameters are seemingly categorical parameters but with an implicit order of their values, e.g., a parameter with three values {`low`, `medium`, `high`}. Numerical parameters, such as the population size and the mutation rate in EAs, have an explicit order of their values. In addition, parameters may be *subordinate* or *conditional* to other parameters, that is, they are only relevant for particular values of other parameters. For example, an evolutionary algorithm may have a parameter that defines the selection operator as either `roulette_wheel` or `tournament`. The value `roulette_wheel` does not have any specific additional parameters, whereas the value `tournament` requires to specify the value of parameter `tournament_size`. In this case, the parameter `tournament_size` is conditional to the fact that the selection operator takes the value `tournament`. Conditional parameters are not the same as constraints on the values of parameters. For example, given parameters a and b, a constraint may be that $a < b$. Such constraints limit the range of values that a certain parameter can take in dependence of other parameters, whereas conditional parameters either are disabled or they have a value from a predefined range.

2.2. The algorithm configuration problem

We briefly introduce the algorithm configuration problem here. A more formal definition is given by Birattari [19]. Let us assume that we have a parameterized algorithm with N^{param} parameters, X_d, $d = 1, \ldots, N^{\text{param}}$, and each of them may take different values (settings). A configuration of the algorithm $\theta = \{x_1, \ldots, x_{N^{\text{param}}}\}$ is a unique assignment of values to parameters, and Θ denotes the possibly infinite set of all configurations of the algorithm.

When considering a problem to be solved by this parameterized algorithm, the set of possible instances of the problem may be seen as a random variable $\mathcal{I}$, i.e., a set with an associated probability distribution, from which instances to be solved are sampled. A concrete example of a problem would be the Euclidean symmetric traveling salesman problem (TSP), where each problem instance is a complete graph, each node in the graph corresponds to a point within a square of some dimensions and the distance between the nodes corresponds to the Euclidean distance between their associated points. If the points are randomly and uniformly generated, this class of instances, called RUE, is frequently used in the evaluation of algorithms for the TSP [48,49]. In principle, the set of RUE instances is infinite and all instances are equally interesting, thus $\mathcal{I}$ would be an infinite set of equally probable RUE instances. In practice, however, each instance may be generated by a concrete pseudorandom instance generator and we are only interested in a particular range of dimensions and number of points, which can be seen as the random variable $\mathcal{I}$ having a particular non-uniform probability distribution, where some elements, i.e., RUE instances outside the range of interest, have a zero probability associated to them.

We are also given a cost measure $\mathcal{C}(\theta, i)$ that assigns a value to each configuration θ when applied to a single problem instance *i*. Since the algorithm may be stochastic, this cost measure is often a random variable and we can only observe the cost value $c(\theta, i)$, that is, a realization of the random variable $\mathcal{C}(\theta, i)$. The cost value may be, for example, the best objective function value found within a given computation time. In decision problems, the cost measure may correspond to the computation time required to reach a decision, possibly bounded by a maximum cut-off time. In any case, the cost measure assigns a cost value to one run of a particular configuration on a particular instance. Finally, when configuring an algorithm for a problem, the criterion that we want to optimize is a function $F(\theta) : \Theta \to \mathbb{R}$ of the cost of a configuration θ with respect to the distribution of the random variable $\mathcal{I}$. The goal of automatic configuration is finding the best configuration θ^* that minimizes $F(\theta)$.

A usual definition of $F(\theta)$ is $E[\mathcal{C}(\theta, i)]$, the expected cost of θ. The definition of $F(\theta)$ determines how to rank the configurations over a set of instances. If the cost values over different instances are incommensurable, the median or the sum of ranks may be more meaningful. The precise value of $F(\theta)$ is generally unknown, and it can only be estimated by sampling. This sampling is performed in practice by obtaining realizations $c(\theta, i)$ of the random variable $\mathcal{C}(\theta, i)$, that is, by evaluating the algorithm configuration θ on instances sampled from $\mathcal{I}$. In other words, as most algorithms of practical interest are sufficiently complex to preclude an analytical computation, the configuration of such algorithms follows an experimental approach, where each experiment is a run of an implementation of the algorithm under specific experimental conditions [11].

2.3. Methods for automated algorithm configuration

The importance of the algorithm configuration problem has been noted by many researchers and, despite its importance, the manual approach has prevailed for a long time. In several papers, proposals were made to exploit more systematically techniques from the field of *design of experiments (DOE)* to set a, usually small, number of parameters. These methods include CALIBRA [2], which tunes up to five parameters using Taguchi partial designs combined with local search methods, methods based on response surface methodology [29] and more systematic applications of ANOVA techniques [80,83].

In configuration scenarios where all parameters are numerical, configuration approaches may rely on the application of classical black-box numerical optimizers, such as CMA-ES [38], BOBYQA [79], or MADS [5]. Although these methods are designed for continuous optimization, they can often optimize integer parameters by simply rounding the decision variables. MADS was used for tuning the parameters of various other direct search methods for continuous optimization. Later, it was extended to more general tuning tasks within the OPAL framework [6]. Yuan et al. [93] compared the three optimizers CMA-ES, BOBYQA and MADS with irace for tuning numerical parameters using various techniques for handling the stochasticity in the tuning problem. They concluded that BOBYQA works best for very few parameters (less than four or five), whereas CMA-ES is the best for a larger number of parameters. In a follow-up study, they introduced the post-selection method, where the numerical optimizers use few evaluations per configuration in a first phase, and the most promising configurations are evaluated by racing more carefully in a second post-selection phase, which deals better with the stochasticity of the configuration problem [94].

If we consider the full automatic configuration problem including conditional and categorical parameters, this problem can essentially be characterized as a stochastic black-box mixed-variables optimization problem. Therefore, apart from the above mentioned continuous direct search methods, many other heuristic optimization algorithms are natural candidates for tackling the algorithm configuration problem. Among the first proposals is the meta-GA algorithm proposed by Grefenstette [37], who used a genetic algorithm (GA) to tune the parameter settings of another GA. A more recent method is REVAC [74], an evolutionary algorithm that uses multi-parent cross-over and entropy measures to estimate the relevance of parameters. The gender-based GA [3] uses various sub-populations and a specialized cross-over operator to generate new candidate configurations. Hutter et al. [41] proposed ParamILS, an iterated local search method for automatic configuration that works only on categorical parameters and, hence, requires discretizing numerical ones. The evolutionary algorithm EVOCA [81] generates at each iteration two new candidates using a fitness proportionate crossover and a local search procedure; the candidates are evaluated a user-defined number of times on each instance to account for the stochastic behavior of the target algorithm.

The evaluation of configurations is typically the most computationally demanding part of an automatic configuration method, since it requires actually executing the target algorithm being tuned. Several methods aim to reduce this computational effort by using surrogate models to predict the cost value of applying a specific configuration to one or several instances. Based on the predictions, one or a subset of the most promising configurations are then actually executed and the prediction model is updated according to these evaluations. Among the first surrogate-based configuration methods is sequential parameter optimization (SPOT) [12]. A more general method also using surrogate models is the sequential model-based algorithm configuration (SMAC) [43]. A recent variant of the gender-based GA also makes use of surrogate models with promising results [4].

Finally, some methods apply racing [64] for selecting one configuration among a number of candidates using sequential statistical testing [20]. The initial candidate configurations for a race may be selected by DOE techniques, randomly or based on problem-specific knowledge. In the case of iterated racing, a sampling model is iteratively refined according to the result of previous races. The next section explains iterated racing, as implemented in the irace package.

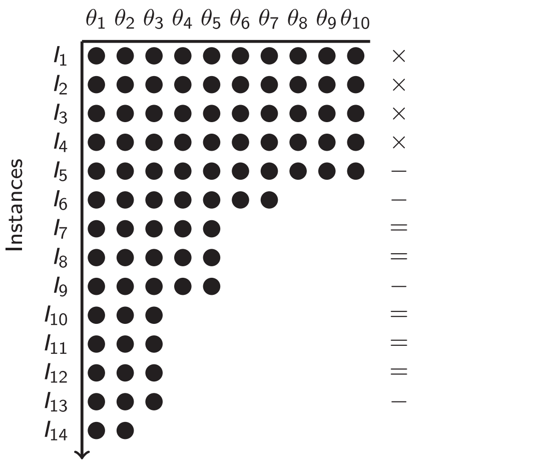
3. Iterated racing

3.1. An overview of iterated racing

The irace package that we describe in this paper is an implementation of iterated racing, of which I/F-Race [10,21] is a special case that uses Friedman's non-parametric two-way analysis of variance by ranks [28].

Iterated racing is a method for automatic configuration that consists of three steps: (1) sampling new configurations according to a particular distribution, (2) selecting the best configurations from the newly sampled ones by means of racing, and (3) updating the sampling distribution in order to bias the sampling towards the best configurations. These three steps are repeated until a termination criterion is met.

In iterated racing as implemented in the irace package, each configurable parameter has associated a sampling distribution that is independent of the sampling distributions of the other parameters, apart from constraints and conditions among parameters. The



θ_1 θ_2 θ_3 θ_4 θ_5 θ_6 θ_7 θ_8 θ_9 θ_{10}
Instances
I_1
I_2
I_3
I_4
I_5
I_6
I_7
I_8
I_9
I_{10}
I_{11}
I_{12}
I_{13}
I_{14}

Fig. 1. Racing for automatic algorithm configuration. Each node is the evaluation of one configuration on one instance. '× ' means that no statistical test is performed, ' – ' means that the test discarded at least one configuration, ' = ' means that the test did not discard any configuration. In this example, $T^{\text{first}} = 5$ and $T^{\text{each}} = 1$.

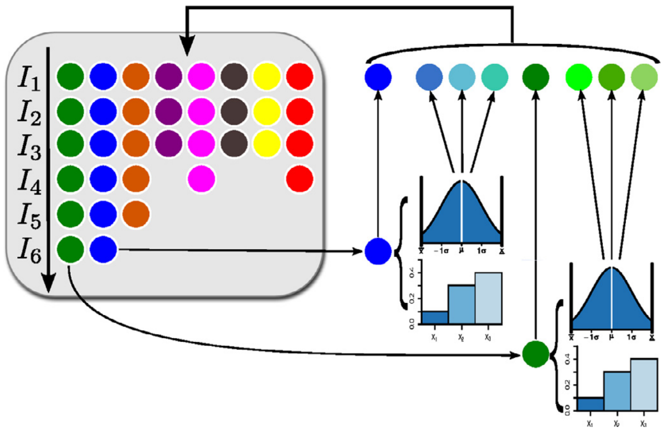


Fig. 2. Scheme of the iterated racing algorithm.

sampling distribution is either a truncated normal distribution for numerical parameters, or a discrete distribution for categorical parameters. Ordinal parameters are handled as numerical (integers). The update of the distributions consists in modifying the mean and the standard deviation in the case of the normal distribution, or the discrete probability values of the discrete distributions. The update biases the distributions to increase the probability of sampling, in future iterations, the parameter values in the best configurations found so far.

After new configurations are sampled, the best configurations are selected by means of racing. Racing was first proposed in machine learning to deal with the problem of model selection [64]. Birattari et al. [20] adapted the procedure for the configuration of optimization algorithms. A race starts with a finite set of candidate configurations. In the example of Fig. 1, there are ten configurations θ_i. At each step of the race, the candidate configurations are evaluated on a single instance (I_j). After a number of steps, those candidate configurations that perform statistically worse than at least another one are discarded, and the race continues with the remaining *surviving* configurations. Since the first elimination test is crucial, typically a higher number of instances (T^{first}) are seen before performing the first statistical test. Subsequent statistical tests are carried out more frequently, every T^{each} instances (by default for every instance). This procedure continues until reaching a minimum number of surviving configurations, a maximum number of instances that have been used or a pre-defined computational budget. This computational budget may be an overall computation time or a number of experiments, where an experiment is the application of a configuration to an instance.

