The `irace` Package: Iterated Race for Automatic Algorithm Configuration

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari

IRIDIA – Technical Report Series

Technical Report No. TR/IRIDIA/2011-004

February 2011

IRIDIA – Technical Report Series
ISSN 1781-3794

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

Technical report number TR/IRIDIA/2011-004

The irace Package: Iterated Race for Automatic Algorithm Configuration

Manuel López-Ibáñez manuel.lopez-ibanez@ulb.ac.be
Jérémie Dubois-Lacoste jeremie.dubois-lacoste@ulb.ac.be
Thomas Stützle stuetzle@ulb.ac.be
Mauro Birattari mbiro@ulb.ac.be

IRIDIA, CoDE, Université Libre de Bruxelles, Brussels, Belgium
Contact: irace@iridia.ulb.ac.be

February 2011

Abstract

The program `irace` implements the *iterated racing* procedure, which is an extension of the Iterated F-race procedure (I/F-Race). Its main purpose is to automatically configure optimization algorithms by finding the most appropriate settings given a set of instances of an optimization problem. It builds upon the `race` package by Birattari and it is implemented in R. This revision documents `irace` version 0.9. The latest version of the `irace` software is available at `http://iridia.ulb.ac.be/irace`.

1 Introduction

The program `irace` implements the *iterated racing* procedure, which is an extension of the Iterated F-race procedure (I/F-Race) proposed by Balaprakash et al. [4] and later developed by Birattari et al. [8]. `irace` is implemented in R [23] and it builds upon the `race` package by Birattari [5, 6]. The main purpose of `irace` is to *automatically configure optimization algorithms* by finding the most appropriate settings given a set of instances of an optimization problem. Many optimization algorithms have a number of configurable parameters, and their performance on a particular problem often depends on the particular settings of these parameters. This is particularly true in metaheuristics, that is, general-purpose solvers such as ant colony optimization, evolutionary algorithms and other stochastic local search methods [6, 12]. Moreover, complex algorithms may be automatically designed by finding a particular configuration of a framework composed by alternative design choices and complementary algorithmic components [17, 19].

The scenario that `irace` addresses is usually described as *offline configuration* [6], that is, there are two clearly delimited phases. In a primary tuning phase, an algorithm configuration is chosen given a set of tuning 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 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.

Offline tuning is frequently performed ad-hoc by algorithm designers as a result of trial-and-error runs. More formal methods involve the use of experimental design techniques [1], evolutionary algorithms [22] and local search [15]. Birattari [6] proposed an automatic configuration approach, F-Race, based on *racing* and Friedman's non-parametric two-way analysis of variance by ranks. This proposal was later improved by performing several iterations of F-Race that successively refine a probabilistic model of the parameter space. The resulting automatic configuration approach was called Iterated F-race (I/F-Race) [4, 8]. Although a formal description of the I/F-Race procedure is given in the original publications, no implementation of it has been publicly available. The `irace` package implements a general *iterated racing* procedure, which includes, as a special case, I/F-Race.

The `irace` software presented here has already been extensively tested in several research projects. Dubois-Lacoste et al. [10, 11] used `irace` for tuning the parameters of several iterated greedy (IG) variants for various objectives in the permutation flow-shop problem (PFSP). López-Ibáñez and Stützle [18] automatically configured a flexible ant colony optimization framework for the bi-objective travelling salesman problem (bTSP). Montes de Oca et al. [21] designed an incremental particle swarm optimization algorithm for large-scale continuous problems by means of automatic configuration. These examples demonstrate the potential of automatic configuration of algorithms, and the profound impact it will have on the way optimisation algorithms are designed and evaluated in the future.

2 Automatic Configuration and I/F-Race

2.1 The algorithm configuration problem

Birattari [6] provides a more general and comprehensive definition of the automatic configuration problem. For brevity, we will restrict here to a simpler definition that is sufficient for our purposes.

A parametrized algorithm is an 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 θ is a unique assignment of values to parameters $\theta = \{x_1, \ldots, x_{N^{\text{param}}}\}$, and Θ denotes the possibly infinite set of all configurations of the algorithm.

When considering a problem to be solved by this parametrized algorithm, the set of possible instances of the problem may be seen as a set $\mathcal{I}$ from which instances to be solved are sampled. We are also given a cost measure $\mathcal{C}\colon \Theta \times \mathcal{I} \to \mathbb{R}$, where $\mathcal{C}(\theta, i)$ measures the quality of configuration $\theta \in \Theta$ on instance $i \in \mathcal{I}$. Since the algorithm may be stochastic, this cost measure is often a random variable and the value $c(\theta, i)$ is a realization of the random variable $\mathcal{C}(\theta, i)$. The cost value may be the best objective function value found within a given computation time, or, perhaps, the deviation from the optimum value if the latter is known. In the case of decision problems, it may correspond to the computation required to reach a decision, possibly bounded by a maximum cut-off time. In any case, the cost measure assigns a quality value to one run of a particular configuration on a particular instance. The criterion that we want to optimize when configuring an algorithm for a problem is a function of the cost of a configuration over the whole set of instances. In that sense, it is a statistical parameter c_θ defined on the conditional distribution $\mathcal{C}(\theta) = \mathcal{C}(\mathcal{I} \mid \theta)$. We can now formally define the automatic configuration problem as finding the best configuration θ^* such that:

$$\theta^* = \arg\min_{\theta \in \Theta} c_\theta \quad . \tag{1}$$

The precise value of c_θ is generally unknown, and it can only be estimated by sampling from $\mathcal{C}(\theta)$. The particular definition of c_θ determines how to rank the configurations over a set of instances. The most typical definition of c_θ is $E[\mathcal{C}(\theta)]$, the expected cost value or mean cost. If the cost values over different instances are incommensurable, the median or the sum of ranks may be more meaningful.

