

# The **irace** Package: User Guide

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

August 11, 2016

## Contents

<b>1</b>	<b>General information</b>	<b>4</b>
1.1	Background . . . . .	4
1.2	Version . . . . .	4
1.3	License . . . . .	4
<b>2</b>	<b>Before starting</b>	<b>4</b>
<b>3</b>	<b>Installation</b>	<b>5</b>
3.1	System requirements . . . . .	5
3.2	<b>irace</b> installation . . . . .	5
3.2.1	Install automatically within R . . . . .	5
3.2.2	Manual download and installation . . . . .	5
3.2.3	Local installation . . . . .	6
3.2.4	Testing the installation and invoking irace . . . . .	7
<b>4</b>	<b>Running irace</b>	<b>7</b>
4.1	Step-by-step setup guide . . . . .	9
4.2	Set-up example for ACOTSP . . . . .	12
<b>5</b>	<b>irace scenario</b>	<b>13</b>
5.1	Target algorithm parameters . . . . .	13
5.1.1	Parameter types . . . . .	13
5.1.2	Parameter domains . . . . .	14
5.1.3	Conditional parameters . . . . .	14
5.1.4	Parameter file format . . . . .	14
5.1.5	Parameters R format . . . . .	15
5.2	Target algorithm runner . . . . .	17
5.2.1	Target runner executable program . . . . .	17
5.2.2	Target runner R function . . . . .	18
5.3	Target evaluator . . . . .	19
5.3.1	Target evaluator R function . . . . .	20
5.3.2	Target evaluator executable program . . . . .	20
5.4	Training instances . . . . .	20

5.5	Initial configurations . . . . .	22
5.6	Forbidden configurations . . . . .	22
<b>6</b>	<b>Parallelization</b>	<b>23</b>
<b>7</b>	<b>Testing of configurations</b>	<b>23</b>
<b>8</b>	<b>Recovering irace runs</b>	<b>24</b>
<b>9</b>	<b>Output and results</b>	<b>25</b>
9.1	Text output . . . . .	25
9.2	Data file output . . . . .	27
9.3	Analysis of results . . . . .	32
<b>10</b>	<b>Advanced topics</b>	<b>37</b>
10.1	Tuning budget . . . . .	37
10.2	Multi-Objective tuning . . . . .	38
10.3	Tuning computation time . . . . .	38
10.4	Heterogeneous scenarios . . . . .	39
10.5	Choosing the statistical test . . . . .	39
10.6	Complex parameters . . . . .	40
10.7	Unreliable target algorithms . . . . .	41
<b>11</b>	<b>irace options</b>	<b>41</b>
11.1	General options . . . . .	41
11.2	Elitist <b>irace</b> . . . . .	42
11.3	Internal <b>irace</b> options . . . . .	43
11.4	Target algorithm parameters . . . . .	44
11.5	Target algorithm execution . . . . .	44
11.6	Initial configurations . . . . .	45
11.7	Training instances . . . . .	45
11.8	Tuning budget . . . . .	45
11.9	Statistical test . . . . .	46
11.10	Recovery . . . . .	46
11.11	Testing . . . . .	46
<b>12</b>	<b>FAQ</b>	<b>47</b>
12.1	Is <b>irace</b> minimizing or maximizing the output of my algorithm? . . . . .	47
12.2	Is it possible to configure a MATLAB algorithm with <b>irace</b> ? . . . . .	47
12.3	My program works perfectly on its own, but not when running under <b>irace</b> . Is <b>irace</b> broken? . . . . .	47
12.4	My program may be buggy and run into an infinite loop. Is it possible to set a maximum timeout? . . . . .	47
12.5	When using the <code>mpi</code> option, <b>irace</b> is aborted with an error message indicating that a function is not defined. How to fix this? . . . . .	48
<b>13</b>	<b>Resources and contact information</b>	<b>48</b>
<b>14</b>	<b>Acknowledgements</b>	<b>49</b>
	<b>Appendix A R installation</b>	<b>50</b>

A.1	GNU/Linux . . . . .	50
A.2	OS X . . . . .	50
A.3	Windows . . . . .	50
<b>Appendix B TargetRunner script check list</b>		<b>50</b>
<b>Appendix C Glossary</b>		<b>53</b>

# 1 General information

## 1.1 Background

The **irace** package implements an *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 and decision algorithms, that is, finding the most appropriate settings of an algorithm given a set of instances of a problem. However, it may also be useful for configuring other types of algorithms when performance depends on the used parameter settings. It builds upon the **race** package by Birattari and it is implemented in R. The **irace** package is available from CRAN. More information about **irace** is available at <http://iridia.ulb.ac.be/irace>.

## 1.2 Version

The current version of the **irace** package is version 2.0. Previous versions of the package can be found in the **irace** package CRAN website.

<https://cran.r-project.org/web/packages/irace/>



Previous versions of **irace** might not be compatible with the file formats detailed in this document.

## 1.3 License

The **irace** package is Copyright © 2016 and distributed under the GNU General Public License version 3.0 (<http://www.gnu.org/licenses/gpl-3.0.en.html>). The **irace** package is free software (software libre): you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

The **irace** package is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

# 2 Before starting

The **irace** package provides an automatic configuration tool for tuning optimization algorithms. The **irace** tool automatically finds good configurations for the parameters values of a (target) algorithm saving the effort that normally requires manual tuning.

Figure 1 gives a general scheme of how **irace** works. **irace** receives as input a *parameter space definition* corresponding to the parameters of the target algorithm that will be tuned, a set of *instances* for which the parameters must be tuned for and a set of *options* for **irace**.

**irace** searches in the parameter search space for good performing algorithm configurations by executing the target algorithm on different instances and with different parameter configurations. To execute the target algorithm with a specific parameter configuration ( $\theta$ ) and instance ( $i$ ) a **targetRunner** must be provided. The **targetRunner** acts as an interface between the execution of the target algorithm and **irace**: It receives the instance and configuration as arguments and must return the evaluation of the execution of the target algorithm.

The following user guide contains guidelines to use **irace** and define the needed components to execute **irace** to automatically configure an optimization algorithm.

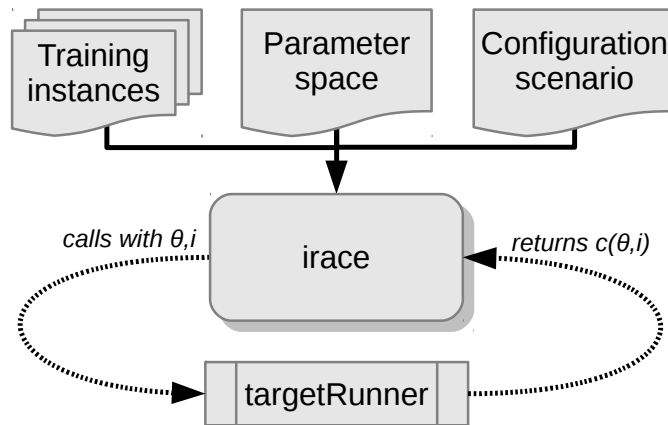


Figure 1: Scheme of **irace** flow of information.

