The irace Package: Iterated Racing 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

Last revision: February 2012

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

Revision history:

TR/IRIDIA/2011-004.001	February 2011
TR/IRIDIA/2011-004.002	February 2012

The irace Package:
Iterated Racing for
Automatic Algorithm Configuration

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle and Mauro Birattari
IRIDIA, CoDE, Université Libre de Bruxelles, Brussels, Belgium

Abstract

The **irace** package implements the *iterated racing* procedure, which is an extension of Iterated F-race (I/F-Race). The main use of **irace** is the automatic configuration of optimization algorithms, that is, finding the most appropriate settings of an optimization algorithm given a set of instances of an optimization problem. It builds upon the **race** package by Birattari and it is implemented in R. This paper describes **irace** version 1.0. The **irace** package is available from CRAN. More information about **irace** is available at `http://iridia.ulb.ac.be/irace`.

Keywords: automatic algorithm configuration, racing, parameter tuning, R.

1. Introduction

The **irace** package implements the *iterated racing* procedure, which is an extension of Iterated F-race (I/F-Race) proposed by Balaprakash, Birattari, and Stützle (2007) and further developed by Birattari, Yuan, Balaprakash, and Stützle (2010). **irace** is implemented as an R package (R Development Core Team 2008) and it builds upon the **race** package by Birattari (2003). The main purpose of **irace** is to automatically configure optimization algorithms by finding the most appropriate settings given a set of tuning instances of an optimization problem. For example, tuning a computer program for logistic routing of trucks such that cost is minimized. Nonetheless, automatic configuration methods are also applicable to any system that has a number of configurable parameters, and whose performance on a particular problem often depends on the particular settings of these parameters. An example could be the tuning of an industrial process.

We are particularly interested in the tuning of metaheuristics, that is, general-purpose optimization algorithms such as evolutionary algorithms (Goldberg 1989), ant colony optimization (Dorigo and Stützle 2004), and other stochastic local search methods (Hoos and Stützle 2005). Automatic configuration methods may also be used for designing complex algorithms, by finding a particular configuration of a framework composed by alternative design choices and complementary algorithmic components (KhudaBukhsh, Xu, Hoos, and Leyton-Brown 2009; López-Ibáñez and Stützle 2012).

The scenario that **irace** addresses is usually described as *offline configuration* (Birattari 2009), 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. In other words, the ultimate purpose is that the high-quality configuration of the algorithm found during the tuning phase generalizes to similar but unseen instances.

Offline configuration is frequently performed ad-hoc by algorithm designers as a result of trial-and-error runs of candidate algorithms they consider. Computational methods for offline configuration involve the use of experimental design techniques (Coy, Golden, Runger, and Wasil 2001; Adenso-Díaz and Laguna 2006), evolutionary algorithms (Nannen and Eiben 2006; Ansótegui, Sellmann, and Tierney 2009), local search (Hutter, Hoos, Leyton-Brown, and Stützle 2009) or regression models (Hutter, Hoos, and Leyton-Brown 2011). Another notable example is sequential parameter optimization (SPO) (Bartz-Beielstein 2006), and the associated **SPOT** package (Bartz-Beielstein, Lasarczyk, and Preuss 2010; Bartz-Beielstein, Ziegenhirt, Konen, Flasch, Koch, and Zaefferer 2011). **SPOT** uses statistical models for finding optimal parameters of optimization algorithms. The main differences between **SPOT** and **irace** are that the former is more oriented towards analyzing the impact and interactions of parameters when applying an algorithm to a single instance or function, whereas the goal of **irace** is to find parameter configurations that perform well over a large set of instances.

Birattari, Stützle, Paquete, and Varrentrapp (2002); Birattari (2004, 2009) proposed an automatic configuration approach, F-Race, based on *racing* (Maron and Moore 1997) 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) (Balaprakash *et al.* 2007; Birattari *et al.* 2010). Although a formal description of the I/F-Race procedure is given in the original publications, no implementation of it has been 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 (2004, 2009), such as the use of the paired t test instead of Friedman's test. We have also added several original contributions, such as a new soft-restart mechanism that prevents premature convergence.