An overview of the main steps of the iterated racing approach is given in Fig. 2. While the actual algorithm implemented in irace is a search process based on updating sampling distributions [96], the key ideas of iterated racing are more general. An iterated racing approach would be, from a more general perspective, any process that iterates the generation of candidate configurations with some form of racing algorithm to select the best configurations. Hence, the search process of an iterated racing approach could be, in principle, very different from the current irace algorithm, and make use, for example, of local searches, population-based algorithms or surrogate-models. The important element here is the appropriate combination of a search process with an evaluation process that takes the underlying stochasticity of the evaluation into account.

The next subsection (Section 3.2) gives a complete description of the iterated racing algorithm as implemented in the irace package. We mostly follow the description of the original papers [10,21], adding some details that were not explicitly given there. Later, in Section 3.3, we introduce a new "soft-restart" mechanism to avoid premature convergence and, in Section 3.4, we describe a new elitist variant of iterated racing aimed at preserving the best configurations found so far. In Section 3.5, we mention other features of irace that were not proposed in previous publications.

3.2. The iterated racing algorithm in the irace package

In this section, we describe the implementation of iterated racing as proposed in the irace package. The setup and options of the irace package itself are given in Appendix A. More details about the use of irace can be found in the user guide of the package [62].

An outline of the iterated racing algorithm is given in Algorithm 1. Iterated racing requires as input a set of instances I sampled from $\mathcal{I}$, a parameter space X, a cost function $\mathcal{C}$, and a tuning budget B.

Algorithm 1 Algorithm outline of iterated racing.

```
Require: $I = \{I_1, I_2, \dots\} \sim \mathcal{I}$,
     parameter space: $X$,
     cost measure: $\mathcal{C}(\theta, i) \in \mathbb{R}$,
     tuning budget: $B$
 1: $\Theta_1 = \text{SampleUniform}(X)$
 2: $\Theta^{\text{elite}} = \text{Race}(\Theta_1, B_1)$
 3: $j = 1$
 4: while $B^{\text{used}} \leq B$ do
 5:     $j = j + 1$
 6:     $\Theta^{\text{new}} = \text{Sample}(X, \Theta^{\text{elite}})$
 7:     $\Theta_j = \Theta^{\text{new}} \cup \Theta^{\text{elite}}$
 8:     $\Theta^{\text{elite}} = \text{Race}(\Theta_j, B_j)$
 9: end while
10: Output: $\Theta^{\text{elite}}$
```

Iterated racing starts by estimating the number of iterations N^{iter} (races) that it will execute. The default setting of N^{iter} depends on the number of parameters with $N^{\text{iter}} = \lfloor 2 + \log_2 N^{\text{param}} \rfloor$. The motivation for this setting is that we should dedicate more iterations for larger parameter spaces, with a minimum of two iterations per run to allow for some intensification of the search. Each iteration performs one race with a limited computation budget $B_j = (B - B^{\text{used}})/(N^{\text{iter}} - j + 1)$, where $j = 1, \dots, N^{\text{iter}}$. Each race starts from a set of candidate configurations Θ_j. The number of candidate configurations is calculated as $|\Theta_j| = N_j = \lfloor B_j/(\mu +$

$T^{\text{each}} \cdot \min\{5, j\})\rfloor$, that is, the number of candidate configurations decreases with the number of iterations, which means that more evaluations per configuration are possible in later iterations. The above setting also means that we do not keep decreasing N_j beyond the fifth iteration, to avoid having too few configurations in a single race. The parameter μ is by default equal to the number of instances needed to do a first test ($\mu = T^{\text{first}}$), and allows the user to influence the ratio between budget and number of configurations, which also depends on the iteration number j. The reason behind the formula above is the intuition that configurations generated in later iterations will be more similar and, hence, more evaluations will be necessary to identify the best ones.

In the first iteration, the initial set of candidate configurations is generated by uniformly sampling the parameter space X (line 1 in Algorithm 1) and the best configurations are determined by a race (line 2). When sampling the parameter space, parameters are considered in the order determined by the dependency graph of conditions, that is, non-conditional parameters are sampled first, those parameters that are conditional to them are sampled next if the condition is satisfied, and so on. When a race starts, each configuration is evaluated on the first instance by means of the cost measure $\mathcal{C}$. Configurations are iteratively evaluated on subsequent instances until a number of instances have been seen (T^{first}). Then, a statistical test is performed on the results. If there is enough statistical evidence to identify some candidate configurations as performing worse than at least another configuration, the worst configurations are removed from the race, while the others, the *surviving* configurations, are run on the next instance.

There are several alternatives for selecting which configurations should be discarded during the race. The F-Race algorithm [19,20] relies on the non-parametric Friedman's two-way analysis of variance by ranks (the Friedman test) and its associated post-hoc test described by Conover [28]. Nonetheless, the irace package, following the race package [18], also implements the paired t-test as an alternative option. Both statistical tests use a default significance level of 0.05 (the value can be customized by the user [62]). The statistical tests in irace are used as a selection heuristic and irace does not attempt to preserve the statistical significance level by sacrificing search performance. For example, the t-test is applied without p-value correction for multiple comparisons, since poor behavior of racing was previously reported if corrections are applied [19], due to the test becoming more conservative and not discarding configurations. Similarly, from a sequential statistical testing perspective, preserving the actual significance level would require additional adjustments that may hinder heuristic performance.

The most appropriate test for a given configuration scenario depends mostly on the tuning objective $F(\theta)$ and the characteristics of the cost function $\mathcal{C}(\theta, i)$. Roughly speaking, the Friedman test is more appropriate when the ranges of the cost function for different instances are not commensurable and/or when the tuning objective is an order statistic, such as the median, or a rank statistic. On the other hand, the t-test is more appropriate when the tuning objective is the mean of the cost function.

After the first statistical test, a new test is performed every T^{each} instances. By default $T^{\text{each}} = 1$, yet in some situations it may be helpful to perform each test only after the configurations have been evaluated on a number of instances. For example, given a configuration scenario with clearly defined instance classes, one may wish to find a single configuration that performs well, in general, for all classes. In that case, the sequence of instances presented to irace should be structured in blocks that contain at least one instance from each class, and T^{first} and T^{each} should be set as multiples of the size of each block. This ensures that configurations are only eliminated after evaluating them on every class, which reduces bias towards specific classes. We recommend this approach when configuring algorithms for continuous optimization benchmarks [39], where very different functions exist within the benchmark set and the goal is to find a configuration that performs well in all functions [57]. In this case, each block will contain one instance of every function, and different blocks will vary the number of decision variables and other parameters of the functions to create different instances of the same function.

Each race continues until the budget of the current iteration is not enough to evaluate all remaining candidate configurations on a new instance ($B_j < N_j^{\text{surv}}$), or when at most N^{min} configurations remain ($N_j^{\text{surv}} \le N^{\text{min}}$). At the end of a race, the surviving configurations are assigned a rank r_z according to the sum of ranks or the mean cost, depending on which statistical test is used during the race. The $N_j^{\text{elite}} = \min\{N_j^{\text{surv}}, N^{\text{min}}\}$ configurations with the lowest rank are selected as the set of elite configurations Θ^{elite}.

In the next iteration, before a race, a number of $N_j^{\text{new}} = N_j - N_{j-1}^{\text{elite}}$ new candidate configurations are generated (line 6 in Algorithm 1) in addition to the N_{j-1}^{elite} elite configurations that continue for the new iteration. For generating a new configuration, first one parent configuration θ_z is sampled from the set of elite configurations Θ^{elite} with a probability:

$$p_z = \frac{N_{j-1}^{\text{elite}} - r_z + 1}{N_{j-1}^{\text{elite}} \cdot (N_{j-1}^{\text{elite}} + 1)/2}, \tag{1}$$

which is proportional to its rank r_z. Hence, higher ranked configurations have a higher probability of being selected as parents.

Next, a new value is sampled for each parameter X_d, $d = 1, \dots, N^{\text{param}}$, according to a distribution that its associated to each parameter of θ_z. As explained before, parameters are considered in the order determined by the dependency graph of conditions, non-conditional parameters are sampled first followed by the conditional ones. If a conditional parameter that was disabled in the parent configuration becomes enabled in the new configuration, then the parameter is sampled uniformly, as in the initialization phase.

If X_d is a numerical parameter defined within the range $[\underline{x}_d, \overline{x}_d]$, then a new value is sampled from the truncated normal distribution $\mathcal{N}\left(x_d^z, (\sigma_d^j)^2\right)$, such that the new value is within the given range.[1] The mean of the distribution x_d^z is the value of parameter d in elite configuration θ_z. The standard deviation σ_d^j is initially set to $(\overline{x}_d - \underline{x}_d)/2$, and it is decreased at each iteration before sampling:

$$\sigma_d^j = \sigma_d^{j-1} \cdot \left(\frac{1}{N_j^{\text{new}}}\right)^{1/N^{\text{param}}}. \tag{2}$$

By reducing σ_d^j in this manner at each iteration, the sampled values are increasingly closer to the value of the parent configuration, focusing the search around the best parameter settings found as the iteration counter increases. Roughly speaking, the multidimensional volume of the sampling region is reduced by a constant factor at each iteration, and the reduction factor is higher when sampling a larger number of new candidate configurations (N_j^{new}).

If the numerical parameter is of integer type, we round the sampled value to the nearest integer. The sampling is adjusted to

[1] Sampling from a truncated normal distribution was never mentioned by previous description of I/F-Race [10,21]. However, naive methods of handling the ranges of numerical parameters, such as "rejection and resampling" or "saturation", may lead to under-sampling or over-sampling of the extreme values better methods exist [82]. For sampling from a truncated normal distribution, we use code from the msm package [46].

avoid the bias against the extremes introduced by rounding after sampling from a truncated distribution.[2]

If X_d is a categorical parameter with levels $X_d \in \{x_1, x_2, \dots, x_{n_d}\}$, then a new value is sampled from a discrete probability distribution $\mathcal{P}^{j,z}(X_d)$. In the first iteration ($j = 1$), $\mathcal{P}^{1,z}(X_d)$ is uniformly distributed over the domain of X_d. In subsequent iterations, it is updated before sampling as follows:

$$\mathcal{P}^{j,z}(X_d = x_j) = \mathcal{P}^{j-1,z}(X_d = x_j) \cdot \left(1 - \frac{j-1}{N^{\text{iter}}}\right) + \Delta\mathcal{P} \tag{3}$$

where

$$\Delta\mathcal{P} = \begin{cases} \dfrac{j-1}{N^{\text{iter}}} & \text{if } x_j = x_z, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Finally, the new configurations generated after sampling inherit the probability distributions from their parents. A set with the union of the new configurations and the elite configurations is generated (line 7 in Algorithm 1) and a new race is launched (line 8).

The algorithm stops if the budget is exhausted ($B^{\text{used}} > B$) or if the number of candidate configurations to be evaluated at the start of an iteration is not greater than the number of elites ($N_j \le N^{\text{elite}}_{j-1}$), since in that case no new configurations would be generated. If the iteration counter j reaches the estimated number of iterations N^{iter} but there is still enough remaining budget to perform a new race, N^{iter} is increased and the execution continues.

Although the purpose of most parameters in irace is to make irace more flexible when tackling diverse configuration scenarios, the iterated F-race procedure implemented in irace has several parameters that directly affect its search behavior. The default settings described here were defined at design time following common sense and experience. A careful fine-tuning of irace would require an analysis over a large number of relevant configuration scenarios. In a preliminary study, we analyzed the effects of the most critical parameters of irace [78] on a few classical scenarios. We could not find settings that are better for all scenarios than the default ones, and settings need to be adapted to scenario characteristics. The user guide of irace [62] provides advice, based on our own experience, for using different settings in particular situations.