2.2 Classes of parameters

There are two main classes of parameters: *categorical* and *numerical* parameters. Categorical parameters represent discrete values without any implicit order or sensible distance measure, whereas numerical parameters have an implicit order of their values. There are also seemingly categorical parameters but with an implicit order or distance measure of their values. An example would be a parameter with three values {`low`, `medium`, `high`}. Such parameters are called *ordinal* and are typically handled as numerical parameters [8]. Finally, parameters may be *conditional* on other parameters, that is, they are only enabled when the values of other parameters satisfy a particular condition. For instance, the tabu list length only matters if tabu search is chosen as the local search algorithm. 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 are either disabled or they have a value according to a predefined range. In some cases, parameter constraints may be modeled by replacing one of the parameters by a surrogate parameter, e.g., $a' \in (0, 1)$, such that $a = a' \cdot b$.

2.3 Racing

Racing was first proposed in machine learning to deal with the problem of model selection [20]. Birattari [6] adapted the procedure for the configuration of optimization algorithms. A race starts with a finite set of candidate configurations. At each step of the race, the candidate configurations are evaluated on a single instance. After each step, those candidate configurations that perform statistically worse than the rest are discarded, and the race continues with the remaining *surviving* configurations. This procedure continues until reaching a minimum number of surviving configurations, a maximum number of instances or a pre-defined computational budget. This computational budget may be an overall computation time or a number of experiments.

There are several alternatives for selecting which configurations should be discarded during the race. The F-Race algorithm [6, 7] relies on the non-parametric Friedman's two-way analysis of variance by ranks, the Friedman test, and its associated post-hoc test, as described by Conover [9]. Nonetheless, the `race` package [5] implements various alternatives based on the paired t-test with and without p-value correction for multiple comparisons, which are also available in the proposed `irace` package.

2.4 I/F-Race

The racing approach is particularly effective if the initial set of configurations is exhaustive. However, it is often the case in practice that the configuration space is too large to be sampled effectively. Moreover, the knowledge gathered during the race itself could be used to sample new candidate configurations.

I/F-Race was proposed [4, 8] as an extension of F-Race in order to search the configuration space more effectively by focusing on the most promising configurations. I/F-Race iterates over a sequence of races, and the candidate configurations evaluated in each subsequent race depend on the results of previous races. Candidate configurations are generated by sampling according to some probability model P_X defined over the parameter space X. Without prior knowledge, the sampling is initially done according to a uniform distribution. In subsequent iterations, numerical parameters are sampled according to a normal distribution, whereas categorical parameters are sampled according to a discrete probability function. The `irace` software that we propose in the next section is an implementation of iterated racing that includes I/F-Race as a special case.

Algorithm 1 Iterated Racing

```
Require: $\{I_1, I_2, \dots, \} \sim \mathcal{I}$,
    parameter space: $X$,
    cost measure: $\mathcal{C} \colon \Theta \times \mathcal{I} \to \mathbb{R}$,
    tuning budget: $B$
 1: $\Theta_1 \sim \mathsf{SampleUniform}(X)$
 2: $\Theta^{\text{elite}} := \mathsf{Race}(\Theta_1,\, B_1)$
 3: $i := 2$
 4: while $B_{\text{used}} \leq B$ do
 5:    $\Theta^{\text{new}} \sim \mathsf{Sample}(X, \Theta^{\text{elite}})$
 6:    $\Theta_i := \Theta^{\text{new}} \cup \Theta^{\text{elite}}$
 7:    $\Theta^{\text{elite}} := \mathsf{Race}(\Theta_i,\, B_i)$
 8:    $i := i + 1$
 9:    if $i > N^{\text{iter}}$ then
10:       $N^{\text{iter}} := i$
11:    end if
12: end while
13: Output: $\Theta^{\text{elite}}$
```

3 Iterated Racing

In this section, we describe the exact implementation of iterated race as proposed in the `irace` package. Implementation details and setup of the `irace` package itself are given in the next section.

An outline of the iterated racing algorithm is given in Algorithm 1. It requires as input: a set of instances, a parameter space, a cost function, and a tuning budget (B).

Iterated race requires an estimation of the number of iterations N^{iter} (races) that it will execute. The default setting of N^{iter} depends on the number of parameters with $N^{\text{iter}} = \lfloor 2 + \log_2 N^{\text{param}} \rfloor$. Each iteration performs one race with a limited computation budget $B_i = (B - B_{\text{used}})/(N^{\text{iter}} - i + 1)$, where $i = 1, \dots, N^{\text{iter}}$. Each race starts from a set of candidate configurations Θ_i. The number of candidate configurations is calculated as $|\Theta_i| = N_i = \lfloor B_i/(\mu + i) \rfloor$. Thus, the number of candidate configurations decreases with the number of iterations, which means that more evaluations per configuration will be performed in later iterations. Moreover, increasing the value of μ also increases the number of evaluations per candidate configuration.

In the first iteration, the initial set of candidate configurations is generated by uniformly sampling the parameter space X. When a race starts, each configuration is evaluated on the first instance by means of the, possibly stochastic, 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 the rest, the worst configurations are removed from the race, while the rest, the *surviving* candidates, are run on the next instance. A new statistical test is performed every T^{each} instances. The race continues until the budget of the current iteration is not enough to test all remaining candidate configurations on a new instance ($B_i < N^{\text{surv}}$), or when at most N^{min} configurations remain, $N^{\text{surv}} \leq N^{\text{min}}$.