The **irace** package presented here has already been extensively tested in several research projects. Dubois-Lacoste, López-Ibáñez, and Stützle (2011b,a) used **irace** for tuning the parameters of several iterated greedy (IG) variants for various objectives in the permutation flow-shop problem (PFSP), outperforming the state-of-the-art. López-Ibáñez and Stützle (2010) automatically configured a flexible ant colony optimization framework for the bi-objective travelling salesman problem (bTSP). Montes de Oca, Aydin, and Stützle (2011) designed an incremental particle swarm optimization algorithm for large-scale continuous optimization problems by means of automatic configuration. F-Race, the precursor of I/F-Race, was used to configure an algorithm for the timetabling competition (Chiarandini, Birattari, Socha, and Rossi-Doria 2006). Other automatic configuration methods have been used in the literature to configure new state-of-the-art algorithms. For example, FocusedILS (Hutter *et al.* 2009) was used to automatically configure a highly parameterized tree search (Hutter, Babić, Hoos, and Hu 2007), and a framework of SAT solvers (KhudaBukhsh *et al.* 2009) that won several prizes in the International SAT competition; as well as tuning the commercial mixed-integer programming solver CPLEX, and obtaining a significant speedup over the default settings (Hutter, Hoos, and Leyton-Brown 2010). These examples demonstrate the potential of automatic configuration of algorithms, and the profound impact it will have on the way optimization algorithms are designed and evaluated in the future.

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. As an example, evolutionary algorithms (EAs) (Goldberg 1989) often require the user to specify settings like the mutation rate, the recombination operator and the population size. Another example is CPLEX, a mixed-integer programming solver, that has more than 76 configurable parameters affecting the main algorithm used internally by CPLEX, e.g., one can select among 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 (Adenso-Díaz and Laguna 2006; Birattari 2009).

There are two main classes of parameters: *categorical* and *numerical* parameters. Categorical parameters represent discrete values without any implicit order or sensible distance measure. An example is the different recombination operators in EAs. Numerical parameters have an implicit order of their values. Examples are the population size and the mutation rate in EAs. There are also seemingly categorical parameters but with an implicit order of their values. An example would be a parameter with three values {`low`, `medium`, `high`}. Such parameters are called *ordinal*, and we handle them as numerical parameters. Finally, parameters may be *subordinate* to other parameters, that is, they are only relevant for particular values of other parameters. For example, in a genetic algorithm there may be a parameter that defines the selection operator. More concretely, the selection operator could take the values `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 subordinate to the fact that the selection operator takes the value `tournament`. Subordinate 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 subordinate 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 $a = a' \cdot b$.

2.2. The algorithm configuration problem

In the following, we briefly introduce the algorithm configuration problem, a formal definition is given by Birattari (2009). Let us assume that we have a parametrized algorithm with N^{param} parameters, X_d, $d = 1, \dots, N^{\text{param}}$, 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 parametrized algorithm, the set of possible instances of the problem may be seen as a random variable $\mathcal{I}$ from which instances to be solved are sampled. We are also given a cost measure $\mathcal{C}(\theta, i)$ that assigns a value to each configuration when applied to a single problem instance i, which is a realization of $\mathcal{I}$. 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 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. The criterion that we want to optimize when configuring an algorithm for a problem is a function c_θ 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 c_θ.

A usual definition of c_θ is the expected cost of θ. The definition of c_θ 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 c_θ 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 $\mathcal{C}(\theta, i)$. In other words, by evaluating an algorithm configuration on instances sampled from $\mathcal{I}$. Since most algorithms of practical interest are sufficiently complex to preclude an analytical approach, 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 (Bartz-Beielstein 2006).

3. Iterated racing

3.1. An overview of iterated racing

The **irace** package that we propose in this paper is an implementation of iterated racing, of which I/F-Race (Balaprakash *et al.* 2007; Birattari *et al.* 2010) is a special case that uses the Friedman's non-parametric two-way analysis of variance by ranks.