## 3 Installation

### 3.1 System requirements

- R (version  $\geq 2.15$ ) is required for running **irace**, but you don't need to know the R language to use it. R is freely available and you can download it from the R project website (<http://www.r-project.org>). See [Appendix A](#) for a quick installation guide of R.
- For GNU/Linux and OS X, the command-line executables **irace** and **parallel-irace** require GNU Bash.

### 3.2 **irace** installation

The **irace** package can be installed automatically within R or by manual download and installation. We advise to use the automatic installation unless particular circumstances do not allow it. The instructions to install **irace** with the two mentioned methods are the following:

#### 3.2.1 Install automatically within R

In the R console execute the following line to install the package :

```
> install.packages("irace")
```

Select a mirror close to your location, and test the installation in the R console with:

```
> library("irace")
> CTRL+d
```

Alternatively, within the R graphical interface, you may use the **Packages** and **data ->Package installer** menu on OS X or the **Packages** menu on Windows.

#### 3.2.2 Manual download and installation

From the **irace** package CRAN website (<http://cran.r-project.org/package=irace>), download one of the three versions available depending on your operating system:

- `irace_2.0.tar.gz` (Unix/BSD/GNU/Linux)
- `irace_2.0.tgz` (OS X)
- `irace_2.0.zip` (Windows)

To install the package on GNU/Linux and OS X, you must execute the following command at the shell:

```
# Replace <package> with the path of the downloaded file.
R CMD INSTALL <package>
```

To install the package on Windows open R and execute the following line on the R console:

```
# Replace <package> with the path of the downloaded file.
install.packages("<package>", repos = NULL)
```

If the previous installation instructions fail because of insufficient permissions and you do not have sufficient admin rights to install **irace** system-wide, then you need to force a local installation.

### 3.2.3 Local installation

Let's assume you wish to install **irace** on a path denoted by `<R_LIBS_USER>`, which is a filesystem path for which you have sufficient rights. This directory **must** exist before attempting the installation. Moreover, you must provide to R the path to this library when loading the package. However, the latter can be avoided by adding the path to the system variable `R_LIBS` or to the R internal variable `.libPaths`, as we will see below.<sup>1</sup>

On GNU/Linux or OS X, execute the following commands to install the package on a local directory:

```
export R_LIBS_USER="<R_LIBS_USER>"
# Create R_LIBS_USER if it doesn't exist
mkdir $R_LIBS_USER
# Replace <package> with the path to the downloaded file.
R CMD INSTALL --library=$R_LIBS_USER <package>
# Tell R where to find R_LIBS_USER
export R_LIBS=${R_LIBS_USER}:${R_LIBS}
```

On Windows, you can install the package on a local directory by executing the following lines in the R console:

```
> # Replace <package> with the path to the downloaded file.
> # Replace <R_LIBS_USER> with the path used for installation.
> install.packages("<package>", repos = NULL, lib = "<R_LIBS_USER>")
> # Tell R where to find R_LIBS_USER.
> # This must be executed for every new session.
> .libPaths(c("<R_LIBS_USER>", .libPaths()))
```

<sup>1</sup>On Windows, see also [https://cran.r-project.org/bin/windows/base/rw-FAQ.html#I-don\\_0027t-have-permission-to-write-to-the-R\\_002d3\\_002e3\\_002e1\\_005clibrary-directory](https://cran.r-project.org/bin/windows/base/rw-FAQ.html#I-don_0027t-have-permission-to-write-to-the-R_002d3_002e3_002e1_005clibrary-directory).

### 3.2.4 Testing the installation and invoking irace

Once **irace** has been installed, load the package and test that the installation was successful by opening an R console and executing:

```
> # Load the package
> library("irace")
> # Obtain the installation path
> system.file(package = "irace")
```

The last command must print out the filesystem path where **irace** is installed. In the remainder of this guide, the variable `$IRACE_HOME` is used to denote this path. When executing any provided command that includes the `$IRACE_HOME` variable do not forget to replace this variable with the installation path of **irace**.

On GNU/Linux or OS X, you can let the operating system know where to find **irace** by defining the `$IRACE_HOME` variable and adding it to the system `PATH`. Append the following commands to `~/.bash_profile`, `~/.bashrc` or `~/.profile`:

```
# Replace <IRACE_HOME> with the irace installation path
export IRACE_HOME=<IRACE_HOME>
export PATH=${IRACE_HOME}/bin/:$PATH
# Tell R where to find R_LIBS_USER
# Use the following line only if local installation was forced
export R_LIBS=${R_LIBS_USER}:${R_LIBS}
```

Then, open a new terminal and launch **irace** as follows:

```
irace --help
```

On Windows, you need to add both R and the installation path of **irace** to the environment variable `PATH`. To edit the `PATH`, search for “Environment variables” in the control panel, edit `PATH` and add a string similar to `C:\R_PATH\bin;C:\IRACE_HOME\bin`, where `R_PATH` is the installation path of R and `IRACE_HOME` is the installation path of **irace**. If **irace** was installed locally, you also need to edit the environment variable `R_LIBS` to add `R_LIBS_USER`. Then, open a new terminal (run program `cmd.exe`) and launch **irace** as:

```
irace.bat --help
```

Alternatively, you may directly invoke **irace** from within the R console by executing:

```
> library("irace")
> irace.cmdline("--help")
```

## 4 Running irace

Before performing the tuning of your algorithm, it is necessary to define an **irace** tuning scenario that will give **irace** all the necessary information to optimize the parameters of the algorithm. The tuning scenario is composed of the following elements:

1. Target algorithm parameter description (see [Section 5.1](#)).
2. Target algorithm runner (see [Section 5.2](#)).
3. Training instances list (see [Section 5.4](#)).
4. Irace options (see [Section 11](#)).
5. *Optional*: Initial configurations (see [Section 5.5](#)).
6. *Optional*: Forbidden configurations (see [Section 5.6](#)).
7. *Optional*: Target algorithm evaluator (see [Section 5.3](#)).

These scenario elements can be provided as plain text files or as R objects. This user guide provides examples of both types, but we advise the use of plain text files, which we consider the simpler option.