3.3. *Soft-restart*

The iterated racing algorithm implemented in irace incorporates a "soft-restart" mechanism to avoid premature convergence. In the original I/F-Race proposal [10], the standard deviation, in the case of numerical parameters, or the discrete probability of unselected parameter settings, in the case of categorical ones, decreases at every iteration. Diversity is introduced by the variability of the sampled configurations. However, if the sampling distributions converge to a few, very similar configurations, diversity is lost and newly generated candidate configurations will be very similar to the ones already tested. Such a premature convergence wastes the remaining budget on repeatedly evaluating minor variations of the same configurations, without exploring new alternatives.

The "soft-restart" mechanism in irace checks for premature convergence after generating each new set of candidate configurations. We consider that there is premature convergence when the "distance" between two candidate configurations is zero. The distance between two configurations is the maximum distance between their parameter settings, which is defined as follows:

- If the parameter is conditional and disabled in both configurations, the distance is zero;
- if it is disabled in one configuration but enabled in the other, the distance is one;
- if the parameter is enabled in both configurations (or it is not conditional), then:
  - in the case of numerical parameters (integer or real), the distance is the absolute normalized difference between their values if this difference is larger than a threshold value $10^{-\texttt{digits}}$, where digits is a parameter of irace; if the difference is smaller, it is taken as zero;
  - in the case of ordinal and categorical parameters, the distance is one if the values are different and zero otherwise.

When premature convergence is detected, a "soft-restart" is applied by partially reinitializing the sampling distribution. This reinitialization is applied only to the elite configurations that were used to generate the candidate configurations with zero distance. The other elite configurations do not suffer from premature convergence, thus they may still lead to new configurations.

In the case of categorical parameters, the discrete sampling distribution of elite configuration z, $\mathcal{P}^{j,z}(X_d)$, is adjusted by modifying each individual probability value $p \in \mathcal{P}^{j,z}(X_d)$ as follows:

$$p' = 0.9 \cdot p + 0.1 \cdot \max\{\mathcal{P}^{j,z}(X_d)\},$$

and the resulting probabilities are normalized to [0, 1].

For numerical and ordinal parameters, the standard deviation of elite configuration z, $\sigma^{j,z}_d$, is "brought back" two iterations, with a maximum limit of its value in the second iteration. For numerical parameters this is done using

$$\sigma^{j,z}_d = \min\left\{\sigma^{j,z}_d \cdot \left(N^{\text{new}}_j\right)^{2/N^{\text{param}}}, \ \frac{\overline{x}_d - \underline{x}_d}{2} \cdot \left(\frac{1}{N^{\text{new}}_j}\right)^{1/N^{\text{param}}}\right\}$$

and for ordinal parameters, $\overline{x}_d - \underline{x}_d$ is replaced by $|X_d| - 1$, as these are the corresponding upper and lower bounds for an ordinal parameter.

After adjusting the sampling distribution of all affected elite configurations, the set of candidate configurations that triggered the soft-restart is discarded and a new set of N^{new} configurations is sampled from the elite configurations. This procedure is applied at most once per iteration.

3.4. *Elitist iterated racing*

The iterated racing procedure described above does not take into account the information from previous races when starting a new race. This may lead irace to erroneously discard a configuration based on the information from the current race, even though this configuration is the best found so far based on the information from all previous races. For example, if the first race identified a configuration as the best after evaluating it on ten instances, this configuration may get discarded in the next race after seeing only

[2] Let us assume we wish to sample an integer with range 1, 2, 3. The naive way would be to sample from a truncated distribution $\mathcal{N}(\mu = 2, \underline{x} = 1, \overline{x} = 3)$, and then round such that

$$\text{round}(x) = \begin{cases} 1 & \text{if } 1 < x < 1.5 \\ 2 & \text{if } 1.5 \le x < 2.5 \\ 3 & \text{if } 2.5 \le x < 3 \end{cases}$$

however, given these ranges, the interval of values that are rounded to 2 is twice the length of the interval that are rounded to either 1 or 3 and, thus, the sampling would be biased against the extreme values. We remove the bias if we instead sample from $\mathcal{N}(\mu = 2.5, \underline{x} = 1, \overline{x} = 4)$, and round such that

$$\text{round}(x - 0.5) = \begin{cases} 1 & \text{if } 1 < x < 2 \\ 2 & \text{if } 2 \le x < 3 \\ 3 & \text{if } 3 \le x < 4. \end{cases}$$

five instances (the default value for T^{first}), without taking into account the data provided by the ten previous evaluations. This may happen simply because of unlucky runs or because the best configuration overall may not be the best for a particular (small) subset of training instances. An empirical example of this situation is given in Section 4.3.

Ideally, the best configuration found should be the one evaluated in the largest number of instances, in order to have a precise estimation of its cost statistic $F(\theta)$ [41]. Therefore, we present here an elitist variant of iterated racing.[3] This elitist iterated racing aims at preserving the best configurations found so far, called elite configurations, unless they become worse than a new configuration that is evaluated in as many instances as the elite ones.

The main changes are in the racing procedure of irace. After the first race (iteration), the elite configurations have been evaluated on a number e of instances, for example, $I_1, \dots, I_e$. In the next race, we first randomize the order of the instances in this set and prepend to it a number of T^{new}newly sampled instances (by default one). Randomizing the order of the instances should help to avoid biases in the elimination test induced by a particularly lucky or unlucky order of instances. The rationale for prepending new instances is to give new configurations the opportunity to survive based on results on new instances, thus reducing the influence of already seen instances. This new set of $T^{\text{new}} + e$ instances is used for the new race. In a race, a new configuration is eliminated as usual, that is, if it is found to be statistically worse than the best one after performing a statistical test. However, elite configurations are not eliminated from the race unless the new configurations have been evaluated on at least $T^{\text{new}} + e$ instances, that is, as many instances as the elite configurations were evaluated in the previous race plus T^{new} new ones. If the race continues beyond $T^{\text{new}} + e$ instances, then new instances are sampled as usual, and any configuration may be eliminated from the race.

If a race stops before reaching $T^{\text{new}} + e$, which may happen when the number of remaining configurations is no more than N^{min}, configurations that were not elite are evaluated on fewer instances than the ones that were elite at the start of the race, thus there would be no unique value of e for the next iteration. We avoid this problem by keeping track of the instances on which each configuration has been evaluated so that we can calculate a value of e for each elite configuration.

With the elitist irace procedure described above, the number of configurations sampled at each race is limited by the number of instances seen in previous iterations. In particular,

$$|\Theta_j| = N_j = \left\lfloor \frac{B_j + (N_j^{\text{elite}} \cdot e)}{\max\{\mu + T^{\text{each}} \cdot \min\{5, j\},\ \text{nm}(T^{\text{new}} + e, T^{\text{each}})\}} \right\rfloor \tag{5}$$

where B_j is the computational budget assigned to the current iteration j, N_j^{elite} is the number of elite configurations, e is the maximum number of instances seen by the elite configurations, T^{new} is the minimum number of new instances to be evaluated in this iteration, and μ (by default T^{first}) and T^{each} control the frequency of statistical tests performed, as explained in Section 3.2. The function nm(x, d) gives the smallest multiple of d that is not less than x.

In elitist irace, as shown by the equation above, the number of new configurations that we can sample at each iteration is limited by the number of instances seen so far. Thus, if a particular iteration evaluates a large number of instances, then, subsequent iterations will be strongly limited by this number. This situation may arise, for example, in the first iteration, if most configurations are discarded after the first test and the remaining budget for this iteration will be spent on evaluating a few surviving configurations on new instances, but not being able to discard enough configurations to reach N^{min} and stop the race. This will result in a large value of e for the subsequent iterations, thus reducing the number of new configurations that can be sampled. In order to prevent this situation, we added a new stopping criterion that stops the race if there are T^{max} (by default 2) consecutive statistical tests without discarding any candidate. This stopping criterion is only applied after seeing $T^{\text{new}} + e$ instances, that is, when the statistical test may discard any elite configuration.

The described elitist strategy often results in a faster convergence to good parameter values, which has the disadvantage of reducing the exploration of new alternative configurations. Unfortunately, the soft-restart mechanism explained above is not sufficient to increase exploration in elitist irace, since it applies only when the sampling model of all parameters has converged, that is, when sampling a configuration almost identical to its parent. However, we observed in elitist irace that categorical parameters, in particular, tend to converge quite rapidly to consistently good values. Probably this happens because good overall values are easier to find at first and, in the elitist variant, it is harder to discard the elite configurations that contain them. Hence, they get continuously reinforced when updating their associated probabilities, but differences in numerical values prevent a soft-restart. In order to increase exploration, we limit the maximum probability associated to a categorical parameter value after updating it as follows:

$$p_i = \min\{p_i, 0.2^{1/N^{\text{param}}}\}. \tag{6}$$

and re-normalizing the probabilities to [0, 1].

3.5. Other features of irace

We have implemented in irace several extensions that were never mentioned in the original I/F-Race.

Initial configurations

We can seed the iterated race procedure with a set of initial configurations, for example, one or more default configurations of the algorithm. In that case, only enough configurations are sampled to reach N_1 in total.

Parallel evaluation of configurations

The training phase carried out by irace is computationally expensive, since it requires many runs of the algorithm being tuned. The total time required by a single execution of irace is mostly determined by the number of runs of the algorithm being tuned (the tuning budget, B) and the time required by those runs. In irace, these runs can be executed in parallel, either across multiple cores or across multiple computers using MPI. It is also possible to submit each run as a job in a cluster environment such as SGE or PBS. The user guide of irace [62] describes all the technical details.

Forbidden configurations

A user may specify that some parameter configurations should not be evaluated by defining such *forbidden configurations* in terms of logical expressions that valid configurations should not satisfy. In other words, no configuration that satisfies any of these expressions will be evaluated by irace. For example, given a parameter space with two parameters, a numerical one `param1` and a cat-

[3] This is the default racing procedure in irace starting from version 2.0, which is the latest version at the time of writing.

```
# name       switch            type values    [| conditions]
algorithm    "--"              c    (as, mmas, eas, ras, acs)
ls           "--localsearch "  c    (0, 1, 2, 3)
alpha        "--alpha "        r    (0.01, 5.00)
beta         "--beta "         r    (0.01, 10.00)
rho          "--rho  "         r    (0.00, 1.00)
ants         "--ants "         i    (5, 100)
nnls         "--nnls "         i    (5, 50)     | ls %
q0           "--q0 "           r    (0.0, 1.0)  | algorithm == "acs"
dlb          "--dlb "          c    (0, 1)      | ls %
rank         "--rasranks "     i    (1, 100)    | algorithm == "ras"
eants        "--elitistants "  i    (1, 750)    | algorithm == "eas"
```

Fig. 3. Parameter file (`parameters.txt`) for tuning ACOTSP. The first column is the name of the parameter; the second column is a label, typically the command-line switch that controls this parameter, which irace will concatenate to the parameter value when invoking the target algorithm; the third column gives the parameter type (either *integer*, *real*, *ordinal* or *categorical*); the fourth column gives the range (in case of numerical parameters) or domain (in case of categorical and ordinal ones); and the (optional) fifth column gives the condition that enables this parameter.

```
# Tuning budget in number of runs
maxExperiments = 5000
```

Fig. 4. Minimal scenario file (`scenario.txt`) for tuning ACOTSP. We use the default values for other options (e.g., `parameterFile=‘‘parameters.txt’’`, `targetRunner=‘‘./target-runner’’` and `trainInstancesDir=‘‘./Instances’’`), thus we do not need to specify them.

egorical one `param2`, and the following logical expression (in R syntax):

```
(param1 > 6 & param2 == "x1")
```

then a configuration such as {7, `‘‘x1’’`} will not be evaluated, whereas {7, `‘‘x2’’`} would be. This is useful, for example, if we know that certain combinations of parameter values lead to high memory requirements and, thus, they are infeasible in practice.