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, each configurable parameter has an independent sampling distribution, which is either a normal distribution for numerical parameters, or a discrete distribution for categorical parameters. The update of the distributions consists of modifying the sampling distributions, the mean and 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

Algorithm 1 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: $\Theta_1 \sim \mathsf{SampleUniform}(X)$
 2: $\Theta^{\text{elite}} := \mathsf{Race}(\Theta_1,\ B_1)$
 3: $j := 2$
 4: while $B_{\text{used}} \leq B$ do
 5:     $\Theta^{\text{new}} \sim \mathsf{Sample}(X, \Theta^{\text{elite}})$
 6:     $\Theta_j := \Theta^{\text{new}} \cup \Theta^{\text{elite}}$
 7:     $\Theta^{\text{elite}} := \mathsf{Race}(\Theta_j,\ B_j)$
 8:     $j := j + 1$
 9: end while
10: Output: $\Theta^{\text{elite}}$
```

configurations found.

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 (Maron and Moore 1997). Birattari *et al.* (2002) 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 at least another one 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 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.

The next subsection gives a complete description of the iterated racing algorithm as implemented in the **irace** package. We mostly follow the description of the original papers (Balaprakash *et al.* 2007; Birattari *et al.* 2010), adding some details that were not explicitly given there. Later in Section 4, we mention several extensions that were not proposed in the original 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 of the **irace** package itself is given in Section 5.

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).

Iterated racing 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 $B_j = (B - B_{\text{used}})/(N^{\text{iter}} - j + 1)$, 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 $|\Theta_j| = N_j = \lfloor B_j/(\mu + \min(5, j)) \rfloor$. 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. The parameter μ allows the user to influence the ratio between budget and number of configurations, which also depends on the iteration number j. The idea behind this setting is that configurations generated in later iterations will be more similar, and, hence, more evaluations will be necessary to identify the best ones. On the other hand, we do not consider for computing this setting more than five iterations, in order to avoid having too few configurations in a single race.

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 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* candidates, are run on the next instance.

There are several alternatives for selecting which configurations should be discarded during the race. The F-Race algorithm (Birattari *et al.* 2002; Birattari 2009) 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 (Conover 1999). Nonetheless, the **race** package (Birattari 2003) 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.

A new statistical test is performed every T^{each} instances. By default $T^{\text{each}} = 1$, but in some situations it may be helpful to only perform each test after the configurations have been evaluated on a number of instances. The race continues until the budget of the current iteration is not enough to test all remaining candidate configurations on a new instance ($B_j < N_j^{\text{surv}}$), or when at most $N^{\min}$ configurations remain, $N_j^{\text{surv}} \leq N^{\min}$.

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 $N_j^{\text{elite}} = \min(N_j^{\text{surv}}, N^{\min})$ 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. 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, $d = 1, \ldots, N^{\text{param}}$, 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, non-subordinate parameters are sampled first, those parameters that are subordinate to them are sampled next if the condition is satisfied, and so on. Moreover, if a subordinate 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 $\mathcal{N}(x^z_d, \sigma^j_d)$, such that the new value is within the given range.[1] The mean of the distribution x^z_d is the value of parameter d in elite configuration θ_z. The parameter σ^j_d is initially set to $(\overline{x}_d - \underline{x}_d)/2$, and it is decreased at each iteration before sampling:

$$\sigma^j_d := \sigma^{j-1}_d \cdot \left(\frac{1}{N^{\text{new}}_j}\right)^{1/N^{\text{param}}} \tag{2}$$