For a step-by-step guide to create the scenario elements for your target algorithm continue to [Section 4.1](#). For an example execution of **irace** using the **ACOTSP** scenario go to [Section 4.2](#).

Once all the scenario elements are prepared you can execute **irace**. **irace** can be executed using the command-line wrappers provided by the package or directly from the R console:

1. Execute **irace** from the command-line as (on Windows, you should execute **irace.bat**):

```
# $IRACE_HOME is the installation directory of irace.
$IRACE_HOME/bin/irace --scenario scenario.txt
```

For this example we assume that the needed scenario files have been set properly in the **scenario.txt** file using the options described in [Section 11](#). Most **irace** options can be specified in the command line or directly in the **scenario.txt** file.

2. Execute **irace** from the R console as:

```
> library("irace")
> parameters <- readParameters("parameters.txt")
> scenario <- readScenario(filename = "scenario.txt",
+                          scenario = defaultScenario())
> irace(scenario = scenario, parameters = parameters)
```

**irace** provides an option (**--check**) to check that the scenario is correctly defined. We recommend to perform a check every time you create a new scenario. When performing the check, **irace** will verify that the scenario and parameter definitions are correct and will test the execution of the target algorithm. To check your scenario execute the following commands:

1. From the command-line (on Windows, execute **irace.bat**):

```
# $IRACE_HOME is the installation directory of irace.
$IRACE_HOME/bin/irace --scenario scenario.txt --check
```

2. From the R console:



```

> library("irace")
> parameters <- readParameters("parameters.txt")
> scenario <- readScenario(filename = "scenario.txt",
+                           scenario = defaultScenario())
> checkIraceScenario(scenario = scenario, parameters = parameters)

```

## 4.1 Step-by-step setup guide

This section provides a guide to setup a basic execution of **irace**. The template files provided in the package (`$IRACE_HOME/templates`) will be used as basis for creating your new scenario. Please follow carefully the indications provided in each step and in the template files used; if you have doubts check the the sections that describe each option in detail.

1. Create a directory (e.g., `~/tuning/`) for the scenario setup. This directory will contain all the files that describe the scenario. On GNU/Linux or OS X, you can do this as follows:

```

mkdir ~/tuning
cd ~/tuning

```

2. Copy all the template files from the `$IRACE_HOME/templates/` directory to the scenario directory.

```

# $IRACE_HOME is the installation directory of irace.
cp $IRACE_HOME/templates/*.tmpl ~/tuning/

```

Remember that `$IRACE_HOME` is the path to the installation directory of **irace**. It can be obtained in the R console with:

```

> library("irace")
> system.file(package = "irace")

```

3. For each template in your tuning directory, remove the `.tmpl` suffix, and modify them following the next steps.
4. Define the target algorithm parameters to be tuned, follow the instructions in `parameters.txt`. Available parameter types and other guidelines can be found in [Section 5.1](#).
5. *Optional*: Define the initial parameter configuration(s) of your algorithm, this option allows you to provide good starting configurations (if you know some) for the tuning. Follow the instructions in `configurations.txt`. More information in [Section 5.5](#). **Important**: If you do not need to define initial configurations remove this file from the directory.
6. *Optional*: Define forbidden parameter value combinations, that is, configurations that **irace** must not consider in the tuning. Follow the instructions in `forbidden.txt`. More information about forbidden configurations in [Section 5.6](#). **Important**: If you do not need to define forbidden configurations remove this file from the directory.

7. Place the instances you would like to use for the tuning of your algorithm in the folder `~/tuning/Instances/`. In addition, you can create a file (e.g., `instances-list.txt`) that specifies which instances from that directory should be run and which instance-specific parameters to use. To use such an instance file, set the appropriate option in `scenario.txt`, e.g., `trainInstancesFile = "instances-list.txt"`. See [Section 5.4](#) for guidelines.
8. Uncomment and assign in `scenario.txt` only the options for which you need a value different than the default. The names of the template files match the default names of the scenario options. Some common parameters that you might want to adjust are:

**execDir** (`--exec-dir`): the directory in which **irace** will execute the target algorithm; the default value is the current directory.

**maxExperiments** (`--max-experiments`): the maximum number of executions of the target algorithm that **irace** will perform.

**maxTime** (`--max-time`): the total maximum execution time of the target algorithm. Note that you must provide either **maxTime** or **maxExperiments**.

For setting the tuning budget see [Section 10.1](#). For more information on **irace** options and their default values see [Section 11](#).

9. Modify the `target-runner` script to run your algorithm. This script must execute your algorithm with the parameters and instance specified by **irace** and return the evaluation of the execution and *optionally* the execution time (`cost [time]`). When the **maxTime** option is used, returning time is mandatory. The template we use in this guide is in GNU Bash scripting language, which can be executed easily in GNU/Linux and OS X systems. However, you may use any other programming language. As an example, we provide a Python template in the `$IRACE_HOME/examples/python` directory. Follow these instructions to adjust the given `target-runner` template to your algorithm:

- (a) Set the `EXE` variable with the path to the executable of the target algorithm.
- (b) Set the `FIXED_PARAMS` if you need extra arguments in the execution line of your algorithm. An example could be the time that your algorithm is required to run (`FIXED_PARAMS="--time 60"`) or the number of evaluations required (`FIXED_PARAMS="--evaluations 10000"`).
- (c) The line provided in the template executes the executable described in the `EXE` variable.

```
$EXE ${FIXED_PARAMS} -i ${INSTANCE} --seed ${SEED} ${CONFIG_PARAMS}
```

You must change this line according to the way your algorithm is executed. In this example, the algorithm receives the instance to solve with the flag `-i` and the seed of the random number generator with the flag `--seed`. The variable `CONFIG_PARAMS` adds to the command line the parameters that **irace** has given for the execution. You must set the command line execution as needed. For example, the instance might not need a flag and might need to be the first argument:

```
$EXE ${INSTANCE} ${FIXED_PARAMS} --seed ${SEED} ${CONFIG_PARAMS}
```

The output of your algorithm is saved to the file defined in the `$STDOUT` variable, and error output is saved in the file given by `$STDERR`. The line:

```
if [ -s "$STDOUT" ]; then
```

checks if the file containing the output of your algorithm is not empty. The example provided in the template assumes that your algorithm prints in the last output line the best result found (only a number). The line:

```
COST=$(cat ${STDOUT} | grep -e '^[[:space:]]*[+-]\?[0-9]' | cut -f1)
```

parses the output of your algorithm to obtain the result from the last line. The `target-runner` script must return **only** one number. In the template example, the result is returned with `echo "$COST"` (assuming `maxExperiments` is used) and the used files are deleted.