At the end of a race, a number $N_i^{\text{elite}} = \min(N^{\text{surv}}, N^{\text{min}})$ of elite configurations Θ^{elite} are selected from the surviving configurations, by ranking them according to the sum of ranks or the mean cost, depending on which statistical test is used during the race. Then, a weight w_z is assigned to each elite configuration θ_z, $z = 1, \ldots, N^{\text{elite}}$, according to their ranks r_z:

$$w_z = \frac{N_i^{\text{elite}} - r_z + 1}{N_i^{\text{elite}} \cdot (N^{\text{elite}} + 1)/2} \quad . \tag{2}$$

This weight w_z is used for generating new candidate configurations for the next race.

In the next race, a number of $N_i^{\text{new}} = N_i - N_i^{\text{elite}}$ new candidate configurations are generated. For generating a new configuration, first one parent configuration is sampled from the set of elite configurations $\theta_z \sim \Theta^{\text{elite}}$ with a probability proportional to its weight w_z. Next, a new value is sampled for each parameter X_d, $d = 1, \ldots, N^{\text{param}}$, according to a distribution that its associated to each parameter of θ_z. Parameters are considered in the order determined by the dependency graph of conditions, that is, unconditional parameters are sampled first, those parameters that are conditional on them are sampled next if the condition is satisfied, and so on. Moreover, 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 of $X_d \in [\underline{x}_d, \overline{x}_d]$, then a new value is sampled from the truncated normal distribution $\mathcal{N}(x_d^z, (\overline{x}_d - \underline{x}_d) \cdot \sigma_d^i)$, such that the new value is within the range.[1] The mean of the distribution x_d^z is simply the value of parameter d in the elite configuration θ_z. The parameter σ_d^i controlling the standard deviation of the distribution is initially set to 1, and it is updated at each iteration before sampling as follows:

$$\sigma_d^i := \sigma_d^{i-1} \cdot (N_i^{\text{new}})^{\frac{1-i}{N^{\text{param}}}} \tag{3}$$

If the numerical parameter is of integer type, we round the sampled value to the nearest integer. Parameters of ordinal type are encoded as integers.

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

$$\mathcal{P}^{i,z}(X_d = x_j) := \mathcal{P}^{i-1,z}(X_d = x_j) \cdot \left(1 - \frac{i-1}{N^{\text{iter}}}\right) + \Delta\mathcal{P} \tag{4}$$

where

$$\Delta\mathcal{P} = \begin{cases} \dfrac{i-1}{N^{\text{iter}}} & \text{if } x_j = x_z \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Finally, the new configurations generated after sampling inherit the probability distributions from their parents, and a new race is launched with the union of the new configurations and the elite configurations.

The algorithm stops if the budget is exhausted ($B_{\text{used}} > B$) or if the number of candidate configurations to be generated at the start of an iteration is not greater than the number of elites ($N_i \leq N_i^{\text{elite}}$), since in that case no new configurations would be generated. If the iteration counter i reaches the estimated number of iterations N^{iter}, we simply adjust the number of iterations accordingly and continue.

4 Extensions

We have implemented some extensions that were not proposed in the original publications.

[1]For sampling from a truncated normal distribution, we use the `msm` package [16].

4.1 Initial configurations

We can seed the iterated race procedure with a set of initial configurations. In that case, only enough configurations are sampled to reach N_1 in total. For how to specify the initial set of configurations, see option `candidatesFile`.

4.2 Tuning for computation time

Sometimes the goal is to minimize the computation time required to perform a task. For example, in decision problems, the goal is to minimize the time that an algorithm takes to reach an answer. In the case of complete algorithms that optimally solve an optimization problem, the goal is to minimize the time to reach the optimal solution.

When automatically configuring an algorithm for minimizing computation time, it is often preferred to define the tuning budget B as a computation time limit, instead of a maximum number of experiments. In such case, the budget of each iteration is computed as $B_i = (B - B_{\text{used}})/(\hat{t}_i \cdot (N^{\text{iter}} - i + 1))$, where $i = 1, \ldots, N^{\text{iter}}$, and $\hat{t}_i$ is an estimate of the computation time required by a single run of the algorithm. The initial value of this estimate is provided by the user, for example, as a cut-off time after which an algorithm run is stopped even if no decision has been reached. In subsequent iterations, $\hat{t}$ is updated by computing the average time used by the experiments performed so far.

4.3 Soft-restart

Our implementation incorporates a "soft-restart" mechanism to avoid premature convergence. In the original I/F-Race proposal [4], 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 configurations. However, if the tuning converges to a few, very similar elite configurations in few iterations, the diversity is lost and new candidate configurations are very similar. Such a premature convergence wastes the remaining budget on repeatedly evaluating very similar configurations, without exploring new alternatives.

We implemented a "soft-restart" mechanism that checks for premature convergence after generating each new set of candidate configurations. If premature convergence has occurred, the probabilistic model is partially reinitialized. We consider that there is premature convergence when the "distance" between two candidate configurations is zero. The distance between two configurations is defined as 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 (integral or real), the distance is the absolute normalized difference between their values (the values are rounded up to the number of significant digits specified by the option `signifDigits`) ;
  - 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 probabilistic model. This reinitialization is applied only to the elite configurations that were used to generate the candidate configurations with zero distance, because the other elite configurations may not suffer from premature convergence, and it would be detrimental to restart their probabilistic models. In the case of categorical parameters, the discrete probability model of elite configuration z, $\mathcal{P}^{i,z}(X_d)$, is adjusted by modifying each individual probability value $p \in \mathcal{P}^{i,z}(X_d)$ with respect to the maximum value $p_{\max} = \max\{\mathcal{P}^{i,z}(X_d)\}$ as follows:

$$p := \frac{0.9 \cdot p + 0.1 \cdot p_{\max}}{\sum_{p' \in \mathcal{P}^{i,z}(X_d)} 0.9 \cdot p' + 0.1 \cdot p_{\max}} \; .$$

For numerical and ordinal parameters, the standard deviation of elite configuration z, $\sigma_d^{i,z}$, is “brought back” two iterations, with a maximum limit of its value in the second iteration, as follows:

$$\sigma_d^{i,z} := \min\{\sigma_d^{i,z} \cdot (N_i^{\text{new}})^{\frac{2}{N^{\text{param}}}}, (N_i^{\text{new}})^{\frac{-1}{N^{\text{param}}}}\}$$

After adjusting the probability model 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.