By reducing σ^j_d 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 multi-dimensional volume of the sampling region is reduced by a constant factor at each iteration, but the reduction factor is higher when sampling a larger number of new candidate configurations (N^{new}_j).

If the numerical parameter is of integer type, we round the sampled value to the nearest integer. The sampling is adjusted to avoid the bias against the extremes introduced by rounding after sampling from a truncated distribution.

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 $\mathcal{P}^{j,z}(X_d)$. 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, 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 evaluated at the start of an iteration is not greater than the number of elites ($N_j \leq N^{\text{elite}}_{j-1}$), 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, we simply increase N^{iter} and continue the algorithm.

4. Extensions

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

[1]For sampling from a truncated normal distribution, we use the **msm** package (Jackson 2011).

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.

4.2. Soft-restart

Our implementation incorporates a "soft-restart" mechanism to avoid premature convergence. In the original I/F-Race proposal (Balaprakash *et al.* 2007), 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 tuning converges to a few, very similar elite configurations in few iterations, the diversity is lost and newly generated candidate configurations will not be very different from 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.

We implemented a "soft-restart" mechanism that 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 defined as the maximum distance between their parameter settings, which is defined as follows:

- If the parameter is subordinate 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 subordinate), then:
  - in the case of numerical parameters (integral or real), the distance is the absolute normalized difference between their values;
  - 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, whereas reinitializing their sampling distribution would mean to lose all the knowledge accumulated on them.

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 $p \in \mathcal{P}^{j,z}(X_d)$ with respect to the maximum value $p_{\max} = \max\{\mathcal{P}^{j,z}(X_d)\}$ as follows:

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

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

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

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.

5. The irace package

We provide here a brief summary of the **irace** package. The full documentation is available together with the 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, that is, the parameters to configure, their types, ranges and constraints. Section 5.2 summarizes how to define a parameter space in **irace**.
2. The set of tuning instances $\{I_1, I_2, \dots\}$, which in practice is a finite, representative sample of $\mathcal{I}$. The particular options for specifying the set of tuning instances are given in Section 5.1.
3. The configuration of **irace** itself, which is defined by a number of options. Table 1 maps the description of iterated racing in the previous section to the configuration options in **irace**. The complete list of options is available in the software documentation.

In addition, **irace** requires a function (or an auxiliary program) called `hookRun`, which is responsible of applying a particular configuration to an instance and returning the corresponding cost value.

The **irace** package is designed to be used either from within R, or from the command-line by means of a wrapper. We illustrate these two usage modes by means of simple examples in Section 6.

5.1. Tuning instances

The set of tuning instances $\{I_1, I_2, \dots\}$ may be given explicitly as a configuration option of **irace**. Alternatively, the instances may be read from an instance file (`instanceFile`). The string given by option `instanceDir` will be prefixed to them. 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 as given in `instanceFile` if this option is set or in alphabetical order if there is no `instanceFile`.

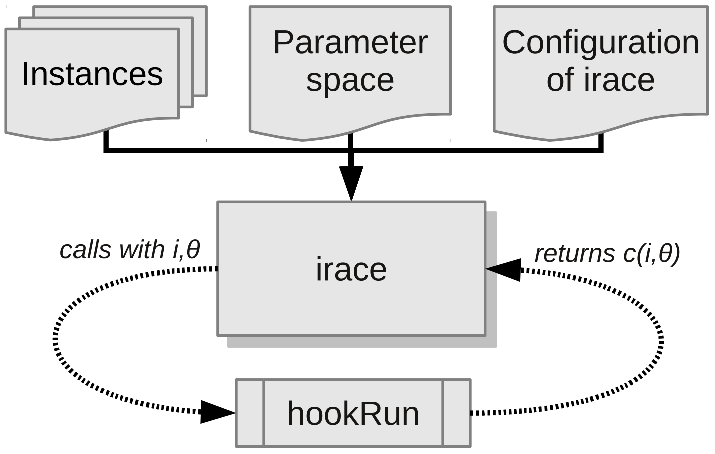

Figure 1: Scheme of **irace** flow of information.

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

Table 1: Configuration options of **irace** corresponding to the description of iterated racing given in Section 3.2. The full list of options is available in the complete documentation.

5.2. Parameter space

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