The `target-runner` script must be executable.

You can test the target runner from the R console by checking the scenario as explained earlier in Section 4

If you have problems related to the `target-runner` script when executing `irace` see [Appendix B](#) for a check list to diagnose common problems. For more information about the `targetRunner` please see [Section 5.2](#),

10. *Optional*: Modify the `target-evaluator` file. You can follow the guidelines provided for defining the `targetEvaluator` in [Section 5.3](#).

Once the files have been prepared you can execute `irace` using the command-line or directly from the R console:

- **On the console**: call the command:

```
cd ~/tuning/  
$IRACE_HOME/bin/irace
```

- **On the R console**: open an R console and execute:

```
> library("irace")  
> # Go to the directory containing the scenario files  
> setwd("~/tuning")  
> # Create the R objects scenario and parameters  
> parameters <- readParameters("parameters.txt")  
> scenario <- readScenario(filename = "scenario.txt",  
+                           scenario = defaultScenario())  
> irace(scenario = scenario, parameters = parameters)
```

This will perform one run of `irace`. See the output of `irace --help` in the command-line or `irace.usage()` in R for quick information on additional `irace` parameters. For more information about `irace` options, see [Section 11](#).



Command-line parameters override the scenario setup specified in the `scenario.txt` file.

## 4.2 Set-up example for ACOTSP

The **ACOTSP** tuning example can be found in the package installation:

```
$IRACE_HOME/examples/acotsp
```

Additionally, a number of example scenarios can be found in the **examples** folder. More examples of tuning scenarios can be found in the Algorithm Configuration Library (Aclib):

<http://www.aclib.net/>

In this section, we describe how to execute the **ACOTSP** scenario. If you wish to start setting up your own scenario, continue in the next section. For this example, we assume a GNU/Linux system but making the necessary changes in the commands and **targetRunner**, it can be executed in any system that has a C compiler. To execute this scenario follow the steps described in the following:

1. Create a directory for the tuning (e.g., `~/tuning/`) and copy the example scenario files located in the **examples** folder to the created directory:

```
mkdir ~/tuning
cd ~/tuning
# $IRACE_HOME is the installation directory of irace.
cp $IRACE_HOME/examples/acotsp/* ~/tuning/
```

2. Download the training instances from <http://iridia.ulb.ac.be/irace/> to the `~/tuning/` directory.
3. Create the instance directory (e.g., `~/tuning/Instances`) and decompress the instance files on it.

```
mkdir ~/tuning/Instances/
cd ~/tuning/
tar -xvf tsp-instances-training.tar.bz2 Instances/
```

4. Download the **ACOTSP** software from <http://www.aco-metaheuristic.org/aco-code/> to the `~/tuning/` directory and compile it.

```
cd ~/tuning/
tar -xvf ACOTSP-1.03.tgz
cd ~/tuning/ACOTSP-1.03
make
```

5. Create a directory for the executable and copy it:

```
mkdir ~/bin/
cp ~/tuning/ACOTSP-1.03/acotsp ~/bin/
```

6. Create a directory for executing the experiments and execute **irace**:

```
mkdir ~/tuning/acotsp-arena/
cd ~/tuning/
# $IRACE_HOME is the installation directory of irace.
$IRACE_HOME/bin/irace
```

7. You can also execute **irace** from the R console using:

```
> library("irace")
> setwd("~/tuning/")
> parameters <- readParameters("parameters-acotsp.txt")
> scenario <- readScenario(filename = "scenario.txt",
+                          scenario = defaultScenario())
> irace(scenario = scenario, parameters = parameters)
```

## 5 irace scenario

### 5.1 Target algorithm parameters

The parameters of the target algorithm are defined by a parameter file as described in [Section 5.1.4](#). Optionally, when executing **irace** from the R console, the parameters can be specified directly as an R object (see [Section 5.1.5](#)). For defining your parameters follow the guidelines provided in the following sections.

#### 5.1.1 Parameter types

Each target parameter has an associated type that defines its domain and the way **irace** handles them internally. Understanding the nature of the domains of the target parameters is important to select appropriate types. The four basic types supported by **irace** are the following:

- *Real* parameters are numerical parameters that can take 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.

### 5.1.2 Parameter domains

For each target parameter, an interval or a set of values must be defined according to its type, as described above. There is no limit for the size of the set or the length of the interval, but keep in mind that larger domains could increase the difficulty of the tuning task. Choose always values that you consider relevant for the tuning. In case of doubt, we recommend to choose larger intervals, as occasionally best parameter settings may be not intuitive a priori. All intervals are considered as closed intervals.

It is possible to define parameters that will have always the same value. Such “*fixed*” parameters will not be tuned but their values are used when executing the target algorithm and they are affected by constraints defined on them. All fixed parameters must be defined as categorical parameters and have a domain of one element.

### 5.1.3 Conditional parameters

Conditional parameters are active only when others have certain values. These dependencies define a hierarchical relation between parameters. For example, the target algorithm may have a parameter `localsearch` that takes values (`sa`, `ts`) and another parameter `ts-length` that only needs to be set if the first parameter takes precisely the value `ts`. Thus, parameter `ts-length` is conditional on `localsearch == "ts"`.

### 5.1.4 Parameter file format

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, e.g., ‘ants’.
- <label> A *label* for this parameter. This is a string that will be passed together with the parameter to `targetRunner`. In the default `targetRunner` provided with the package (Section 5.2), this is the command-line switch used to pass the value of this parameter, for instance “`--ants` ”.  
The value of the parameter is concatenated *without separator* to the switch string when invoking `targetRunner`, thus whitespace is significant. Following the same example, when parameter `ants` takes value 5, the default `targetRunner` will pass the parameter as “`--ants 5`”.
- <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 delimited by parentheses. e.g., (0,1) or (a,b,c,d).
- <condition> An optional *condition* that determines whether the parameter is enabled or disabled, thus making the parameter conditional. If the condition evaluates to false, then no value is assigned to this parameter, and neither the parameter value nor the corresponding label are passed to `targetRunner`. The condition must be a valid R logical expression<sup>2</sup>. 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.

Figure 2 shows as an example the parameters file of the **ACOTSP** scenario.

# 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	(0.00, 5.00)	
beta	--beta "	r	(0.00, 10.00)	
rho	--rho "	r	(0.01, 1.00)	
ants	--ants "	i	(5, 100)	
npls	--npls "	i	(5, 50)	localsearch %in% c(1, 2, 3)
q0	--q0 "	r	(0.0, 1.0)	algorithm == "acs"
dlb	--dlb "	c	(0, 1)	localsearch %in% c(1,2,3)
rasrank	--rasranks "	i	(1, 100)	algorithm == "ras"
elitistants	--elitistants "	i	(1, 750)	algorithm == "eas"

Figure 2: Parameter file (parameters.txt) for tuning **ACOTSP**.