5 The `irace` software

This section describes the requirements, implementation details, and the setup of the `irace` package.

The scheme in Fig. 1 describes how the different parts of `irace` interact with each other. The program `irace` requires three main inputs:

1. A description of the parameter space X, which is provided by a file `parameterFile` that describes the parameters to configure, their types, ranges and constraints. The format of this file is explained in detail in Section 5.5.

2. The set of tuning instances $\{I_1, I_2, \ldots\}$, which in practice is a finite, hopefully representative, sample of $\mathcal{I}$. The particular options for specifying the set of tuning instances are given in Section 5.3.

3. The configuration of `irace` itself, which is defined by a number of options. These options may be defined in a configuration file or in the command line. Table 1 maps the description of iterated racing in the previous section to the configuration options in `irace`. The complete list of options is given in Appendix A.

An initial set of configurations may be provided by means of an optional candidates file (see option `candidatesFile`). The evaluation of candidate configurations is performed in two phases by means of two auxiliary programs called “hooks”. These hooks are described in Section 5.6. A step-by-step guide of setting up a new tuning scenario is provided in Section 5.7; examples of tuning scenarios are given in Section 6.

5.1 Requirements

- R version $\geq$ 2.10 (`http://www.r-project.org/`).

- The wrappers `irace` and `parallel-irace` require GNU Bash.

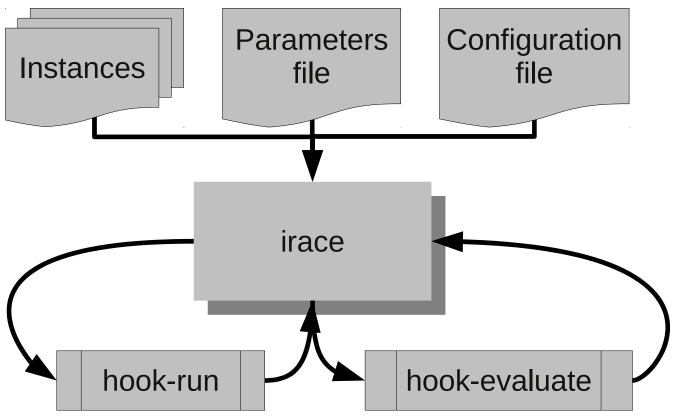

Figure 1: Scheme of `irace` flow of information.

Table 1: Configuration options of `irace` corresponding to the description of Iterated racing given in Section 3. The full list of options is available in Appendix A.

Iterated racing parameter	`irace` **configuration option**
B	either `maxNbExperiments` or `timeBudget`
$\mathcal{C}$ (cost measure)	`hookRun` and `hookEvaluate`, see also Section 5.6
N^{iter}	`nbIterations`
μ	`mu`
N^{min}	`minNbSurvival`
T^{first}	`firstTest`
T^{each}	`eachTest`
Statistical test	`testType`
$\hat{t}_1$	`timeEstimate`, see also Section 4.2

- The option `--parallel` requires the `multicore` package [25].
- The option `--mpi` requires the `Rmpi` package [26].
- Cluster-mode (`--sge-cluster 1`) requires Grid Engine commands `qsub` and `qstat`. The command `qsub` should return a message that contains the string: `"Your job JOBID"`. The command `"qstat -j JOBID"` returns nonzero if JOBID has finished, otherwise it should return zero.

5.2 Installation

Installation simply involves copying the files to a particular location, e.g., `/usr/irace/`. To invoke `irace`, just either add its path to the environment variable `$PATH`, or invoke it explicitly as `/usr/irace/irace`.

5.3 Tuning Instances

The set of tuning instances $\{I_1, I_2, \dots\}$, which in practice is a finite, hopefully representative, sample of $\mathcal{I}$, is read from `instanceFile`. The path given by option `instanceDir` will be prefixed to them, although these instances do not need to correspond to physical files. If the option `instanceFile` is not set, then `irace` considers all files found in `instanceDir`, and recursively in its subdirectories, as tuning instances. The order in which instances are considered by `irace` is randomized if the option `sampleInstances` is enabled. Otherwise, the order is the same given in the `instanceFile`, if this option is set, or in alphabetical order if there is no `instanceFile`.

5.4 Configuration File

The configurable parameters of `irace` can be defined in a configuration file or on the command-line. The full list of configurable parameters is given in Appendix A. If the same parameter is defined both in the configuration file and on the command-line, the value given on the command-line has preference. If a parameter is not set neither in the configuration file nor on the command-line, the default value is used.

The configuration file uses the R syntax, for example:

```
paramNumeric <- 1
paramString  <- "a string"
```

Moreover, anything in a line after the symbol '`#`' is ignored.

Many of the parameter exposed affect the internals of `irace` and we advise to not change their default values without fully understanding their effects.

5.5 Parameters file

The parameters of the algorithm to be tuned are described in a *parameter file*, see the example in `templates/parameters.tmpl`. Each line defines a configurable parameter:

```
<name> <switch> <type> <range> [ | <condition> ]
```

where each field is defined as follows:

`<name>`	The name of the parameter as an unquoted alphanumeric string, for instance `ants`.
`<switch>`	The command-line switch used to pass the value of this parameter to `hookRun`, for instance `"--ants "`. The value of the parameter is concatenated *without separator* to the switch string when invoking `hookRun`.
`<type>`	The type of the parameter, either *integer*, *real*, *ordinal* or *categorical*, given as a single letter: `i`, `r`, `o` or `c`.
`<range>`	The range or set of values of the parameter.
`<condition>`	An optional *condition* that determines whether the parameter is enabled or disabled. If the condition evaluates to false, then no value is assigned to this parameter, and the corresponding switch is not passed to `hookRun`. 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 contain any cycle. Otherwise, `irace` will detect the cycle and stop with an error.