```
<name> <label> <type> <range> [ | <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 `hookRun`. In the default `hookRun` provided with the package (Section 5.3), 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`’.
`<range>`	The range or set of values of the parameter.
`<condition>`	An optional *condition* that determines whether the parameter is enabled or disabled, thus making the parameter subordinate. 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 `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 *decimal places* specified by 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 indexes of the values.

Section 6 shows examples on how to read the parameters table either from a string literal or from a file.

5.3. hookRun

The evaluation of a candidate configuration is done by means of a user-given function or, alternatively, a user-given auxiliary program. The function (or program name) is specified by the option `hookRun`.

When `hookRun` is an R function, then it is invoked for each candidate configuration as:

```
hookRun(instance, candidate, extra.params, config)
```

where `instance` is a single instance, `extra.params` is a user-defined value associated to this instance, `config` is the configuration of **irace**, and `candidate` is a list with three components: *(1)* `$index`, which is a numeric value identifying this candidate; *(2)* `$values`, which is a one-row data frame with one column per parameter value; and *(3)* `$labels`, which is a list of the labels of each parameter. The function `hookRun` must return a numerical value corresponding to the cost measure of the candidate configuration on the given instance.

When `hookRun` is an auxiliary executable program, then it is invoked for each candidate configuration, passing as arguments: the instance, 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 label (switch), *without separator*, the value of the parameter, following the order given in the parameter table. The program `hookRun` must print (only) a real number, which corresponds to the cost measure of the candidate configuration for the given instance. The working directory of `hookRun` is set to the execution directory specified by the option `execDir`. This allows the user to execute several runs of **irace** in parallel without the runs interfering with each other.

6. Examples of tuning scenarios

The next two sections illustrate two different ways of using **irace**. The first example shows how to set up and use the **irace** package by means of R programming. The second example shows how to tune an external program via the command-line options of the **irace** stand-alone program provided with the package.

6.1. Tuning `optim()` from R

In this illustrative example, our goal is to tune the parameters of the simulated annealing algorithm (SANN) provided by the `optim()` function in the **Rbase** package. In particular, let's say we are interested in optimizing instances of the following family of functions:

$$f(x) = \lambda \cdot f_{\text{Rastrigin}}(x) + (1 - \lambda) \cdot f_{\text{Rosenbrock}}(x) \tag{5}$$

where λ follows a normal distribution $\mathcal{N}(0.9, 0.02)$, and $f_{\text{Rastrigin}}$ and $f_{\text{Rosenbrock}}$ are the well-known Rastrigin and Rosenbrock benchmark functions. We use the implementation of these functions provided by the package **cmaes** (Trautmann, Mersmann, and Arnu 2011).

```
f_rosenbrock <- function (x) {
  d <- length(x)
  z <- x + 1
  hz <- z[1:(d - 1)]
  tz <- z[2:d]
  s <- sum(100 * (hz^2 - tz)^2 + (hz - 1)^2)
  return(s)
}
f_rastrigin <- function (x) {
  sum(x * x - 10 * cos(2 * pi * x) + 10)
}
```

In this scenario, different instances are given by different values of λ. Hence, we first generate 200 instances as follows:

```
weights <- rnorm(200, mean = 0.9, sd = 0.02)
```

We are interested in optimizing two parameters of the SANN algorithm: `tmax` and `temp`. We setup the parameter space as follows:

```
parameters.table <- '
tmax "" i (1, 5000)
temp "" r (0, 100)
'
```

and we use the **irace** function `readParameters` to read this table:

```
R> library("irace")
R> parameters <- readParameters(text=parameters.table)
```

Next, we define the function that will evaluate each candidate configuration on a single instance. For simplicity, we restrict to three-dimensional functions and we set the maximum number of iterations of SANN to 5000.

```
hook.run <- function(instance, candidate, extra.params = NULL, config = list())
{
  D <- 3
  par <- runif(D, min=-1, max=1)
  fn <- function(x) {
    weight <- instance
    return(weight * f_rastrigin(x) + (1 - weight) * f_rosenbrock(x))
  }
  res <- optim(par,fn, method="SANN",
               control=list(maxit=5000
                 , tmax = as.numeric(candidate$values[["tmax"]])
                 , temp = as.numeric(candidate$values[["temp"]])
                 ))
  return(res$value)
}
```

We are now ready to launch **irace**. We do it by means of the `irace` function by setting `hookRun` to the function define above, `instances` to the first 100 random weights, and a maximum budget of 1000 calls to `hookRun`.

```
R> result <- irace(tunerConfig = list(
                    hookRun = hook.run,
                    instances = weights[1:100],
                    maxExperiments = 1000),
                  parameters = parameters)
```

The function `irace` will print information about its progress. We can print the best configurations found as follows:

```
R> candidates.print(result)
```

```
      tmax   temp
118 3501 0.8984
126 3487 1.1865
103 3441 0.3150
```

We could now evaluate the cost of the best configuration found by **irace** versus the default configuration of SANN on the other 100 instances previously generated and not used during training.

```
R> default <- sapply(weights[101:200], hook.run,
                  candidate=list(values=list(tmax=10,temp=10)))

R> result.list <- as.list(removeCandidatesMetaData(result[1,]))
R> tuned <- sapply(weights[101:200], hook.run, candidate=list(values=result.list))

R> boxplot(list(default=default, tuned=tuned))
```

The resulting plot is given in Fig. 2. The boxplot clearly shows that the tuned configuration is able to find better solutions than the default configuration of SANN. This small example is included in the documentation of the **irace** package and can be run with the command:

```
R> example(irace)
```

6.2. Tuning ACOTSP

ACOTSP (Stützle 2002) is a software package that implements various ant optimization algorithms to tackle the symmetric traveling salesman problem (TSP). The example proposed 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. The setup of the tuning procedure is defined through various files.