### 5.1.5 Parameters R format

The target parameters are stored in an R list that you can obtain from the R console using the following command:

```
> parameters <- readParameters(file="parameters.txt")
```

See the help of the `readParameters` function (`?readParameters`) for more information. The structure of the parameter list that is created is as follows:

<sup>2</sup>For a quick list of R operators see: <https://stat.ethz.ch/R-manual/R-devel/library/base/html/Syntax.html>

<b>names</b>	Vector that contains the names of the parameters.
<b>types</b>	Vector that contains the type of each parameter 'i', 'c', 'r', 'o'.
<b>switches</b>	Vector that contains the labels of the parameters. e.g., switches to be used for the parameters on the command line.
<b>domain</b>	List of vectors, where each vector may contain two values (minimum, maximum) for real and integer parameters, or a set of values for categorical and ordinal parameters.
<b>conditions</b>	List of R logical expressions, with variables corresponding to parameter names.
<b>isFixed</b>	Logical vector that specifies which parameter is fixed and, thus, it does not need to be tuned.
<b>nbParameters</b>	An integer, the total number of parameters.
<b>nbFixed</b>	An integer, the number of parameters with a fixed value.
<b>nbVariable</b>	Number of variable (i.e., to be tuned) parameters.

The following example shows the structure of the `parameters` R object for the `algorithm`, `ants` and `q0` parameters of the **ACOTSP** scenario:

```
> print(parameters)

# $names
# [1] "algorithm" "ants"      "q0"
#
# $types
# algorithm    ants      q0
#      "c"      "i"      "r"
#
# $switches
# algorithm    ants      q0
#      "--"  "--ants"  "--q0"
#
# $domain
# $domain$algorithm
# [1] "as"  "mmas" "eas" "ras" "acs"
#
# $domain$ants
# [1] 5 100
#
# $domain$q0
# [1] 0 1
#
#
# $conditions
# $conditions$algorithm
# expression(TRUE)
#
# $conditions$ants
# expression(TRUE)
```



```

#
# $conditions$q0
# expression(algorithm %in% c("acs"))
#
#
# $isFixed
# algorithm      ants      q0
#   FALSE      FALSE      FALSE
#
# $nbParameters
# [1] 3
#
# $nbFixed
# [1] 0
#
# $nbVariable
# [1] 3

```

## 5.2 Target algorithm runner

The execution of a candidate configuration on a single instance is done by means of a user-given auxiliary program or, alternatively, a user-given R function. The function (or program name) is specified by the option `targetRunner`. The `targetRunner` must return the evaluation of the execution unless a post-execution evaluation (e.g., multi-objective evaluation) is required, see [Section 5.3](#) for details.



The objective of `irace` is to minimize the obtained evaluations. If you wish to maximize, you can multiply the evaluations by `-1` before returning them.

### 5.2.1 Target runner executable program

When `targetRunner` is an auxiliary executable program, it is invoked for each candidate configuration, passing as arguments:

```
<id.configuration> <id.instance> <seed> <instance> [<extra.params>] <configuration>
```

<code>id.configuration</code>	an alphanumeric string that uniquely identifies a configuration;
<code>id.instance</code>	an alphanumeric string that uniquely identifies a pair <code>&lt;instance, seed&gt;</code> ;
<code>seed</code>	seed for the random number generator to be used for this evaluation, ignore the seed for deterministic algorithms;
<code>instance</code>	string giving the instance to be used for this evaluation;
<code>extra.params</code>	user-defined parameters associated to the instance;
<code>configuration</code>	the pairs parameter label-value that describe this candidate configuration. Typically given as command-line switches to be passed to the executable program.

The experiment list shown in [Section 5.2.2](#), would result in the following execution line:

```
target-runner 1 113 734718556 /home/user/instances/tsp/2000-533.tsp \
--eas --localsearch 0 --alpha 2.92 --beta 3.06 --rho 0.6 --ants 80
```

The command line switches that describe the candidate configuration are 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 `targetRunner` must print a real number, which corresponds to the cost measure of the candidate configuration for the given instance and optionally its execution time (mandatory when `maxTime` is used). The working directory of `targetRunner` is set to the execution directory specified by the option `execDir`. This allows the user to execute independent runs of `irace` in parallel using different values for `execDir`, without the runs interfering with each other.

### 5.2.2 Target runner R function

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

```
> targetRunner(experiment, scenario)
```

where `experiment` is a list that contains the information of candidate and instance to execute one experiment and `scenario` is the scenario list. The structure of the `experiment` list is as follows:

<code>id.configuration</code>	an alphanumeric string that uniquely identifies a configuration;
<code>id.instance</code>	an alphanumeric string that uniquely identifies a pair <code>&lt;instance, seed&gt;</code> ;
<code>seed</code>	seed to be used for this evaluation;
<code>instance</code>	string giving the instance to be used for this evaluation;
<code>extra.params</code>	user-defined parameters associated to the instance;
<code>configuration</code>	1-row data frame with a column per parameter name.
<code>switches</code>	vector of parameter switches in the order of parameters used in <code>configuration</code> .

The following is an example of an experiment list for the **ACOTSP** scenario:

```
> print(experiment)
# $id.configuration
# [1] 1
#
# $id.instance
# [1] 113
#
# $seed
# [1] 734718556
#
# $configuration
#   algorithm localsearch alpha beta rho ants nls q0  dlb
# 1      eas           0 2.92 3.06 0.6  80  NA NA <NA>
```

```

#   rasrank elitistants
# 1   NA           588
#
# $instance
# [1] "/home/user/instances/tsp/2000-533.tsp"
#
# $extra.params
# NULL
#
# $switches
#   algorithm      localsearch      alpha
#   "--" "--localsearch " "--alpha "
#   beta          rho            ants
#   "--beta "      "--rho "      "--ants "
#   nnls          q0              dlb
#   "--nnls "      "--q0 "      "--dlb "
#   rasrank      elitistants
#   "--rasranks " "--elitistants "

```

The function `targetRunner` must return a numerical value corresponding to the evaluation of the candidate configuration on the given instance and optionally the execution time (mandatory when `maxTime` is used).

### 5.3 Target evaluator

The evaluation of the execution of a candidate configuration on an instance must be returned when finalizing the `targetRunner` execution (see [Section 5.2](#)). Nevertheless, there are cases when the evaluation of the candidate configurations must be delayed until all candidate configurations in a race have been executed on a instance.