4. Applications of irace

In this section, we present two detailed examples of configuration scenarios and how to tackle them using irace. The first scenario illustrates the tuning of a single-objective metaheuristic (ant colony optimization) on a well-known problem (the traveling salesman problem). The second scenario concerns the tuning of a framework of multi-objective ant colony optimization algorithms. We have chosen these scenarios for illustrative purposes and because their setup is available in AClib [44].

4.1. Example of tuning scenario: tuning ACOTSP

ACOTSP [87] is a software package that implements various ant colony optimization algorithms to tackle the symmetric traveling salesman problem (TSP). The configuration scenario illustrated here concerns the automatic configuration of all its 11 parameters. The goal is to find a configuration of ACOTSP that obtains the lowest solution cost in TSP instances within a given computation time limit. We explain here the setup of this configuration scenario. For a more detailed overview of the possible options in the irace package, we refer to Appendix A.

First, we define a parameter file (`parameters.txt`, Fig. 3) that describes the parameter space, as explained in Section A.2. We also create a scenario file (`scenario.txt`, Fig. 4) to set the tuning budget (`maxExperiments`) to 5 000 runs of ACOTSP. Next, we place the training instances in the subdirectory `‘‘./Instances/’’`, which is the default value of the option `trainInstancesDir`. We create a basic `target-runner-run` script that runs the ACOTSP software for 20 CPU-seconds and prints the objective value of the best solution found.

At the end of a run, irace prints the best configurations found as a table and as command-line parameters:

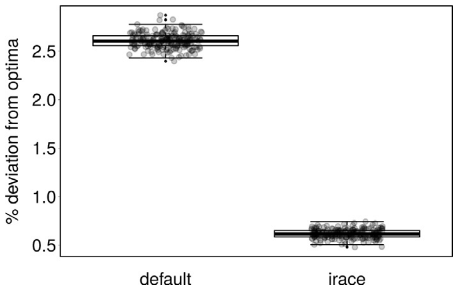


Fig. 5. Comparison of configurations obtained by (elitist) irace and the default configuration of ACOTSP.

```
# Best configurations (first number is the configuration ID)
    algorithm  ls alpha beta  rho ants nnls   q0 dlb rank eants
530       acs   3  1.93 8.27 0.36   21   10 0.78   1   NA    NA
635       acs   3  2.52 9.69 0.29   42   11 0.83   1   NA    NA
552       acs   3  1.81 7.86 0.25   20   16 0.72   1   NA    NA
741       acs   3  1.78 5.74 0.23   31   13 0.64   1   NA    NA
700       acs   3  1.85 5.97 0.23   34   12 0.68   1   NA    NA
# Best configurations as commandlines (first number is the configuration ID)
530  --acs --localsearch 3 --alpha 1.93 --beta 8.27 --rho  0.36 \
     --ants 21 --nnls 10 --q0 0.78 --dlb 1
635  --acs --localsearch 3 --alpha 2.52 --beta 9.69 --rho  0.29 \
     --ants 42 --nnls 11 --q0 0.83 --dlb 1
552  --acs --localsearch 3 --alpha 1.81 --beta 7.86 --rho  0.25 \
     --ants 20 --nnls 16 --q0 0.72 --dlb 1
741  --acs --localsearch 3 --alpha 1.78 --beta 5.74 --rho  0.23 \
     --ants 31 --nnls 13 --q0 0.64 --dlb 1
700  --acs --localsearch 3 --alpha 1.85 --beta 5.97 --rho  0.23 \
     --ants 34 --nnls 12 --q0 0.68 --dlb 1
```

where the first number of each row is a unique number that identifies a particular configuration within a single run of irace, and `NA` denotes that a parameter did not have a value within a particular configuration (because it was not enabled).

Fig. 5 compares the configurations obtained by 30 independent runs of elitist irace and the default configuration of ACOTSP. Each run of irace uses the settings described above and a training set of 200 Euclidean TSP instances of size 2 000 nodes. The best configurations found by each run of irace are run once on a (different) test set of 200 instances of size 2 000, while the default configuration of ACOTSP is run 30 times on the same set using different random seeds. We then report the percentage deviation from the optimal objective value. Each data point shown in Fig. 5 corresponds to an instance, values are the mean of the results obtained either by the 30 runs of the default configuration of ACOTSP or by the 30 configurations produced by irace. In order to reduce variability, we associate a random seed to each instance and use this seed for all runs performed on that instance.

As we can observe, the improvement of the tuned configuration over the default one is significant. In practice, we often observe that the largest improvements are obtained when configuring an optimization algorithm for scenarios that differ substantially from those for which it was designed, either in terms of problem in-

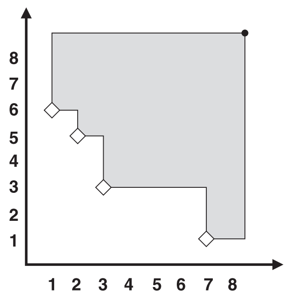


Fig. 6. Example of the computation of the hypervolume quality measure. Each white diamond represents a solution in the objective space of a bi-objective minimization problem. The black point is a reference point that is worse in all objectives than any Pareto-optimal solution. The area of the objective space dominated by all solutions in the set and bounded by the reference point is called its hypervolume. The larger the hypervolume of a set, the higher the quality of the set.

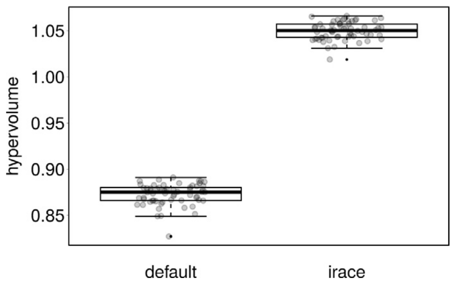


Fig. 7. Comparison of a configuration obtained by (elitist) irace and the default configuration of MOACO. Hypervolume values should be maximized (since irace minimizes by default, `targetRunner` multiplies the values by −1 before returning them to irace).

stances or in terms of other characteristics of the scenario, such as termination criteria or computation environment. Nonetheless, it is not rare that an automatic configuration method finds a better configuration than the default even for those scenarios considered when designing an algorithm and even when not providing the default configuration as an initial configuration as in our examples here.

4.2. A more complex example: tuning multi-objective optimization algorithms

In this section, we explain how to apply irace to automatically configure algorithms that tackle multi-objective optimization problems in terms of Pareto optimality. This example illustrates the use of an additional script (or R function) called `targetEvaluator`.

In multi-objective optimization in terms of Pareto optimality, the goal is to find the Pareto front, that is, the image in the objective space of those solutions for which there is no other feasible solution that is better in all objectives. For many interesting problems, finding the whole Pareto front is often computationally intractable, thus the goal becomes to approximate the Pareto front as well as possible. Algorithms that approximate the Pareto front, such as multi-objective metaheuristics, typically return a set of nondominated solutions, that is, solutions for which no other solution in the same set is better in all objectives.

Automatic configuration methods, such as irace, have been primarily designed for single-objective optimization, where the quality of the output of an algorithm can be evaluated as a single numerical value. In the case of multi-objective optimization, unary quality measures, such as the hypervolume (see Fig. 6) and the ϵ-measure [95], assign a numerical value to a set of nondominated solutions, thus allowing the application of standard automatic configuration methods [58,91]. However, computing these unary quality measures often requires a reference point (or set), which depends on the sets being evaluated. One may define the reference point a priori based on some knowledge about the instances being tackled, such as lower/upper bounds. On the other hand, it would be desirable if the reference point could be computed from the results obtained while carrying out the automatic configuration process. In irace, the latter can be achieved by first running all candidate configurations on a single instance and, once all these runs have finished, computing the cost/quality value of each configuration in a separate step.

In practical terms, this means that the `targetRunner` program is still responsible for running the configuration θ on instance i, but it does not compute the value $c(\theta, i)$. This value is computed by a different `targetEvaluator` program that runs after all `targetRunner` calls for a given instance i have finished. The communication between `targetRunner` and `targetEvaluator` is scenario-specific and, hence, defined by the user.

In the case of elitist irace, `targetEvaluator` might be called including configurations that were evaluated in a previous race, since they were elite. Therefore, one way to dynamically compute the reference point for the hypervolume computation may be by `targetRunner` saving the nondominated sets corresponding to each pair (θ, i) and `targetEvaluator` using them (and not deleting them since they might be needed again later) to update the reference point or the normalization bounds.

We have applied the above method to automatically configure various multi-objective optimization algorithms by means of irace. In particular, we first applied irace to instantiate new algorithmic designs from a framework of multi-objective ant colony optimization algorithms (MOACO) [58]. In that work, we tested the combination of irace with the hypervolume measure and the ϵ-measure, but we did not find significant differences between the results obtained with each of them. The MOACO algorithms automatically instantiated by irace were able to significantly outperform previous MOACO algorithms proposed in the literature. Fig. 7 compares the results of a configuration obtained by (elitist) irace and the best manually-designed configuration from the literature [58] on 60 bi-objective Euclidean TSP instances of sizes {500,600,700,800,900,1000}. The complete MOACO scenario is too complex to describe here, but it is provided as an example together with irace and it is also included in AClib [44].

4.3. Comparison of irace and elitist irace

In this section, we compare irace with and without elitism in three configuration scenarios.[4]

ACOTSP is similar to the scenario described in Section 4.1. We consider a budget of 5 000 runs of ACOTSP and 20 s of CPU-time per run. As benchmark set, we consider Euclidean TSP instances of size 2000, in particular, 200 training instances and 200 test instances.

MOACO is similar to the scenario described in Section 4.2. The benchmark instances are bi-objective Euclidean TSP instances of sizes {500, 600, 700, 800, 900, 1000}, 10 training instances and 60 test instances of each size. We use a budget of 5 000 runs of the MOACO framework, and each run of a MOACO algorithm is stopped after $4 \cdot (n/100)^2$ CPU-seconds, where n is the instance size.

SPEAR where the goal is to minimize the mean runtime of Spear, an exact tree search solver for SAT problems [9] with

[4] All files needed to replicate these scenarios are provided as supplementary material [61].

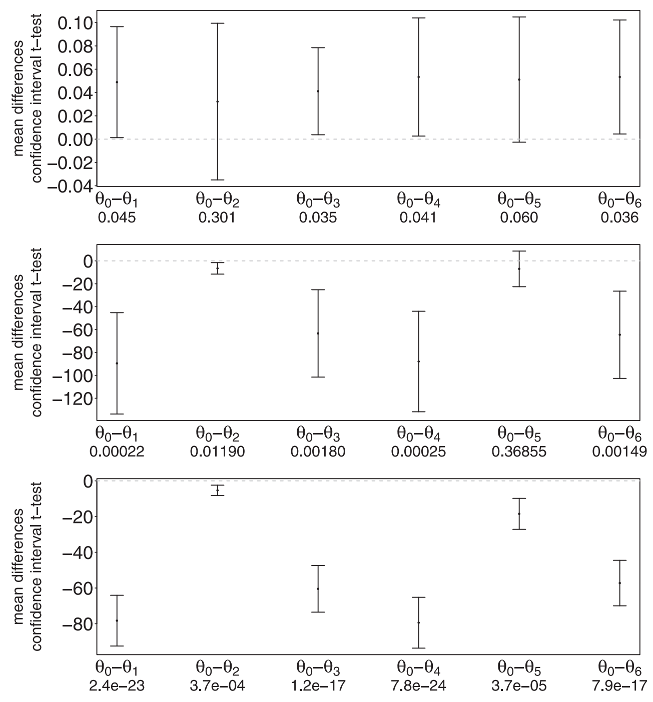


Fig. 8. Example of a good elite configuration (θ_0) lost when configuring SPEAR with the non-elitist irace using t-test as statistical test. The plots give the 95% confidence intervals of the mean differences between the results obtained by θ_0 and the new elite configurations ($\theta_1, \dots, \theta_6$) on different subsets of the training set. The top plot considers only the 9 instances seen by irace at the iteration in which θ_0 was discarded, the middle plot considers the 37 instances on which θ_0 was evaluated since the start of this run, and the bottom plot considers the full training set. Negative values indicate that θ_0 has a better performance than the configuration to which it is compared. If the interval contains zero, there is no statistical difference. The p-values of the t-test performed between θ_0 and ($\theta_1, \dots \theta_6$) are reported below each interval.