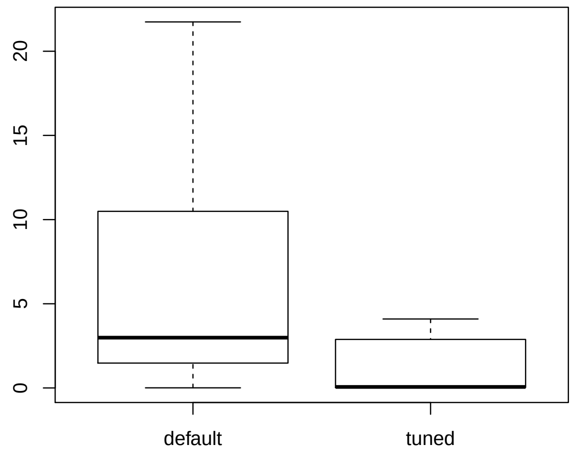

Figure 2: Boxplot of default vs. tuned configuration of SANN

```
# name      switch           type values    [| conditions (using R syntax)]
algorithm   "--"             c    (as,mmas,eas,ras,acs)
localsearch "--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)     | localsearch %in% c(1, 2, 3)
q0          "--q0 "          r    (0.0, 1.0) | algorithm %in% c("acs")
dlb         "--dlb "         c    (0, 1)      | localsearch %in% c(1,2,3)
rasrank     "--rasranks "    i    (1, 100)    | algorithm %in% c("ras")
elitistants "--elitistants " i    (1, 750)    | algorithm %in% c("eas")
```

Figure 3: Parameter file (`parameters.txt`) for tuning **ACOTSP**.

```
execDir <- "./tuning/"
maxExperiments <- 1000
```

Figure 4: Configuration file (`tune-conf`) for tuning **ACOTSP**.

First, we define a parameter file (`parameters.txt`, Fig. 3) that describes the parameter space. We also create a configuration file (`tune-conf`, Fig. 4) to overwrite some default options of **irace**. In particular, we set an execution directory (e.g., `./tuning/`) where temporary files are stored, and we set the tuning budget to 1 000 runs of **ACOTSP**. Next, we place the tuning instances in the subdirectory `./Instances/`, which is the default value of the option `instanceDir`. We create a basic `hook-run` script that simply runs the **ACOTSP** software for 20 seconds and prints the objective value of the best solution found.[2] We can now launch the tuning procedure as follows:

```
R> library("irace")
R> irace.cmdline()
```

The package provides a convenient command-line wrapper for Unix environments, called `irace`, located in `file.path(system.file(package="irace"), "bin")`, that basically invokes R and executes the commands above. The command-line wrapper also allows the user to specify many options directly from the command-line.