The `targetEvaluator` parameter is an auxiliary program (or an R function) that allows postponing the evaluations of the candidate configurations. For each instance seen, the evaluation of `targetEvaluator` for any configuration is only invoked after all the calls of `targetRunner` for all alive candidate configurations on the same instance have already finished.



When using `targetEvaluator`, `targetRunner` must not return the evaluation of the configuration. If `maxTime` is used, `targetRunner` must return only execution time.

As an example, `targetEvaluator` may be used to dynamically find normalization bounds for the output returned by an algorithm for each individual instance. In this case, `targetRunner` will save the output of the algorithm, then the first call to `targetEvaluator` will examine the output produced by all calls to `targetRunner` for the same instance, update the normalization bounds and return the normalized output. Subsequent calls to `targetEvaluator` for the same instance will simply return the normalized output.

A similar need arises when using quality measures for multi-objective optimization algorithms, such as the hypervolume, which typically require specifying reference points or sets. By using `targetEvaluator`, it is possible to dynamically compute the reference points or sets while `irace` is running. Examples are provided at [examples/hypervolume](#). See also [Section 10.2](#) for more information on how to tune multi-objective algorithms.

### 5.3.1 Target evaluator R function

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

```
> targetEvaluator(experiment, num.configurations, all.conf.id, scenario,  
+                target.runner.call)
```

where `experiment` is a list that contains the information of one experiment (See [Section 5.2.2](#)), `num.configurations` is the number of configurations alive on the race, `all.conf.id` is the list of the alive candidates configurations ids, `scenario` is the scenario list and `target.tunner.call` is the string of the `targetRunner` execution line.

The function `targetEvaluator` must return a numerical value corresponding to the cost measure of the candidate configuration on the given instance.

### 5.3.2 Target evaluator executable program

When `targetEvaluator` is an auxiliary executable program, then it is invoked for each candidate with the following arguments:

```
<id.configuration> <id.instance> <seed> <instance> <num.configurations> <all.conf.id>
```

<code>id.configuration</code>	an alphanumeric string that uniquely identifies a configuration;
<code>id.instance</code>	an alphanumeric string that uniquely identifies a pair <code>&lt;instance, seed&gt;</code> ;
<code>seed</code>	seed to be used for this evaluation;
<code>instance</code>	string giving the instance to be used for this evaluation;
<code>num.configurations</code>	number of alive candidate configurations;
<code>all.conf.id</code>	list of IDs of the alive configurations separated by whitespace.

The `targetEvaluator` executable must print a numerical value corresponding to the cost measure of the candidate configuration on the given instance.

## 5.4 Training instances

The `irace` options `trainInstancesDir` and `trainInstancesFile` specify where to find the training instances.

By default, the value of `trainInstancesFile` is empty. This means that `irace` will consider all files within the directory given by `trainInstancesDir` (by default `./Instances`) as training instances.

Otherwise, the value of `trainInstancesFile` may specify a text file. The format of this file is one instance per line, and the first alphanumeric string of each line corresponds to the instance filename. The remainder text within each line are considered as extra parameters to be supplied to `targetRunner` for this specific instance. The following example shows a training instance file for the **ACOTSP** scenario:

The value of `trainInstancesDir` is always prefixed to the instance name, that is, the instances names are treated as relative to this directory. For example, given the above file as `trainInstancesFile` and the default value of `trainInstancesDir` (`./Instances`), then a possible invocation of `targetRunner` would be:

```
# Example training instances file
100/100-1_100-2.tsp --time 1
100/100-1_100-3.tsp --time 2
100/100-1_100-4.tsp --time 3
```

Figure 3: Training instances file for tuning **ACOTSP**.

```
target-runner 1 113 734718556 ./Instances/100/100-1_100-2.tsp --time 1 --alpha 2.92 ...
```

Training instances do not need to be files, **irace** just passes their names to **targetRunner**, thus the names can denote benchmark functions or descriptive labels that the target algorithm understands. The extra instance parameters could actually be the definition of the instance. In that case, **trainInstancesDir** is usually set to the empty string (`--train-instances-dir=""`). For example,

```
# Example training instances file
rosenbrock_20 --function=12 --nvar 20
rosenbrock_30 --function=12 --nvar 30
rastrigin_20 --function=15 --nvar 20
rastrigin_30 --function=15 --nvar 30
```

Optionally, when executing **irace** from the R console, the list of instances and their specific parameters might be provided explicitly by means of the variables `scenario$instances` and `scenario$instances.extra-params`, respectively. Thus, the previous example would be equivalent to:

```
> scenario$instances <- c("rosenbrock_20", "rosenbrock_40",
+ "rastrigin_20", "rastrigin_40")
> scenario$instances.extra.params <-
+ c("--function=12 --nvar 20", "--function=12 --nvar 30",
+ "--function=15 --nvar 20", "--function=15 --nvar 30")
```

By default, **irace** assumes that the target algorithm is stochastic (the value of the option **deterministic** is 0), thus, the same configuration can be executed more than once on the same instance and obtain different results. In this case, **irace** generates pairs `<instance,seed>` by assigning a random seed to each instance. Once all pairs have been seen within a run of **irace**, new pairs are generated with different seeds.

If **deterministic** is set to 1, then each instance will be used at most once per race. This should only be used for target algorithms that do not have a stochastic behavior and, therefore, executing the target algorithm on the same instance several times with different seeds does not make sense.



If **deterministic** is active and the number of training instances provided to **irace** is less than **firstTest** (default: 5), no statistical test will be performed on the race.

Finally, **irace** randomly re-orders the sequence of instances provided. This random sampling may be disabled by using the option **sampleInstances** (`--sample-instances 0`) if keeping the order provided in the instance file is important.



We advise to always sample instances to prevent biasing the tuning due to the instance order.

## 5.5 Initial configurations

The scenario option `configurationsFile` allows specifying a text file that contains an initial set of configurations to start the execution of **irace**. If the number of initial configurations supplied in the file is less than the number of configurations required by **irace** in the first iteration, additional configurations will be sampled uniformly at random.

The format of the configurations file is one configuration per line, and one parameter value per column. The first line must give the parameter name corresponding to each column (names must match those given in the parameters file). Each configuration must satisfy the parameter conditions (NA should be used for those parameters that are not enabled for a given configuration) and not be forbidden by the constraints that define forbidden configurations (Section 5.6), if any.

Figure 4 gives an example file that corresponds to the **ACOTSP** scenario:

```
## Initial candidate configuration for irace
algorithm localsearch alpha beta rho ants nls dlb q0 rasrank elitistants
as          0           1.0  1.0  0.95 10  NA  NA  0  NA      NA
```

Figure 4: Initial configuration file (`default.txt`) for tuning **ACOTSP**.