26 categorical parameters. We consider a budget of 10 000 runs, a maximum runtime of 300 CPU-seconds per run, and a training and a test set of 302 different SAT instances [8].

Instance homogeneity is an important factor when tuning an algorithm. We measure instance homogeneity by means of the Kendall concordance coefficient (W) [78] computed from the results of 100 uniformly random generated algorithm configurations executed on the training set. Values of W close to 1 indicate high homogeneity while values close to 0 indicate high heterogeneity. The W values for the ACOTSP, MOACO and SPEAR scenarios are 0.98, 0.99 and 0.16 respectively, showing that SPEAR is a highly heterogeneous scenario.

Given the characteristics of each scenario [78], we use as the statistical elimination test in irace the default F-test in the ACOTSP and MOACO scenarios, and the t-test in the SPEAR scenario. For each scenario, we run irace 30 times on the training set, obtaining 30 different algorithm configurations. Then, we run each of these configurations on the test set, which is always different from the training set.

As explained in Section 3.4, the non-elitist irace may discard high-quality configurations due to the use of partial information, ignoring the results obtained in past iterations. A concrete example is shown in Fig. 8 for an actual run of irace on the SPEAR scenario. The plots give the 95% confidence intervals of the mean differences between the results obtained by configuration θ_0, which is an elite configuration indentified in iteration 5, and the elite candidates ($\theta_1, \dots, \theta_6$) obtained in iteration 6. Configuration θ_0 is discarded by irace in iteration 6 due to statistically significantly worse performance than configuration θ_6 after 9 instances were executed (top plot); θ_0 is also statistically significantly worse than $\theta_{1,3,4}$ and it has a worse mean than $\theta_{2,5}$. However, θ_0 has a statistically better performance on the full training set when compared to all elite candidates of iteration 6 (bottom plot). Even if we consider only the 37 training instances on which θ_0 was evaluated from the start of this run of irace up to the moment it was discarded, θ_0 is significantly better than all configurations that irace selects as final elites (see middle plot of Fig. 8). Hence, using all available information, irace would have detected that θ_0 is the best configuration of the group. The loss of such potentially winning configurations happened in eight of the 30 executions of the non-elitist irace on SPEAR.

Next, we perform experiments comparing both variants of irace on the three configuration scenarios mentioned above across 30 repetitions. Fig. 9 shows the results for each scenario as the mean performance reached by each of the 30 configurations generated for each scenario on the test set. On the left, it shows box-plots of the results and, on the right, scatter plots where each point pairs the executions of elitist and non-elitist irace using the same set of initial configurations. In none of the three scenarios, we observe statistically significant differences.

4.4. Heterogeneous scenario setting

Examining closer the results in the previous section (middle plots of Fig. 9), one may notice that the non-elitist irace produces the two worst configurations found for SPEAR. These two particularly bad runs of irace discard elite configurations as explained above. We conjecture that for heterogeneous training sets such as in the SPEAR scenario, the elitist version may avoid the loss of high-quality configurations, thus producing more consistent results. In fact, facing scenarios with an heterogeneous set of training instances is a difficult task for automatic configuration methods, which normally work better in homogeneous scenarios [85]. In an heterogeneous scenario, measuring the quality of a configuration typically requires evaluating on a large number of instances in order to find configurations that optimize the algorithm performance across the training set and to capture also the possible existence of few rare but hard instances. Unfortunately, evaluating on more instances with the same tuning budget strongly reduces the ability of the tuner to explore new configurations and, hence, there is a trade-off between increasing the confidence on the quality of configurations and sampling effectively the configuration space.

When using irace to tackle very heterogeneous scenarios, it may be useful to adjust the default settings to increase the number of instances evaluated by each configuration. For elitist irace this can be achieved by increasing the number of new instances added initially to a race (T^{new}); in non-elitist irace this can be achieved by increasing the number of instances needed to perform the first statistical test (T^{first}). Fig. 10 gives the mean runtime per candidate on the test set of the algorithm configurations obtained by 10 runs of the elitist irace (top) using various values of T^{new} and of the non-elitist irace using various values of T^{first} (bottom) on the SPEAR scenario. We can observe that using a larger value than the default for T^{new} and T^{first} strongly improves the cost (mean runtime) of the final configurations, because configurations are evaluated on more instances before selecting which one should be discarded. Further increasing the values of T^{new} and T^{first} does not lead to further improvements because enough instances are already seen to account for their heterogeneity. It does lead, however, to fewer configurations being explored, thus, at some point, larger values will actually generate worse configurations. This effect is stronger for T^{first} because all configurations at each iteration have to be evaluated on that many instances, which consumes a substantial amount of budget and results in a much lower number of configurations being generated. This is shown by the number

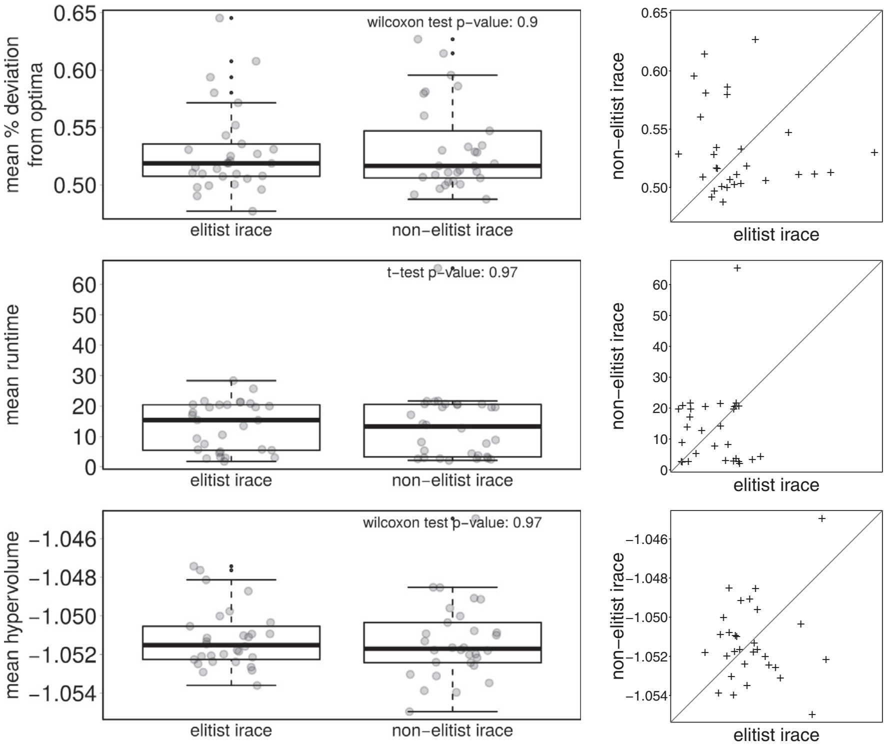


Fig. 9. Comparison between elitist and non-elitist irace. Plots give the mean candidate performance on the test instance set as % deviation from optima, runtime and hypervolume for the ACOTSP (top), SPEAR (middle), and MOACO (bottom) scenarios respectively. Hypervolume values are multiplied by −1 so that all scenarios must be minimized. The *p*-values of the statistical test are reported on the left plots.

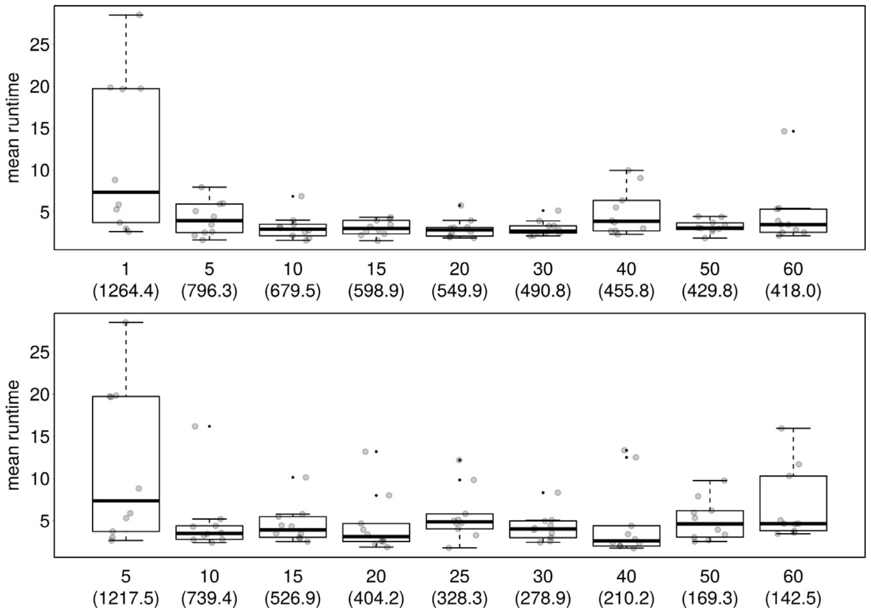


Fig. 10. Comparison between 10 configurations obtained by the elitist irace (top plot) on SPEAR adding $T^{\text{new}} = \{1, 5, 10, 15, 20, 30, 40, 50, 60\}$ new instances at the beginning of the race (default value is 1) and 10 candidates obtained by the non-elitist irace (bottom plot) using $T^{\text{first}} = \{5, 10, 15, 20, 25, 30, 40, 50, 60\}$ as the required number of instances to perform the first statistical test of the race (default value is 5). Values in parenthesis are the mean number of candidates sampled by the 10 irace executions.

within parentheses in Fig. 10. In the case of T^{new}, non-elite configurations may be discarded before seeing T^{new} instances and the effect on the budget consumed is lower. The same experiment for the ACOTSP scenario showed that the best configurations become worse when T^{new} or T^{first} are increased. This is due to the fact that ACOTSP has a homogeneous training set and, therefore, sampling new candidates is more important than executing a large number of instances.

5. Other applications of irace

Since the first version of the irace package became publicly available in 2012, there have been many other applications of irace. In this section, we provide a list of the applications of the irace package of which we are aware at the time of writing. Some of these applications go beyond what is traditionally understood as algorithm configuration, demonstrating the flexibility of irace.

5.1. Algorithm configuration

The traditional application of irace is the automatic configuration of algorithms. Several publications have used irace when evaluating or designing algorithms for problems such as the traveling salesman with time windows [60], slot allocation [77], generalized hub location [68], flow shop [14], virtual machine placement [86], online bin packing [92], graph coloring [23], image binarization [69], network pricing [90], combined routing and packing problems [25], real-time routing selection [84], capacitated arc routing [26], bike sharing rebalancing [30], energy planning [47], university course timetabling [72], time series discretization [1], finite state machine construction [27], minimum common string partition and minimum covering arborescence [24], and continuous (real-valued) optimization [7,52,53,55,56,75].

Automatic configuration is also useful when the goal is to analyze the effect of particular parameters or design choices. Instead of a (factorial) experimental design, which often is intractable because of the large number of parameters and/or the limited computation time available, the analysis starts from very high-performing configurations found by an automatic configuration method, and proceeds by changing one parameter at a time. An example is the analysis of a hybrid algorithm that combines ant colony optimization and a MIP solver to tackle vehicle routing problems (VRP) with black-box feasibility constraints [67]. Pellegrini et al. [76] applied this principle to the analysis of parameter adaptation approaches in ant colony optimization. More recently, Bezerra et al. [15] have applied the same idea to analyze the contribution of various algorithmic components found in multi-objective evolutionary algorithms.