Most of the output is generated by the underlying **race** package, which prints a detailed progress of each race. After each race finishes, the set of elite configurations are printed. At the end, the best configurations found are printed as a table and as command-line parameters:

```
# Best candidates
    algorithm localsearch alpha   beta    rho ants nnls     q0 dlb rasrank elitistants
189       acs           3 1.268 6.0930 0.6846   20   17 0.2812   1      NA          NA
332       acs           3 3.100 0.7011 0.4789   55   11 0.2630   1      NA          NA
320      mmas           3 1.361 9.0390 0.6852   64   25     NA   1      NA          NA
# Best candidates (as commandlines)
                                                                                 command
189  --acs --localsearch 3 --alpha 1.268 --beta 6.0930 --rho  0.6846 --ants 20 --nnls 17  \
    --q0 0.2812 --dlb 1
332  --acs --localsearch 3 --alpha 3.100 --beta 0.7011 --rho  0.4789 --ants 55 --nnls 11  \
    --q0 0.2630 --dlb 1
320  --mmas --localsearch 3 --alpha 1.361 --beta 9.0390 --rho  0.6852 --ants 64 --nnls 25 \
    --dlb 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 `tunerResults`, whose elements are:

- `tunerConfig`: the configuration of **irace**.
- `parameters`: the parameter space.
- `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. The first column (‘`instance`’) indicates the instance tested in the experiments of the same row. The second column (‘`iteration`’) gives the iteration (race) number in which the experiments in the same row where performed. The remainder of the columns represent configurations, and their column names correspond to their

[2]The package includes an example of this `hook-run` script for Unix environments, which can be found at `file.path(system.file(package="irace"), "examples","acotsp")`.

IDs. Finally, '`NA`' represents that for some reason the candidate was not evaluated on a particular instance at that iteration, either because it did not exist yet or it was removed earlier.

- `allCandidates`: a data frame with all candidate configurations tested during the execution of **irace**.

7. Conclusion

This paper presents 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 algorithm. However, it may also be used for determining good settings in other computational systems. 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. In GNU/Linux and MacOS X, a command-line wrapper makes the use of R completely transparent to the user.

The **irace** package is available from CRAN. More information about **irace** is available at http://iridia.ulb.ac.be/irace.

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. Thomas Stützle, Mauro Birattari and Manuel López-Ibáñez acknowledge support from the Belgian F.R.S.-FNRS, of which they are Research Associates and Postdoctoral researcher, respectively. The authors also acknowledge support from the FRFC project "*Méthodes de recherche hybrides pour la résolution de problèmes complexes*".

References

Adenso-Díaz B, Laguna M (2006). "Fine-Tuning of Algorithms Using Fractional Experimental Design and Local Search." *Operations Research*, **54**(1), 99–114.

Ansótegui C, Sellmann M, Tierney K (2009). "A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms." In IP Gent (ed.), *Principles and Practice of Constraint Programming, CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pp. 142–157. Springer-Verlag, Heidelberg, Germany.

Balaprakash P, Birattari M, Stützle T (2007). "Improvement Strategies for the F-Race Algorithm: Sampling Design and Iterative Refinement." In T Bartz-Beielstein, MJ Blesa, C Blum, B Naujoks, A Roli, G Rudolph, M Sampels (eds.), *Hybrid Metaheuristics*, volume 4771 of *Lecture Notes in Computer Science*, pp. 108–122. Springer-Verlag, Heidelberg, Germany.

Bartz-Beielstein T (2006). *Experimental Research in Evolutionary Computation: The New Experimentalism.* Springer-Verlag, Berlin, Germany.

Bartz-Beielstein T, Lasarczyk C, Preuss M (2010). "The Sequential Parameter Optimization Toolbox." In T Bartz-Beielstein, M Chiarandini, L Paquete, M Preuss (eds.), *Experimental Methods for the Analysis of Optimization Algorithms*, pp. 337–360. Springer-Verlag, Berlin, Germany.

Bartz-Beielstein T, Ziegenhirt J, Konen W, Flasch O, Koch P, Zaefferer M (2011). ***SPOT**: Sequential Parameter Optimization.* R package, URL http://cran.r-project.org/package=SPOT.

Birattari M (2003). "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.

Birattari M (2004). *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective.* Ph.D. thesis, Université Libre de Bruxelles, Brussels, Belgium.

Birattari M (2009). *Tuning Metaheuristics: A Machine Learning Perspective*, volume 197 of *Studies in Computational Intelligence.* Springer-Verlag, Berlin / Heidelberg.