We advise to use this feature when a default configuration of the target algorithm exists or when different sets of good parameter values are known. This will allow **irace** to start the search from those parameter values and attempt to improve their performance.

## 5.6 Forbidden configurations

The scenario option `forbiddenFile` specifies a text file containing logical expressions of parameter values that valid configurations should not satisfy, that is, no configuration that satisfies any of these logical expressions will be evaluated by **irace**. This is useful when some combination of parameter values could cause the target algorithm to crash, consume excessive CPU time or memory, or when it is known that they do not produce satisfactory results.

The format of the forbidden configurations file is one constraint per line, where each constraint is a logical expression (in R syntax) containing parameter names as defined by the `parameterFile` (Section 5.1), values and logical operators. For a quick list of R logical operators see:

<https://stat.ethz.ch/R-manual/R-devel/library/base/html/Syntax.html>

If a parameter configuration is generated that makes any of the logical expressions evaluate to TRUE, then the configuration is considered forbidden and it is discarded. Figure 5 shows an example file that corresponds to the **ACOTSP** scenario:

```
## Examples of valid logical operators are:
## == != >= <= > < & | ! %in%
(alpha == 0.0) & (beta == 0.0)
```

Figure 5: Forbidden configurations file (`forbidden.txt`) for tuning **ACOTSP**.



If initial configuration are provided (Section 5.5), they must also comply with the constraints defined in `forbiddenFile`.

## 6 Parallelization

A single run of **irace** can be done much faster by executing the calls to **targetRunner** (the runs of the target algorithm) in parallel. There are four ways to parallelize a single run of **irace**:

- **Parallel processes:** The option **parallel** allows executing in parallel, within a single computer, the calls to **targetRunner**, by means of the **parallel** R package. For example, adding **--parallel N** to the command line of **irace** will launch in parallel up to  $N$  calls of the target algorithm.
- **MPI:** By enabling the option **mpi**, calls to **targetRunner** will be executed in parallel by using the message passing interface (MPI) protocol (requires the **Rmpi** R package). In this case, the option **parallel** controls the number of slave nodes used by **irace**. For example, adding **--mpi 1 --parallel N** to the command-line will create  $N$  slaves + 1 master, and execute up to  $N$  calls of **targetRunner** in parallel.

The user is responsible for setting up the required MPI environment. MPI is commonly available in computing clusters and requires launching **irace** in some particular way. An example script for using MPI mode in a SGE cluster is given at `$IRACE_HOME/examples/mpi/`.

- **SGE cluster:** This mode uses the commands **qsub** and **qstat** often found in Sun Grid Engine (SGE) and compatible clusters. The command **qsub** must return a message that contains the string: "Your job JOBID", where JOBID is a unique identifier for the job submitted. The command **qstat -j JOBID** must return nonzero if JOBID has finished its execution, and zero otherwise.

Enabling the option **sgeCluster** (**--sge-cluster 1**) will launch as many calls of **targetRunner** as possible and use **qstat** to wait for cluster jobs. In this mode, **irace** must run in the submission node of the cluster, and hence, **qsub** should not be used to invoke **irace** itself. The user must call **qsub** from within **targetRunner** with the appropriate settings for their cluster, otherwise **targetRunner** will not submit jobs to the cluster. Moreover, the use of a separate **targetEvaluator** script is required to parse the results of **targetRunner** and return them to **irace**. See the examples in `$IRACE_HOME/examples/sge-cluster/`.

- **targetRunnerParallel:** This option allows users to fully control the parallelization of **targetRunner**. Its value must be an R function that will be invoked by **irace** as follows:

```
targetRunnerParallel(experiments, targetRunner, scenario)
```

where **experiments** is a list that contains elements with the information of configurations and instances to be executed (see [Section 5.2](#) for a description), **targetRunner** is the **targetRunner** script or function and **scenario** is the scenario list. The **targetRunnerParallel** function must execute the given **targetRunner** using the **experiments** and **scenario** provided, and return a list of the same length as **experiments** containing the output of each call to **targetRunner**.

## 7 Testing of configurations

Once the tuning process is finished, **irace** commonly returns a set of configurations corresponding to the elite configurations at the end of the run, ordered from best to worst. To further

investigate the quality of these configurations, **irace** offers the possibility of evaluating these configurations on a test instance set, typically different from the training set used during the tuning phase. These evaluations will use the same settings for parallel execution, `targetRunner` and `targetEvaluator`.

The test instance set can be specified by the options `testInstancesDir` and `testInstancesFile`, or by setting directly the variable `scenario$testInstances`, which behave the same as their counterparts for the training instances (Section 5.4). In particular, each test instance is assigned a different seed in the same way as done for the training instances.

The options `testNbElites` and `testIterationElites` control which configurations are evaluated during the testing phase. In particular, setting `testIterationElites = 1` will test not only the final set of elite configurations (those returned at the end of the training phase), but also the set of elites at the end of each race (iteration). The option `testNbElites` limits the maximum number of configurations considered within each set. Some examples:

- `testIterationElites = 0; testNbElites = 1` means that only the best configuration found during the run of **irace**, the final best, will be used in the testing phase.
- `testIterationElites = 1; testNbElites = 1` will test, in addition to the final best, the best configuration found at each iteration.
- `testIterationElites = 1; testNbElites = 2` will test the two best configurations found at each iteration, in addition to the final best and second-best configurations.

The testing can be also (re-)executed at a later time by using the following R command:

```
> testing.main(logFile = "./irace.Rdata")
```

This line will load the **irace** results found in the generated `logFile` file to perform the testing. The testing results will be saved in the **irace** log file specified in `scenario$logFile` in the `iraceResults$testing` R object. The structure of the object is described in Section 9.2. For examples on how to analyse the data see Section 9.3.

## 8 Recovering irace runs

Problems like power cuts, hardware malfunction or the need to use computational power for other tasks may occur during the execution of **irace**, terminating a run before completion. At the end of each iteration, **irace** saves an R data file (`logFile`, by default `./irace.Rdata`) that not only contains the information of the tuning progress (Section 9.2), but also internal information that allows recovering an incomplete execution.