The idea behind the above analysis is that, in terms of performance, there are many more “uninteresting” configurations than “interesting” ones, and statements about the parameters of uninteresting configurations are rarely useful, thus it makes more sense to start the analysis with high-performing configurations. In its general form, such procedure may be used to analyze differences between configurations, which has been described as *ablation* [32].

5.2. Multi-objective optimization metaheuristics

Besides the application to the MOACO framework described above [58], irace has been applied to aid in the design of other multi-objective optimization algorithms. Dubois-Lacoste et al. [31] used irace to tune a hybrid of two-phase local search and Pareto local search (TP + PLS) to produce new state-of-the-art algorithms for various bi-objective permutation flowshop problems. Fisset et al. [33] used irace to tune a framework of multi-objective optimization algorithms for clustering. When applied to a sufficiently flexible algorithmic framework, irace has been used to design new state-of-the-art multi-objective evolutionary algorithms [16,17].

5.3. Anytime algorithms (improve time-quality trade-offs)

There is often a trade-off between solution quality and computation time: Algorithms that converge quickly tend to produce better solutions for shorter runtimes, whereas more exploratory algorithms tend to produce better solutions for longer runtimes. Improving the anytime behavior of an algorithm amounts to improving the trade-off curve between solution quality and computation time such that an algorithm is able to produce as high quality solutions as possible at any moment during their execution. López-Ibáñez and Stützle [59] modeled this trade-off curve as a multi-objective optimization problem, and measured the quality of the trade-off curve using the hypervolume quality measure. This approach allows the application of irace to tune the parameters of an algorithm for improving its anytime behavior. They applied this technique to tune parameter variation strategies for ant colony optimization algorithms, and to tune the parameters of SCIP, a MIP solver, in order to improve its anytime behavior. The results show that the tuned algorithms converge much faster to good solutions without sacrificing the quality of the solutions found after relatively longer computation time.

5.4. Automatic algorithm design from a grammar description

Algorithm configuration methods have been used in the literature to instantiate algorithms from flexible algorithmic frameworks in a top-down manner, that is, the framework is a complex algorithm build from components of several related algorithms and specific components can be selected through parameters. One example using ParamILS is SATenstein [51]. Examples using irace include the MOACO framework described above [58] and multi-objective evolutionary algorithms [17]. A different approach describes the potential algorithm designs as a grammar. This provides much more flexibility when composing complex algorithms. Mascia et al. [66] proposed a method for describing a grammar as a parametric space that can be tuned by means of irace in order to generate algorithms. They applied this technique to instantiate iterated greedy algorithms for the bin packing problem and the permutation flowshop problem with weighted tardiness. Marmion et al. [63] applied this idea to automatically design more complex hybrid local search metaheuristics.

5.5. Applications in machine learning

In machine learning, the problem of selecting the best model and tuning its (hyper-)parameters is very similar to automatic algorithm configuration. Thus, it is not surprising that irace has been used for this purpose, for example, for tuning the parameters of support vector machines [70]. Lang et al. [54] used irace for automatically selecting models (and tuning their hyperparameters) for analyzing survival data. The automatically tuned models significantly outperform reference (default) models. The mlr software package [22] uses irace, among other tuning methods, for tuning the hyperparameters of machine learning models as a better performing alternative to random search and grid search.

5.6. Automatic design of control software for robots

A very original application of irace is the automatic design of control software for swarms of robots. Francesca et al. [35] propose a system to automatically design the software that controls a swarm of robots in order to achieve a specific task. The problem is specified as a series of software modules that provide many different robot behaviors and the criteria to transition between behaviors. Each module can be further customized by means of several parameters. A particular combination of behaviors and transitions represents one controller, that is, an instance of the software that controls the robots in the swarm. The performance of a particular controller is evaluated by means of multiple simulations. The search for the best controller over multiple training simulations is carried out by means of irace. The authors report that this system is not only able to outperform a previous system that used F-race [34], but also a human designer, under the scenarios studied by them.

6. Conclusion

This paper presented the irace package, which implements the iterated racing procedure for automatic algorithm configuration. Iterated racing is a generalization of the iterated F-race procedure.

The primary purpose of irace is to automatize the arduous task of configuring the parameters of an optimization algorithm. However, it may also be used for determining good settings in other computational systems such as robotics, traffic light controllers, compilers, etc. The irace package has been designed with simplicity and ease of use in mind. Despite being implemented in R, no previous knowledge of R is required. We included two examples for the purposes of illustrating the main elements of an automatic configuration scenario and the use of irace to tackle it. In addition, we provided a comprehensive survey of the wide range of applications of irace.

There are a number of directions in which we are trying to extend the current version of irace. One is the improvement of the sampling model to take into account interactions among parameters. In some cases, irace converges too quickly and generates very similar configurations; thus, additional techniques to achieve a better balance between diversification and intensification seem worth pursuing. In the same direction, techniques for automatically adjusting some settings of irace (such as T^{new} and T^{first}) in dependence of the heterogeneity of a scenario would be useful. Finally, we are currently adding tools to provide a default analysis of the large amount of data gathered during the run of irace to give the user information about the importance of specific parameters and the most relevant interactions among the parameters.

The iterated racing algorithms currently implemented in irace have, however, a few well-known limitations. The most notable is that they were primarily designed for scenarios where reducing computation time is not the primary objective. Methods designed for such type of scenarios, such as ParamILS [41] and SMAC [43], dynamically control the maximum time assigned to each run of the target algorithm and use an early pruning of candidate configurations in order to not waste time on time-consuming and, therefore, poor configurations. Moreover, the default parameters of irace assume that a minimum number of iterations can be performed and a minimum number of candidate configurations can be sampled. If the tuning budget is too small, the resulting configuration might not be better than random ones.

Finally, automatic configuration methods in general may be difficult to apply when problem instances are computationally expensive, for example, when the computational resources available are limited (lack of multiple CPUs, cluster of computers) or when a single run of the algorithm requires many hours or days. In such situations, two main alternatives have been proposed in the literature. Styles et al. [88] proposed to use easier instances (less computationally expensive) during tuning to obtain several good configurations, and then apply a racing algorithm to these configurations using increasingly difficult instances to discard those configurations that do not scale. Mascia et al. [65] proposed to tune on easy instances, and then identify which parameters need to be modified and how in order for the algorithm to scale to more difficult instances.

The main purpose of automatic algorithm configuration methods is to configure parameters of optimization and other algorithms. Nonetheless, the use of these methods has a crucial role in new ways of designing software, as advocated in the programming by optimization paradigm [40]. Moreover, the importance of properly tuning the parameters of algorithms before analyzing and comparing them is becoming widely recognized. We hope that the development of the irace package will help practitioners and researchers to put these ideas into practice.

Acknowledgements

This work received support from the COMEX project (P7/36) within the Interuniversity Attraction Poles Programme of the Belgian Science Policy Office, and the EU FP7 ICT Project COLOMBO, Cooperative Self-Organizing System for Low Carbon Mobility at Low Penetration Rates (agreement no. 318622). Thomas Stützle and Mauro Birattari acknowledge support from the Belgian F.R.S.-FNRS, of which they are Senior Research Associates. Leslie Pérez Cáceres acknowledges support of CONICYT Becas Chile.

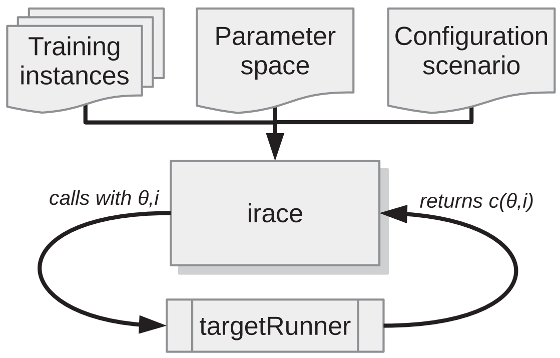


Fig. A.11. Scheme of the user-provided components required by irace.

Table A1
Parameters of irace corresponding to the description of iterated racing given in Section 3.2. The full list of irace parameters is available in the user guide.

Iterated racing parameter	irace parameter
B	`maxExperiments`
$\mathcal{C}$ (cost measure)	`targetRunner`
μ	`mu`
N^{min}	`minNbSurvival`
T^{first}	`firstTest`
T^{each}	`eachTest`
Statistical test	`testType`

Appendix A. The irace package

We provide here a brief summary of the irace package. The full documentation is available together with the package and in the irace user guide [62].

The scheme in Fig. A.11 shows the components that the user must provide to irace before executing it. First, irace requires three main inputs:

1. A description of the parameter space X, that is, the parameters to configure, their types, domains and constraints. Section A.2 summarizes how to define a parameter space in irace.
2. The set of training instances $\{I_1, I_2, \ldots\}$, which in practice is a finite, representative sample of $\mathcal{I}$. The particular options for specifying the set of training instances are given in Section A.1.
3. The configuration scenario, which is defined in terms of options provided to irace. Table A.1 maps the description of iterated racing in Section 3.2 to the options of irace. The complete list of options is available in the software documentation.

In addition, irace requires a program (or R function) called `targetRunner` that is responsible for evaluating a particular configuration of the target algorithm on a given instance and returning the corresponding cost value.

A1. Training instances

The set of training instances $\{I_1, I_2, \ldots\}$ may be given explicitly as an option to irace. Alternatively, the instances may be read, one per line, from an instance file (`trainInstancesFile`). Typically, an instance is a path to a filename and the string given by option `trainInstancesDir` will be prefixed to them. Nonetheless, an instance may also be the parameter settings for selecting a benchmark function implemented in the target algorithm or for invoking an instance generator (in that case, option `trainInstancesDir` should be set to the empty string). If the option `trainInstancesFile` is not set, then irace considers all files found in `trainInstancesDir`, and recursively in its subdirectories, as training instances. The order in which instances are considered by irace is randomized if the option `sampleInstances` is enabled. Otherwise, the order is the same as given in `trainInstancesFile` if this option is set or in alphabetical order, otherwise.

In order to reduce variance, irace uses the same random seed to evaluate different configurations on the same instance. If an instance is seen more than once, a different random seed is assigned to it. Thus, in practice, the sequence of instances seen within a race (Fig. 1) is actually a sequence of instance and seed pairs.

A2. Parameter space

For simplicity, the description of the parameter space is given as a table. Each line of the table defines a configurable parameter:

```
<name> <label> <type> <domain> [ | <condition> ]
```

where each field is defined as follows:

`<name>`	The name of the parameter as an unquoted alphanumeric string, for instance: ‘`ants`’.
`<label>`	A label for this parameter. This is a string that will be passed together with the parameter to `targetRunner`. In the default `targetRunner` provided with the package, this is the command-line switch used to pass the value of this parameter, for instance ‘`"--ants "`’.
`<type>`	The type of the parameter, either *integer*, *real*, *ordinal* or *categorical*, given as a single letter: ‘`i`’, ‘`r`’, ‘`o`’ or ‘`c`’.
`<domain>`	The range (for integers and real parameters) or the set of values (for categorical and ordinal) of the parameter.
`<condition>`	An optional *condition* that determines whether the parameter is enabled or disabled, thus making the parameter conditional. If the condition evaluates to false, then no value is assigned to this parameter, and neither the parameter value nor the corresponding label are passed to `targetRunner`. The condition must be a valid R logical expression. The condition may contain the name of other parameters as long as the dependency graph does not have any cycles. Otherwise, irace will detect the cycle and stop with an error.

Parameter types and domains

Parameters can be of four types:

- *Real* parameters are numerical parameters that can take any floating-point values within a given range. The range is specified as an interval ‘`(<lower bound>,<upper bound>)`’. This interval is closed, that is, the parameter value may eventually be one of the bounds. The possible values are rounded to a number of *decimal places* specified by the option `digits`. For example, given the default number of digits of 4, the values 0.12345 and 0.12341 are both rounded to 0.1234.
- *Integer* parameters are numerical parameters that can take only integer values within the given range. The range is specified as for real parameters.
- *Categorical* parameters are defined by a set of possible values specified as ‘`(<value 1>, ... , <value n>)`’. The values are quoted or unquoted character strings. Empty strings and strings containing commas or spaces must be quoted.
- *Ordinal* parameters are defined by an *ordered* set of possible values in the same format as for categorical parameters. They are handled internally as integer parameters, where the integers correspond to the indices of the values.