Birattari M, Stützle T, Paquete L, Varrentrapp K (2002). "A Racing Algorithm for Configuring Metaheuristics." In WB Langdon, *et al.* (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, pp. 11–18. Morgan Kaufmann Publishers, San Francisco, CA.

Birattari M, Yuan Z, Balaprakash P, Stützle T (2010). "F-Race and Iterated F-Race: An Overview." In T Bartz-Beielstein, M Chiarandini, L Paquete, M Preuss (eds.), *Experimental Methods for the Analysis of Optimization Algorithms*, pp. 311–336. Springer-Verlag, Berlin, Germany.

Chiarandini M, Birattari M, Socha K, Rossi-Doria O (2006). "An Effective Hybrid Algorithm for University Course Timetabling." *Journal of Scheduling*, **9**(5), 403–432.

Conover WJ (1999). *Practical Nonparametric Statistics.* Third edition. John Wiley & Sons, New York, NY.

Coy SP, Golden BL, Runger GC, Wasil EA (2001). "Using Experimental Design to Find Effective Parameter Settings for Heuristics." *Journal of Heuristics*, **7**(1), 77–97.

Dorigo M, Stützle T (2004). *Ant Colony Optimization.* MIT Press, Cambridge, MA.

Dubois-Lacoste J, López-Ibáñez M, Stützle T (2011a). "A Hybrid TP+PLS Algorithm for Bi-objective Flow-Shop Scheduling Problems." *Computers & Operations Research*, **38**(8), 1219–1236.

Dubois-Lacoste J, López-Ibáñez M, Stützle T (2011b). "Improving the Anytime Behavior of Two-Phase Local Search." *Annals of Mathematics and Artificial Intelligence*, **61**(2), 125–154.

Goldberg DE (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Boston, MA, USA.

Hoos HH, Stützle T (2005). *Stochastic Local Search—Foundations and Applications.* Morgan Kaufmann Publishers, San Francisco, CA.

Hutter F, Babić D, Hoos HH, Hu AJ (2007). “Boosting Verification by Automatic Tuning of Decision Procedures.” In *FMCAD'07: Proceedings of the 7th International Conference Formal Methods in Computer Aided Design*, pp. 27–34. IEEE Computer Society, Washington, DC, USA.

Hutter F, Hoos HH, Leyton-Brown K (2010). “Automated Configuration of Mixed Integer Programming Solvers.” In A Lodi, M Milano, P Toth (eds.), *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*, pp. 186–202. Springer-Verlag, Heidelberg, Germany.

Hutter F, Hoos HH, Leyton-Brown K (2011). “Sequential Model-Based Optimization for General Algorithm Configuration.” In *Learning and Intelligent Optimization, 5th International Conference, LION 5*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany.

Hutter F, Hoos HH, Leyton-Brown K, Stützle T (2009). “ParamILS: An Automatic Algorithm Configuration Framework.” *Journal of Artificial Intelligence Research*, **36**, 267–306.

Jackson CH (2011). “Multi-State Models for Panel Data: The **msm** Package for R.” *Journal of Statistical Software*, **38**(8), 1–29.

KhudaBukhsh AR, Xu L, Hoos HH, Leyton-Brown K (2009). “SATenstein: Automatically Building Local Search SAT Solvers from Components.” In C Boutilier (ed.), *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pp. 517–524.

López-Ibáñez M, Stützle T (2010). “Automatic Configuration of Multi-Objective ACO Algorithms.” In M Dorigo, *et al.* (eds.), *Swarm Intelligence, 7th International Conference, ANTS 2010*, volume 6234 of *Lecture Notes in Computer Science*, pp. 95–106. Springer-Verlag, Heidelberg, Germany.

López-Ibáñez M, Stützle T (2012). “The Automatic Design of Multi-Objective Ant Colony Optimization Algorithms.” *IEEE Transactions on Evolutionary Computation.* Accepted.

Maron O, Moore AW (1997). “The Racing Algorithm: Model Selection for Lazy Learners.” *Artificial Intelligence Research*, **11**(1–5), 193–225.

Montes de Oca MA, Aydin D, Stützle T (2011). “An Incremental Particle Swarm for Large-Scale Continuous Optimization Problems: An Example of Tuning-in-the-loop (Re)Design of Optimization Algorithms.” *Soft Computing*, **15**(11), 2233–2255.

Nannen V, Eiben ÁE (2006). “A Method for Parameter Calibration and Relevance Estimation in Evolutionary Algorithms.” In M Cattolico, *et al.* (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2006*, pp. 183–190. ACM press, New York, NY.

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

Stützle T (2002). "**ACOTSP**: A Software Package of Various Ant Colony Optimization Algorithms Applied to the Symmetric Traveling Salesman Problem." URL `http://www.aco-metaheuristic.org/aco-code/`.

Trautmann H, Mersmann O, Arnu D (2011). ***cmaes**: Covariance Matrix Adapting Evolutionary Strategy*. R package, URL `http://cran.r-project.org/package=cmaes`.

Affiliation:

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, Mauro Birattari
IRIDIA, CoDE,
Université Libre de Bruxelles (ULB)
1050 Brussels, Belgium

E-mail:

`manuel.lopez-ibanez@ulb.ac.be`
`jeremie.dubois-lacoste@ulb.ac.be`
`stuetzle@ulb.ac.be`
`mbiro@ulb.ac.be`