Parameter types and range. 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 *significant digits* specified by option `signifDigits`. For example, given the default value number of significant digits of 4, the values 0.12345 and 0.12341 are both rounded to 0.1234. However, the values 0.00001 and 0.00005 remain the same.

- *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.

Conditional parameters. A conditional parameter is one that is enabled or disabled depending on the values of other parameters. A parameter is marked as conditional by adding the character `'|'` followed by an `R` expression involving the names of other parameters. This expression must return `TRUE` when the parameter should be enabled, and `FALSE` otherwise. Examples of operators that are valid in a condition are relational operators (`<, >, <=, >=, ==, !=`) and the logical operators: `!` (not), `&&` (logical-and), `||` (logical-or). Another useful operator is `%in%`, which returns `TRUE` if its left operand matches any element of its right operand. See the example below.

Example. We describe now in detail the example provided in `templates/parameters.tmpl`

```
param1     "--param1 "        i  (1, 10) | mode %in% c("x1", "x2")
param2     "--param2 "        c  (1, 10) | mode == "x1" && real < 3.5
mode       "--"               c  ("x1" ,"x2", "x3")
p.real     "--paramreal="     r  (1.5, 4.5)
p.ordinal  "--level="         o  ("low", "medium", "high", "very high")
#unused    "-u "              c  (1, 2, 10, 20)
```

In this example, `param1` is an integer parameter in the interval $[1, 10]$, and it is only enabled when parameter `mode` takes values `x1` or `x2`. Parameter `param2` is a categorical parameter with only two values, either `"1"` or `"10"`. It is only enabled when parameter `mode` takes the value `x1` and parameter `p.real` has a value lower than 3.5. Parameter `mode` is a categorical parameter with three possible values. Given its corresponding switch, these values are translated into `"--x1"`, `"--x2"` and `"--x3"` when invoking `hookRun`. Parameter `p.real` may take any real value within the interval $[1.5, 4.5]$ (subject to the number of significant digits specified by `signifDigits`). Parameter `p.ordinal` is an ordinal variable. Finally, parameter `unused` is commented out, and, hence, ignored.

It is possible to specify a categorical parameter with a single value. This parameter is added to the command line as usual when invoking `hookRun`, subject to any possible condition, but it is never sampled and it is not counted by N^{param}.

5.6 Hook files

The evaluation of the candidate configurations is done in two phases. In the first phase, the program specified by the option `hookRun` is invoked for each candidate configuration, passing as arguments: the instance name, a numeric identifier, and the command-line parameters of the candidate configuration. The numeric identifier uniquely identifies a configuration within a race (but not across the races in a single iterated race). The command line is constructed by appending to each parameter switch, *without separator*, the value of the parameter, following the order given in the parameter file. If the options `parallel`, `mpi` or `sgeCluster` are enabled, the calls to `hookRun` happen in parallel. The program `hookRun` should not have any output and it should exit with a value of zero. Once all calls to `hookRun` for the current instance have finished, the results are evaluated using the program specified by option `hookEvaluate`.

In this second phase, the program `hookEvaluate` is invoked for each candidate configuration in order of increasing candidate number. The parameters of `hookEvaluate` are the instance name, the numeric identifier and the total number of candidate configurations alive at this point of the race. The program `hookEvaluate` must print (only) a real number, which corresponds to the cost measure of the candidate configuration in the given instance.

The working directory of these programs is set to the execution directory specified by the option `execDir`. This allows the user to execute several runs of `irace` in parallel with the same configuration files without the runs interfering with each other.

5.7 Tuning Setup

The following steps detail how to setup a new tuning environment from scratch:

1. Create a new directory for storing the configuration of the tuning, e.g., `/path/to/tuning/`.

2. Copy the files in the `templates/` directory to your `/path/to/tuning/` directory. For each template hook (`*.tmpl`) in your `/path/to/tuning/` directory, remove the `'.tmpl'` suffix, and modify them following the instructions in each file. In particular, `tune-main.tmpl` should be adjusted depending on your usage (local, cluster, etc). The hooks should be executable. In `tune-conf.tmpl`, uncomment and assign only the parameters for which you need a value different than the default one. See the examples in `examples/`.

3. Put the instances in `/path/to/tuning/Instances/`. As described in Section 5.3, you can also create a file that specifies which instances from that directory should be run and which instance-specific parameters to use. See `tune-conf.tmpl` and `instances-list.tmpl` for examples.

4. Assuming that `irace` is installed in `/installdir/`, calling the command `/installdir/irace` from your tuning directory performs one run of `irace`. See Appendix A for additional configuration options. Command-line parameters override the configuration specified in the configuration file (`tune-conf`). The command `irace` will not attempt to create the execution directory (`execDir`), so it must exist before calling irace.

5. For executing several repetitions of `irace` in parallel, call the program `parallel-irace.sh` from your tuning directory with the number of repetitions. The execution directory of each run of irace will be set to "`/path/to/tuning/TUNE-dd`", where `dd` is a number padded with zeroes. Be careful, `parallel-irace.sh` will create these directories from scratch, deleting them first if they already exist. Check the help of `parallel-irace.sh` by running it without parameters.

There are three ways to execute the calls to `hookRun` in parallel:

1. `--parallel N` will use the `multicore` package [25] to launch locally up to `N` calls of `hookRun` in parallel.