A3. Output of irace

During its execution, irace prints a detailed report of its progress. In particular, after each race finishes, the elite configurations are printed; and at the end, the best configurations found are printed as a table and as command-line parameters (see the example output shown in Section 4.1)

In addition, irace saves an R dataset file, by default as `irace.Rdata`, which may be read from R by means of the function `load()`. This dataset contains a list `iraceResults`, whose most important elements are:

`scenario`: the configuration scenario given to irace (any option not explicitly set has its default value).
`parameters`: the parameter space.
`seeds`: a matrix with two columns, `instance` and `seed`. Rows give the sequence of pairs instance–seed seen by irace.
`allConfigurations`: a data frame with all configurations generated during the execution of irace.
`experiments`: a matrix storing the result of all experiments performed across all iterations. Each entry is the result of evaluating one configuration on one instance at a particular iteration. Columns correspond to configurations and match the row indexes in `allConfigurations`. Rows match the row indexes in the matrix `seeds`, giving the instance–seed pair on which configurations were evaluated. A value of ‘`NA`’ means that this configuration was not evaluated on this particular instance, either because it did not exist yet or it was discarded.

The `irace.Rdata` file can also be used to resume a run of irace that was interrupted before completion.

References

[1] Acosta-Mesa H-G, Rechy-Ramírez F, Mezura-Montes E, Cruz-Ramírez N, Jiménez RH. Application of time series discretization using evolutionary programming for classification of precancerous cervical lesions. J Biomed Inform 2014;49:73–83.
[2] Adenso-Díaz B, Laguna M. Fine-tuning of algorithms using fractional experimental design and local search. Oper Res 2006;54(1):99–114.
[3] Ansótegui C, Sellmann M, Tierney K. A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent IP, editor. Principles and practice of constraint programming, CP 2009, volume 5732 of lecture notes in computer science. Heidelberg, Germany: Springer; 2009. p. 142–57.
[4] Ansótegui C, Malitsky Y, Samulowitz H, Sellmann M, Tierney K. Model-based genetic algorithms for algorithm configuration. In: Yang Q, Wooldridge M, editors. Proceedings of the twenty-fourth international joint conference on artificial intelligence (IJCAI-15). Menlo Park, CA: IJCAI/AAAI Press; 2015. p. 733–9.
[5] Audet C, Orban D. Finding optimal algorithmic parameters using derivative-free optimization. SIAM J Optim 2006;17(3):642–64.
[6] Audet C, Dang C-K, Orban D. Algorithmic parameter optimization of the DFO method with the OPAL framework. In: Naono K, Teranishi K, Cavazos J, Suda R, editors. Software automatic tuning: from concepts to state-of-the-art results. Springer; 2010. p. 255–74.
[7] Aydın D. Composite artificial bee colony algorithms: from component-based analysis to high-performing algorithms. Appl Soft Comput 2015;32:266–85.
[8] Babić D, Hu AJ. Structural abstraction of software verification conditions. In: Computer aided verification: 19th international conference, CAV 2007; 2007. p. 366–78.
[9] Babić D, Hutter F. Spear theorem prover. SAT'08: proceedings of the SAT 2008 race; 2008.
[10] Balaprakash P, Birattari M, Stützle T. Improvement strategies for the F-race algorithm: sampling design and iterative refinement. In: Bartz-Beielstein T, Blesa MJ, Blum C, Naujoks B, Roli A, Rudolph G, Sampels M, editors. Hybrid metaheuristics, volume 4771 of lecture notes in computer science. Heidelberg, Germany: Springer; 2007. p. 108–22.
[11] Bartz-Beielstein T. Experimental research in evolutionary computation: the new experimentalism. Berlin, Germany: Springer; 2006.
[12] Bartz-Beielstein T, Lasarczyk C, Preuss M. Sequential parameter optimization. In: Proceedings of the 2005 congress on evolutionary computation (CEC 2005). Piscataway, NJ: IEEE Press; 2005. p. 773–80.
[13] Battiti R, Brunato M, Mascia F. Reactive search and intelligent optimization, volume 45 of operations research/computer science interfaces. New York, NY: Springer; 2008.

[14] Benavides AJ, Ritt M. Iterated local search heuristics for minimizing total completion time in permutation and non-permutation flow shops. In: Brafman RI, Domshlak C, Haslum P, Zilberstein S, editors. Proceedings of the twenty–fifth international conference on automated planning and scheduling, ICAPS, Jerusalem, Israel, June 7–11. Menlo Park, CA: AAAI Press; 2015. p. 34–41.
[15] Bezerra LCT, López-Ibáñez M, Stützle T. Deconstructing multi-objective evolutionary algorithms: an iterative analysis on the permutation flowshop. In: Pardalos PM, Resende MGC, Vogiatzis C, Walteros JL, editors. Learning and intelligent optimization, 8th international conference, LION 8, volume 8426 of lecture notes in computer science. Heidelberg, Germany: Springer; 2014a. p. 57–172.
[16] Bezerra LCT, López-Ibáñez M, Stützle T. Automatic design of evolutionary algorithms for multi-objective combinatorial optimization. In: Bartz-Beielstein T, Branke J, Filipič B, Smith J, editors. PPSN 2014, volume 8672 of lecture notes in computer science. Heidelberg, Germany: Springer; 2014b. p. 508–17.
[17] Bezerra LCT, López-Ibáñez M, Stützle T. Automatic component-wise design of multi-objective evolutionary algorithms. IEEE Trans Evol Comput 2016;20(3):403–17.
[18] Birattari M. The race package for R: racing methods for the selection of the best. Technical Report TR/IRIDIA/2003-037. IRIDIA, Université Libre de Bruxelles, Belgium; 2003.
[19] Birattari M. Tuning metaheuristics: a machine learning perspective, volume 197 of studies in computational intelligence. Berlin/Heidelberg, Germany: Springer; 2009.
[20] Birattari M, Stützle T, Paquete L, Varrentrapp K. A racing algorithm for configuring metaheuristics. In: Langdon WB, editor. Proceedings of the genetic and evolutionary computation conference, GECCO 2002. San Francisco, CA: Morgan Kaufmann Publishers; 2002. p. 11–18.
[21] Birattari M, Yuan Z, Balaprakash P, Stützle T. F-race and iterated F-race: an overview. In: Bartz-Beielstein T, Chiarandini M, Paquete L, Preuss M, editors. Experimental methods for the analysis of optimization algorithms. Berlin, Germany: Springer; 2010. p. 311–36.
[22] Bischl B., Lang M., Bossek J., Judt L., Richter J., Kuehn T., et al. mlr: machine learning in R. 2013. http://cran.r-project.org/package=mlr. R package.
[23] Blum C, Calvo B, Blesa MJ. FrogCOL and frogMIS: new decentralized algorithms for finding large independent sets in graphs. Swarm Intell 2015;9(2–3):205–27.
[24] Blum C, Pinacho P, López-Ibáñez M, Lozano JA. Construct, merge, solve & adapt: a new general algorithm for combinatorial optimization. Comput Oper Res 2016;68:75–88.
[25] Ceschia S, Schaerf A, Stützle T. Local search techniques for a routing-packing problem. Comput Ind Eng 2013;66(4):1138–49.
[26] Chen Y, Hao J-K, Glover F. A hybrid metaheuristic approach for the capacitated arc routing problem. Eur J Oper Res 2016;553(1):25–39.
[27] Chivilikhin DS, Ulyantsev VI, Shalyto AA. Modified ant colony algorithm for constructing finite state machines from execution scenarios and temporal formulas. Autom Remote Control 2016;77(3):473–84.
[28] Conover WJ. Practical nonparametric statistics, third edition. New York, NY: John Wiley & Sons; 1999.
[29] Coy SP, Golden BL, Runger GC, Wasil EA. Using experimental design to find effective parameter settings for heuristics. J Heuristics 2001;7(1):77–97.
[30] Dell'Amico M, Iori M, Novellani S, Stützle T. A destroy and repair algorithm for the bike sharing rebalancing problem. Comput Oper Res 2016;71:146–62.
[31] Dubois-Lacoste J, López-Ibáñez M, Stützle T. Automatic configuration of state-of-the-art multi-objective optimizers using the TP+PLS framework. In: Krasnogor N, Lanzi PL, editors. Proceedings of the genetic and evolutionary computation conference, GECCO 2011. New York, NY: ACM Press; 2011. p. 2019–26.
[32] Fawcett C, Hoos HH. Analysing differences between algorithm configurations through ablation. In: Proceedings of MIC 2013, the 10th metaheuristics international conference; 2013. p. 123–32.
[33] Fisset B, Dhaenens C, Jourdan L. MO-Mine$_{clust}$: a framework for multi-objective clustering. In: Haenens C, Jourdan L, Marmion M-E, editors. Learning and intelligent optimization, 9th international conference, LION 9, volume 8994 of lecture notes in computer science. Heidelberg, Germany: Springer; 2015. p. 293–305.
[34] Francesca G, Brambilla M, Brutschy A, Trianni V, Birattari M. AutoMoDe: a novel approach to the automatic design of control software for robot swarms. Swarm Intell 2014;8(2):89–112.
[35] Francesca G, Brambilla M, Brutschy A, Garattoni L, Miletitch R, Podevijn G, et al. AutoMoDe-chocolate: automatic design of control software for robot swarms. Swarm Intell 2015.
[36] Goldberg DE. Genetic algorithms in search, optimization and machine learning. Boston, MA, USA: Addison-Wesley; 1989.
[37] Grefenstette JJ. Optimization of control parameters for genetic algorithms. IEEE Trans Syst Man Cybern 1986;16(1):122–8.
[38] Hansen N, Ostermeier A. Completely derandomized self-adaptation in evolution strategies. Evol Comput 2001;9(2):159–95.
[39] Herrera F., Lozano M., Molina D.. Test suite for the special issue of soft computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems. http://sci2s.ugr.es/eamhco/. 2010.
[40] Hoos HH. Programming by optimization. Commun ACM 2012;55(2):70–80.
[41] Hutter F, Hoos HH, Leyton-Brown K, Stützle T. ParamILS: an automatic algorithm configuration framework. J Artif Intell Res 2009;36:267–306.
[42] Hutter F, Hoos HH, Leyton-Brown K. Automated configuration of mixed integer programming solvers. In: Lodi A, Milano M, Toth P, editors. Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, 7th international conference, CPAIOR 2010, volume 6140 of lecture notes in computer science. Heidelberg, Germany: Springer; 2010. p. 186–202.
[43] Hutter F, Hoos HH, Leyton-Brown K. Sequential model-based optimization for general algorithm configuration. In: Coello Coello CA, editor. Learning and intelligent optimization, 5th international conference, LION 5, volume 6683 of lecture notes in computer science. Heidelberg, Germany: Springer; 2011. p. 507–23.
[44] Hutter F, López-Ibáñez M, Fawcett C, Lindauer MT, Hoos HH, Leyton-Brown K, et al. AClib: a benchmark library for algorithm configuration. In: Pardalos PM, Resende MGC, Vogiatzis C, Walteros JL, editors. Learning and intelligent optimization, 8th international conference, LION 8, volume 8426 of lecture notes in computer science. Heidelberg, Germany: Springer; 2014. p. 36–40.
[45] IBM. ILOG CPLEX optimizer. http://www.ibm.com/software/integration/optimization/cplex-optimizer/.
[46] Jackson CH. Multi-state models for panel data: the msm package for R. J Stat Softw 2011;38(8):1–29. http://www.jstatsoft.org/v38/i08/.
[47] Jacquin S, Jourdan L, Talbi E-G. Dynamic programming based metaheuristic for energy planning problems. In: Esparcia-Alcázar AI, Mora AM, editors. Applications of evolutionary computation, volume 8602 of lecture notes in computer science. Heidelberg, Germany: Springer; 2014. p. 165–76.
[48] Johnson DS, McGeoch LA. The traveling salesman problem: a case study in local optimization. In: Aarts EHL, Lenstra JK, editors. Local search in combinatorial optimization. Chichester, UK: John Wiley & Sons; 1997. p. 215–310.
[49] Johnson DS, McGeoch LA. Experimental analysis of heuristics for the STSP. In: Gutin G, Punnen A, editors. The traveling salesman problem and its variations. Dordrecht, The Netherlands: Kluwer Academic Publishers; 2002. p. 369–443.
[50] Karafotias G, Hoogendoorn M, Eiben AE. Parameter control in evolutionary algorithms: trends and challenges. IEEE Trans Evol Comput 2015;19(2):167–87.
[51] KhudaBukhsh AR, Xu L, Hoos HH, Leyton-Brown K. SATenstein: automatically building local search SAT solvers from components. In: Boutilier C, editor. Proceedings of the twenty-first international joint conference on artificial intelligence (IJCAI-09). Menlo Park, CA: AAAI Press; 2009. p. 517–24.
[52] Lacroix B, Molina D, Herrera F. Dynamically updated region based memetic algorithm for the 2013 CEC special session and competition on real parameter single objective optimization. In: Proceedings of the 2013 congress on evolutionary computation (CEC 2013). Piscataway, NJ: IEEE Press; 2013. p. 1945–51.
[53] Lacroix B, Molina D, Herrera F. Region based memetic algorithm for real-parameter optimisation. Inf Sci 2014;262:15–31.
[54] Lang M, Kotthaus H, Marwedel, Weihs C, Rahnenführer J, Bischl B. Automatic model selection for high-dimensional survival analysis. J Stat Comput Simul 2014;85(1):62–76.
[55] Liao T, Stützle T. Benchmark results for a simple hybrid algorithm on the CEC 2013 benchmark set for real-parameter optimization. In: Proceedings of the 2013 congress on evolutionary computation (CEC 2013). Piscataway, NJ: IEEE Press; 2013. p. 1938–44.
[56] Liao T, Montes de Oca MA, Stützle T. Computational results for an automatically tuned CMA-ES with increasing population size on the CEC'05 benchmark set. Soft Comput 2013;17(6):1031–46.
[57] Liao T, Molina D, Stützle T. Performance evaluation of automatically tuned continuous optimizers on different benchmark sets. Appl Soft Comput 2015;27:490–503.
[58] López-Ibáñez M, Stützle T. The automatic design of multi-objective ant colony optimization algorithms. IEEE Trans Evol Comput 2012;16(6):861–75.
[59] López-Ibáñez M, Stützle T. Automatically improving the anytime behaviour of optimisation algorithms. Eur J Oper Res 2014;235(3):569–82.
[60] López-Ibáñez M, Blum C, Ohlmann JW, Thomas BW. The travelling salesman problem with time windows: adapting algorithms from travel-time to makespan optimization. Appl Soft Comput 2013;13(9):3806–15.
[61] López-Ibáñez M., Dubois-Lacoste J., Pérez Cáceres L., Stützle T., Birattari M.. 2016a. The irace package: Iterated racing for automatic algorithm configuration. http://iridia.ulb.ac.be/supp/IridiaSupp2016-003/.
[62] López-Ibáñez M, Pérez Cáceres L, Dubois-Lacoste J, Stützle T, Birattari M. The irace package: user guide. Technical Report TR/IRIDIA/2016-004. IRIDIA, Université Libre de Bruxelles, Belgium; 2016b. http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2016-004.pdf.
[63] Marmion M-E, Mascia F, López-Ibáñez M, Stützle T. Automatic design of hybrid stochastic local search algorithms. In: Blesa MJ, Blum C, Festa P, Roli A, Sampels M, editors. Hybrid metaheuristics, volume 7919 of lecture notes in computer science. Heidelberg, Germany: Springer; 2013. p. 144–58.
[64] Maron O, Moore AW. The racing algorithm: model selection for lazy learners. Artif Intell Res 1997;11(1–5):193–225.
[65] Mascia F, Birattari M, Stützle T. Tuning algorithms for tackling large instances: an experimental protocol. In: Pardalos P, Nicosia G, editors. Learning and intelligent optimization, 7th international conference, LION 7, volume 7997 of lecture notes in computer science. Heidelberg, Germany: Springer; 2013. p. 410–22.
[66] Mascia F, López-Ibáñez M, Dubois-Lacoste J, Stützle T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Comput Oper Res 2014;51:190–9.