2. `--mpi 1 --parallel N` will use the `Rmpi` package [26] to launch `N` slaves + 1 master, in order to execute `N` calls of `hookRun` in parallel. The user is responsible to set up MPI correctly. For an example of using MPI mode in an SGE cluster, see `examples/mpi/`.

3. `--sge-cluster 1` will launch as many calls of `hookRun` as possible and use '`qstat`' to wait for cluster jobs. The user **must** call '`qsub`' from `hookRun` with the appropriate configuration for their cluster, otherwise `hookRun` will not submit jobs to the cluster. In this mode, `irace` must run in the submission node, and hence, '`qsub`' should not be used to invoke `irace` (either directly or through `tune-main`). See the examples in `examples/cluster-mode/`.

6 Examples of Tuning Scenarios

6.1 ACOTSP: Tuning for solution quality

ACOTSP [24] is a software package that implements various ant optimization algorithms to tackle the symmetric traveling salesman problem.

We consider in this section the tuning of all its 11 parameters. In particular, we present the tuning of ACOTSP as a case study of using `irace` for tuning algorithms for optimization problems. We are interested in a parametrization of ACOTSP that finds the best possible solution quality for some TSP instances in a given time. Therefore, the computation effort that can be spent for the tuning is defined by the number of experiments `maxNbExperiments` (we set this option to 3000). The setup used for the tuning of ACOTSP is summarized in Table 2. We use a training set of 1000 instances and a testing set of 300 instances, all of them having 750 cities.

Table 2: Scenario setup for tuning ACOTSP

Max number of experiments (`maxNbExperiments`)	3000
Time per experiment	5 seconds

Note that the number of experiments is a parameter of `irace`, but the time used by the ACOTSP algorithm for each experiment is a fixed parameter, and thus we set it up in the hook file defined by `hookRun`. A second possibility would be to put this parameter in the parameter file with a single possible value, and this would not affect the tuning of the candidate parameters.

As examples of hooks and parameter files we provide the necessary files for tuning ACOTSP in the directory `examples/acotsp`.

6.2 SPEAR: Tuning for computation time

SPEAR [3] is a state-of-the-art theorem prover for solving industrial SAT instances. It is highly configurable, with a total of 26 parameters, including continuous, integer and categorical parameters. For this reason, Hutter et al. [13, 14, 15] have used it extensively as a benchmark for testing automatic tuning tools.

We consider in this section the tuning of SPEAR as a case study of using `irace` for tuning algorithms for decision problems. We are only interested in how much time SPEAR requires to decide whether a given SAT instance is satisfiable. SPEAR stops after a given cut-off time if it cannot determine an answer. Therefore, the computation effort of the tuning is given by the total computation time consumed by SPEAR, and, hence, we assign a total time budget (`irace` option `timeBudget`) of three days (259200 seconds). In order to calculate the number of experiments that may be performed in the remaining budget, we need an estimate of the time required by a single run of SPEAR. In the first iteration of `irace`, this estimate needs to be provided by the user (we set option `timeEstimate` to 30 seconds). In subsequent iterations, the estimate is adjusted by the program. During the tuning phase, we set the cut-off time of SPEAR to 30 seconds. After the tuning phase is over, the best configuration found by `irace` is evaluated by running it once on each test instance with a cut-off time of 1000 seconds. This setup is summarized in Table 3. In our experiments, we use a benchmark set of software verification instances [2], with 302 training instances and 302 test instances.

Table 3: Scenario setup for tuning SPEAR

Total time budget (`timedBudget`)	259200 seconds
Initial estimate of time per run (`timeEstimate`)	30 seconds
Tuning cut-off time	30 seconds
Testing cut-off time	1000 seconds

We first analyze the choice of the statistical test used within `irace` (option `testType`), either the Friedman-test or the t-test. In this first experiment, all runs of `irace` use a discretized parameter space, with all parameters specified as categorical, and the soft-restart feature is disabled. We execute five repetitions of `irace` with different random seeds for each type of statistical test. We use the same five random seeds for all experiments in order to reduce variance. Each repetition of `irace` returns one configuration of SPEAR, and these configurations are evaluated on the test instances, obtaining a computation time for each configuration on each test instance. We use two criteria to compare different tuning approaches:

- The mean computation time required by the configurations obtained when using each tuning approach. Moreover, we assess statistical significance using a paired-samples t-test. Results are paired not only with respect to the instance, but also with respect to the random seed used by the run of `irace` that generated each configuration.
- The percentage of success (%succ), that is, the percentage of times that the computation time of a configuration produced by one tuning approach is lower than the corresponding run of the configuration obtained by the other tuning approach. Results are also paired as before by instance and by the random seed of `irace`. In this case, we use the two-tailed sign-test to assess significance.

Table 4 compares the use of F-test and t-test in `irace` when configuring SPEAR. The table shows that the percentage of successes, that is, the number of instances that are solved faster, are maximized when using the F-test. On the other hand, using the t-test minimizes the mean computation time. Therefore, the choice of test clearly depends on the evaluation criterion.

In the second experiment, we compare the use of discretized, categorical parameters versus using the appropriate type (integer, real or ordered) for some parameters. Table 5 reports the results of this experiment. When using the F-test, there is no significant difference between using either only categorical parameters or not. However, when using the t-test, there is a slight advantage in using only categorical parameters if our goal is to optimize %succ, and an important difference in favor of using different types of parameters if our goal is to optimize the mean time.

In a third experiment, we evaluate the effect of the new soft-restart strategy. The results are summarized in Table 6. The soft-restart strategy always helps to improve %succ, and when `irace` uses the t-test and only categorical parameters, it significantly helps to reduce the mean time. Moreover, it never results in worse results.

Table 4: Comparison of F-test and t-test (only categorical parameters, no soft-restart).

	F-test	t-test	99% CI
%succ	59.54	17.95	$\Pr(a < b) \in [0.74, 0.8]$
mean	61.31	36.72	$\text{mean}(a - b) \in [8.15, 41.03]$

Table 5: Comparison of categorical versus mixed parameters.

F-test			
	Cat	Mixed	99% CI
%succ	36.69	31.26	$\Pr(a < b) \in [0.50, 0.58]$
mean	61.31	58.53	$\text{mean}(a - b) \in [-3.98, 9.54]$
t-test			
	Cat	Mixed	95% CI
%succ	47.28	32.65	$\Pr(a < b) \in [0.55, 0.63]$
mean	36.72	8.10	$\text{mean}(a - b) \in [15.85, 41.40]$

Table 6: Results when using the soft-restart option.

F-test (Cat)			
	Soft-restart	No	99% CI
%succ	43.11	24.17	$\Pr(a < b) \in [0.60, 0.68]$
mean	59.05	61.31	$\text{mean}(a - b) \in [-7.89, 3.37]$
t-test (cat)			
	Soft-restart	No	95% CI
%succ	57.62	22.32	$\Pr(a < b) \in [0.69, 0.75]$
mean	17.89	36.72	$\text{mean}(a - b) \in [-28.76, -8.89]$
F-test (mixed)			
	Soft-restart	No	99% CI
%succ	50.00	17.88	$\Pr(a < b) \in [0.70, 0.77]$
mean	54.59	58.53	$\text{mean}(a - b) \in [-11.81, 3.94]$
t-test (mixed)			
	Soft-restart	No	95% CI
%succ	52.72	26.95	$\Pr(a < b) \in [0.63, 0.70]$
mean	7.52	8.10	$\text{mean}(a - b) \in [-9.08, 1.16]$

The main conclusions of these experiments is that the best approach depends on how the results are evaluated. If the goal is to maximise %succ, the use of F-test is recommended, whereas if the goal is to minimise the mean time, then the t-test leads to better configurations. The soft-restart strategy sometimes leads to further improvements and, hence, we recommend its use by default.

7 Summary and Conclusions

We have presented `irace`, a software package that implements the iterated race procedure for automatic algorithm configuration. The proposed software implements the Iterated F-Race procedure and adds, as well, several improvements to it. The most notable improvements are the ability to automatically configure algorithms when the goal is to minimise computation time, and a soft-restart mechanism that prevents premature stagnation. We make publicly available the `irace` software package in the hope that it will be useful to the research community.

Acknowledgments. This work was supported by the META-X project, an *Action de Recherche Concertée* funded by the Scientific Research Directorate of the French Community of Belgium, and by the MIBISOC network, an Initial Training Network funded by the European Commission, grant PITN–GA–2009–238819. Manuel López-Ibáñez acknowledges support from the FRFC project "*Méthodes de recherche hybrides pour la résolution de problèmes complexes*". Mauro Birattari and Thomas Stützle acknowledge support from the Belgian F.R.S.-FNRS, of which they are Research Associates.

References

[1] B. Adenso-Díaz and M. Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99–114, 2006.

[2] D. Babić and A. J. Hu. Structural abstraction of software verification conditions. In *Computer Aided Verification: 19th International Conference, CAV 2007*, pages 366–378, 2007.

[3] D. Babić and F. Hutter. Spear theorem prover. In *SAT'08: Proceedings of the SAT 2008 Race*, 2008.

[4] P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In T. Bartz-Beielstein, M. J. Blesa, C. Blum, B. Naujoks, A. Roli, G. Rudolph, and M. Sampels, editors, *Hybrid Metaheuristics*, volume 4771 of *Lecture Notes in Computer Science*, pages 108–122. Springer, Heidelberg, Germany, 2007.

[5] M. Birattari. 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.

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

[7] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[8] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-race and iterated F-race: An overview. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer, Berlin, Germany, 2010.

[9] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, New York, NY, third edition, 1999.

[10] J. Dubois-Lacoste, M. López-Ibáñez, and T. Stützle. Improving the anytime behavior of two-phase local search. Technical Report TR/IRIDIA/2010-022, IRIDIA, Université Libre de Bruxelles, Belgium, Brussels, Belgium, 2010. URL http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2010-022r001.pdf. Submitted to Annals of Mathematics and Artificial Intelligence.

[11] J. Dubois-Lacoste, M. López-Ibáñez, and T. Stützle. A hybrid TP+PLS algorithm for bi-objective flow-shop scheduling problems. *Computers & Operations Research*, 38(8):1219–1236, 2011. doi: 10.1016/j.cor.2010.10.008.

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

[13] F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *FMCAD'07: Proceedings of the 7th International Conference Formal Methods in Computer Aided Design*, pages 27–34, Austin, Texas, USA, 2007. IEEE Computer Society, Washington, DC, USA.

[14] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proc. of the Twenty-Second Conference on Artifical Intelligence (AAAI '07)*, pages 1152–1157, 2007.

[15] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, Oct. 2009.

[16] C. H. Jackson. Multi-state models for panel data: The msm package for R. *Journal of Statistical Software*, 38(8):1–29, 2011. URL http://www.jstatsoft.org/v38/i08/.

[17] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 517–524, 2009.

[18] M. López-Ibáñez and T. Stützle. Automatic configuration of multi-objective ACO algorithms. In M. Dorigo et al., editors, *Ant Colony Optimization and Swarm Intelligence, 7th International Conference, ANTS 2010*, volume 6234 of *Lecture Notes in Computer Science*, pages 95–106. Springer, Heidelberg, Germany, 2010. doi: 10.1007/978-3-642-15461-4_9.

[19] M. López-Ibáñez and T. Stützle. The automatic design of multi-objective ant colony optimization algorithms. Technical Report TR/IRIDIA/2011-003, IRIDIA, Université Libre de Bruxelles, Belgium, Brussels, Belgium, 2011. URL http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2011-003r001.pdf.