[67] Massen F, López-Ibáñez M, Stützle T, Deville Y. Experimental analysis of pheromone-based heuristic column generation using irace. In: Blesa MJ, Blum C, Festa, P, Roli A, Sampels M, editors. Hybrid metaheuristics, volume 7919 of lecture notes in computer science. Heidelberg, Germany: Springer; 2013. p. 92–106.

[68] Meier J.F., Clausen U.. 2014. A versatile heuristic approach for generalized hub location problems. Preprint, Provided upon personal request.

[69] Mesquita RG, Silva RM, Mello CA, Miranda PB. Parameter tuning for document image binarization using a racing algorithm. Expert Syst Appl 2015;42(5):2593–603.

[70] Miranda P, Silva RM, Prudêncio RB. Fine-tuning of support vector machine parameters using racing algorithms. In: 22st European symposium on artificial neural networks, computational intelligence And machine learning, Bruges. ESANN; 2014. p. 325–30.

[71] Montes de Oca MA, Aydın D, Stützle T. An incremental particle swarm for large-scale continuous optimization problems: an example of tuning-in-the-loop (re)design of optimization algorithms. Soft Comput 2011;15(11):2233–55.

[72] Mühlenthaler M. Fairness in academic course timetabling. Springer; 2015.

[73] Nannen V, Eiben AE. A method for parameter calibration and relevance estimation in evolutionary algorithms. In: Cattolico M, editor. Proceedings of the genetic and evolutionary computation conference, GECCO 2006. New York, NY: ACM Press; 2006. p. 183–90.

[74] Nannen V, Eiben AE. Relevance estimation and value calibration of evolutionary algorithm parameters. In: Veloso MM, editor. Proceedings of the twentieth international joint conference on artificial intelligence (IJCAI-07). Menlo Park, CA: AAAI Press; 2007. p. 975–80.

[75] Nashed YSG, Mesejo P, Ugolotti R, Dubois-Lacoste J, Cagnoni S. A comparative study of three GPU-based metaheuristics. In: Coello Coello CA, editor. PPSN 2012, part II, volume 7492 of lecture notes in computer science. Springer, Heidelberg, Germany; 2012. p. 398–407.

[76] Pellegrini P, Birattari M, Stützle T. A critical analysis of parameter adaptation in ant colony optimization. Swarm Intell 2012a;6(1):23–48.

[77] Pellegrini P, Castelli L, Pesenti R. Metaheuristic algorithms for the simultaneous slot allocation problem. IET Intell Transport Syst 2012b;6(4):453–62.

[78] Pérez Cáceres L, López-Ibáñez M, Stützle T. An analysis of parameters of irace. In: Blum C, Ochoa G, editors. Proceedings of EvoCOP 2014–14th European conference on evolutionary computation in combinatorial optimization, volume 8600 of lecture notes in computer science. Heidelberg, Germany: Springer; 2014. p. 37–48.

[79] Powell M. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical Report Cambridge NA Report NA2009/06. University of Cambridge, UK; 2009.

[80] Ridge E, Kudenko D. Tuning the performance of the MMAS heuristic. In: Stützle T, Birattari M, Hoos HH, editors. International workshop on engineering stochastic local search algorithms (SLS 2007), volume 4638 of lecture notes in computer science. Heidelberg, Germany: Springer; 2007. p. 46–60.

[81] Riff M-C, Montero E. A new algorithm for reducing metaheuristic design effort. In: Proceedings of the 2013 congress on evolutionary computation (CEC 2013). Piscataway, NJ: IEEE Press; 2013. p. 3283–90.

[82] Robert CP. Simulation of truncated normal variables. Stat Comput 1995;5(2):121–5.

[83] Ruiz R, Maroto C. A comprehensive review and evaluation of permutation flow-shop heuristics. Eur J Oper Res 2005;165(2):479–94.

[84] Samà M, Pellegrini P, Ariano AD, Rodriguez J, Pacciarelli D. Ant colony optimization for the real-time train routing selection problem. Transp Res Part B 2016;85:89–108.

[85] Schneider M, Hoos HH. Quantifying homogeneity of instance sets for algorithm configuration. In: Hamadi Y, Schoenauer M, editors. Learning and intelligent optimization, 6th international conference, LION 6, volume 7219 of lecture notes in computer science. Heidelberg, Germany: Springer; 2012. p. 190–204.

[86] Stefanello F, Aggarwal V, Buriol LS, Gonçalves JF, Resende MGC. A biased random-key genetic algorithm for placement of virtual machines across geo-separated data centers. In: Silva S, Esparcia-Alcázar AI, editors. Proceedings of the genetic and evolutionary computation conference, GECCO 2015. New York, NY: ACM Press; 2015. p. 919–26.

[87] Stützle T. ACOTSP: a software package of various ant colony optimization algorithms applied to the symmetric traveling salesman problem. 2002. http://www.aco-metaheuristic.org/aco-code/.

[88] Styles J, Hoos HH. Ordered racing protocols for automatically configuring algorithms for scaling performance. In: Blum C, Alba E, editors. Proceedings of the genetic and evolutionary computation conference, GECCO 2013. New York, NY: ACM Press; 2013. p. 551–8.

[89] Thornton C, Hutter F, Hoos HH, Leyton-Brown K. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: Dhillon IS, Koren Y, Ghani R, Senator TE, Bradley P, Parekh R, He J, Grossman RL, Uthurusamy R, editors. The 19th ACM SIGKDD international conference on knowledge discovery and data mining, KDD 2013. New York, NY: ACM Press; 2013. p. 847–55.

[90] Violin A. Mathematical programming approaches to pricing problems. Faculté de Sciences, Université Libre de Bruxelles and Dipartimento di Ingegneria e Architettura, Università degli studi di Trieste; 2014. Phd thesis.

[91] Wessing S, Beume N, Rudolph G, Naujoks B. Parameter tuning boosts performance of variation operators in multiobjective optimization. In: Schaefer R, Cotta C, Kolodziej J, Rudolph G, editors. Parallel problem solving from nature, PPSN XI, volume 6238 of lecture notes in computer science. Heidelberg, Germany,: Springer; 2010. p. 728–37.

[92] Yarimcam A, Asta S, Ozcan E, Parkes AJ. Heuristic generation via parameter tuning for online bin packing. In: Evolving and autonomous learning systems (EALS), 2014 IEEE symposium on. IEEE; 2014. p. 102–8.

[93] Yuan Z, Montes de Oca MA, Stützle T, Birattari M. Continuous optimization algorithms for tuning real and integer algorithm parameters of swarm intelligence algorithms. Swarm Intell 2012;6(1):49–75.

[94] Yuan Z, Montes de Oca MA, Stützle T, Lau HC, Birattari M. An analysis of post-selection in automatic configuration. In: Blum C, Alba E, editors. Proceedings of GECCO 2013. New York, NY: ACM Press; 2013. p. 1557–64.

[95] Zitzler E, Thiele L, Laumanns M, Fonseca CM, Fonseca VGd. Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans Evol Comput 2003;7(2):117–32.

[96] Zlochin M, Birattari M, Meuleau N, Dorigo M. Model-based search for combinatorial optimization: acritical survey. Ann Oper Res 2004;131(1–4):373–95.