[20] O. Maron and A. W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Research*, 11(1–5):193–225, 1997.

[21] M. A. Montes de Oca, D. Aydin, and T. Stützle. An incremental particle swarm for large-scale continuous optimization problems: An example of tuning-in-the-loop (re)design of optimization algorithms. *Soft Computing*, 2011. doi: 10.1007/s00500-010-0649-0.

[22] V. Nannen and A. E. Eiben. A method for parameter calibration and relevance estimation in evolutionary algorithms. In M. Cattolico et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2006*, pages 183–190. ACM press, New York, NY, 2006. doi: 10.1145/1143997.1144029.

[23] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

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

[25] S. Urbanek. *multicore: Parallel processing of R code on machines with multiple cores or CPUs*, 2010. URL `http://www.rforge.net/multicore/`. R package version 0.1-3.

[26] H. Yu. *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface)*, 2010. URL `http://cran.r-project.org/package=Rmpi`. R package version 0.5-8.

Appendix A: `irace` options

The following is a list of configurable parameters in `irace`. At each entry, we list first the name of the parameter in the configuration file, next the corresponding command-line switch (both long and short forms if available) and, finally, the default value. The value of a parameter may be given in the configuration file, in the command-line or in both. In the latter case, the command-line setting is given preference.

`-c --config-file ./tune-conf`
This specifies the file that contains the configuration for `irace`; see Section 5.4 for more information on the configuration file. The path can be given either as an absolute one or a relative one. In the latter case, the path is relative to the working directory.

parameterFile `-p --param-file ./parameters.txt`
This specifies the file that contains a description of the parameters for the algorithm to be tuned. See Section 5.5 for more information on the parameter file. The path can be given either as an absolute one or a relative one. In the latter case, the path is relative to the working directory.

execDir `--exec-dir ./TUNE`
This specifies the directory where the program to be tuned is run.

instanceDir `--instance-dir ./Instances`
This specifies the directory where tuning instances are located, either absolute or relative to `execDir`.

instanceFile `--instance-file ""`
A file containing a list of instances and (optionally) additional parameters for them. If this argument is empty, then the instances are simply taken from the directory specified by `instanceDir`.

candidatesFile `--candidates-file ""`
A file containing a list of initial candidates. If empty or `NULL`, do not use a file.

hookRun `--hook-run "./hook-run"`
The script called by `irace` for each candidate that launches the program to be tuned. See Section 5.6 for more information.

hookEvaluate `--hook-evaluate "./hook-evaluate"`
The scrip that provides a numeric value for each candidate. See Section 5.6 for more information.

expName `--exp-name "Experiment Name"`
Experiment name for output report.

expDescription `--exp-description "Experiment Description"`
Longer experiment description for output report.

maxNbExperiments `--max-experiments 1000`
The maximum number of runs (invocations of `hookRun`) that will performed. It determines the (maximum) budget of experiments for the tuning, unless `timeBudget` is positive.

timeBudget `--timeBudget 0`
If the value is greater than 0, it means tuning for computation time, and sets the maximum computation time that should be used for the whole tuning, in seconds. The default value (`0`), means tuning for solution quality (rather than for computation time) the budget in this case being defined by `maxNbExperiments`.

timeEstimate `--time-estimate 1`
An estimation of the average time in seconds required for one experiment. Only required if `timeBudget` is positive.

signifDigits `--signif-digits 4`
Indicates the significant digits to be considered for the real parameters.

debugLevel `--debug-level 0`
A value of 0 silences all debug messages. Higher values provide more verbose debug messages.

nbIterations `--iterations 0`
Number of iterations of `irace`. Do not use something else than the default (that is, the dynamic value) unless you know exactly what you are doing. The default value (`0`), means that the appropriate setting is determined by `irace`.

nbExperimentsPerIteration `--experiments-per-iteration 0`
Number of experiments per iteration. Do not use something else than the default (that is, the dynamic value) unless you know exactly what you are doing. The default value (`0`), means that the appropriate setting is determined by `irace`.

sampleInstances `--sample-instances 1`
Sample the instances or take them always in the same order.

maxInstances `--max-instances 1000`
The maximum number of instances to be considered for each iteration.

testType `--test-type F-test`
Specifies the statistical test type: `F-test` (Friedman test) or `t-test`.

firstTest `--first-test 5`
Specifies after how many instances are seen before the first test is performed.

eachTest `--each-test 1`
Specifies how many instances are seen before each test after the first one is performed.

minNbSurvival `--min-survival 0`
The minimum number of candidates that should survive to continue one iteration. Do not use something else than the default (that is, the dynamic value) unless you know exactly what you are doing.

nbCandidates `--num-candidates 0`
The number of candidates that should be sampled and evaluated at each iteration. Do not use something else than the default (that is, the dynamic value) unless you know exactly what you

are doing.

mu `--mu 5`
This value is used to determine the number of candidates to be sampled and evaluated at each iteration. Do not use something else than the default unless you know exactly what you are doing.

seed `--seed NA`
Seed of the random number generator (must be a positive integer, `NA` means use a random seed).

softRestart `--soft-restart 1`
Enable/disable the soft restart strategy that avoids premature convergence of the probabilistic model.

parallel `--parallel 0`
Number of calls to `hookRun` to execute in parallel. Less than 2 means calls to `hookRun` are sequentially executed.

sgeCluster `--sge-cluster 0`
Enable/disable SGE cluster mode. Use `qstat` to wait for cluster jobs to finish (`hookRun` must invoke `qsub`).

mpi `--mpi 0`
Enable/disable MPI. Use MPI to execute `hookRun` in parallel (if MPI is enabled with `1`, the value of parameter `parallel` defines the number of slaves).