To recover an incomplete **irace** run, set the option `recoveryFile` to the log file previously produced, and **irace** will continue the execution from the last saved iteration. The state of the random generator is saved and loaded, therefore, as long as the execution is continued in the same machine, the obtained results will be exactly the same as executing **irace** in one step (although external factors, such as CPU load and disk caches, may affect the target algorithm and that may affect the results). You can specify the `recoveryFile` from the command-line or from the scenario file, and execute **irace** as described in Section 4. For example, from the command-line use:



```
irace --recovery-file "./irace-backup.Rdata"
```



When recovering a previous run, **irace** will try to save data on the file specified by the `logFile` option. Thus, you must specify different files for `logFile` and `recoveryFile`. Before recovering, we strongly advise to rename the saved R data file as in the example above, which uses `"irace-backup.Rdata"`.



Do not change anything in the log file or the scenario file before recovering, as it may have unexpected effects on the recovered run of **irace**. In case of doubt, please contact us first (Section 13).

## 9 Output and results

During its execution, **irace** prints information about the progress of the tuning in the standard output. Additionally, after each iteration, an R data file is saved (`logFile` option) containing the state of **irace**.

### 9.1 Text output

Figure 6 shows the output, up to the end of the first iteration, of a run of elitist **irace** applied to the **ACOTSP** scenario with 1000 evaluations as budget.

First, **irace** gives the user a warning informing that it has found a file with the default scenario file name and it will use it. Then, general information about the selected **irace** options is printed:

- `nbIterations` indicates the minimum number of iterations **irace** has calculated for the scenario. Depending on the development of the tuning the final iterations that are executed can be more.
- `minNbSurvival` indicates the minimum number of alive configurations that are required to continue a race. When less configurations are alive the race is stopped and a new iteration begins.
- `nbParameters` is the number of parameters of the scenario.
- `seed` is the number that was used to initialize the random number generator in **irace**.
- `confidence level` is the confidence level of the statistical test.
- `budget` is the total number of evaluations available for the tuning.
- `time budget` maximum execution time available for the tuning.
- `mu` is a value used for calculating the minimum number of iterations.
- `deterministic` indicates if the target algorithm has been marked as deterministic.

In each iteration information about the progress of the execution printed as follows:

- `experimentsUsedSoFar` is the number of experiments from the total budget has been used until the actual iteration.
- `timeUsed` time used so far in the experiments. Only available when reported in the `targetRunner` (activate it with `maxTime` option).

```

*****
Warning: A default scenario file './scenario.txt' has been found and will be read
# 2016-05-02 19:24:50 CEST: Elitist race
# Elitist instances: 1
# Elitist limit: 2

# 2016-05-02 19:24:50 CEST: Initialization
# nbIterations: 5
# minNbSurvival: 5
# nbParameters: 11
# seed: 1234
# confidence level: 0.95
# budget: 1000
# time budget: 0
# mu: 5
# deterministic: FALSE

# 2016-05-02 19:24:50 CEST: Iteration 1 of 5
# experimentsUsedSoFar: 0
# remainingBudget: 1000
# currentBudget: 200
# nbConfigurations: 33
Markers:
  x No test is performed.
  - The test is performed and some configurations are discarded.
  = The test is performed but no configuration is discarded.
  ! The test is performed and configurations could be discarded but elite
    configurations are preserved.

+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Instance| Alive| Best| Mean best| Exp so far| W time| rho|KenW| Qvar|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|x|      1|  33|  15| 23268924.00|      33|00:01:55|  NA|  NA|  NA|
|x|      2|  33|   8| 23185736.50|     66|00:01:53|+0.97|0.99|0.0025|
|x|      3|  33|   8| 23239054.33|     99|00:01:56|+0.96|0.97|0.0030|
|x|      4|  33|   8| 23168442.50|    132|00:01:55|+0.96|0.97|0.0027|
|-|      5|   3|   8| 23222299.80|    165|00:01:56|-0.05|0.16|0.7109|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
Best configuration:           8      mean value:      23222299.80
Description of the best configuration:
.ID. algorithm localsearch alpha beta rho ants nmls q0 dlb rasrank
8      8      acs          1 3.8157 8.5915 0.4141 59 10 0.5812 1      NA
elitistants .PARENT.
           NA      NA

# 2016-05-02 19:34:27 CEST: Elite configurations:
algorithm localsearch alpha beta rho ants nmls q0 dlb rasrank elitistants
8      acs          1 3.8157 8.5915 0.4141 59 10 0.5812 1      NA      NA
18     mmas         2 3.1134 7.3864 0.4623 60 32      NA  1      NA      NA
15     ras          3 2.5838 6.5086 0.5082 42  6      NA  0     90      NA

```

Figure 6: Sample text output of **irace**.

- `remainingBudget` is the number of evaluations that have not been used yet.
- `timeEstimate` estimation of the mean execution time, this is used to calculate the remaining budget when `maxTime` is used.
- `currentBudget` is the number of evaluations **irace** has allocated to the current iteration.

- **nbConfigurations** is the number of configurations **irace** will use in the current iteration. On the first iteration this number of configurations include the initial configurations provided and in later iterations includes the elite candidates of the previous iterations.

After the iteration information a table shows the progress of the iteration execution. Each row of the table gives information about the execution of an instance in the race. The first column contains a symbol that describes the results or non application of the statistical test:

In each iteration is initially printed information about the progress of the execution:

- **|x|** : No statistical test was performed for this instance. To adjust in which instances of a race statistical tests are performed see **irace** options **firstTest** and **eachTest** in [Section 11](#).
- **|-|** : Statistical test performed and configurations have been discarded. To know how many configurations have been discarded see the table column **Alive**.
- **|=|** : Statistical test performed and no configurations have been discarded. This means **irace** needs more information to identify the best configurations.
- **|||**: This indicator exists only for the elitist version of **irace**. It indicates that the statistical test was performed and some elite configurations show bad performance and could be discarded but they are kept given the elitist **irace** discarding rules. See **irace** option **elitist** in [Section 11](#) for more information.

The **instance** column gives the number of `<instance,seed>` pair executed. This number corresponds to the index of the list found in `state$.irace$instancesList`. See [Section 9.2](#) for more information.

The **Alive** column gives the number of configurations that have not been discarded after the statistical test was performed. The column **Best** gives the id of the best configuration according to the experiments performed so far in the race (includes previous instances). The **Mean best** column gives the mean of the best configuration across all the instances executed so far in the race. The **Exp so far** gives the number of evaluations performed so far. The **W time** column gives the waiting time to execute all the configurations in the current instance.

The columns **rho** and **KenW** give the values of the Spearman's rho and the Kendall concordance coefficient of the configurations across the instances executed so far. The **Qvar** gives the variance measure across the instances. Use **rho**, **KenW** and **Qvar** to analyze how consistent is the performance of the configurations across the instances. Note that these values are only valid for the instances that were already executed in the iteration. Values close to 1 for **rho** and **KenW** and values close to 0 for the **Qvar** indicate that the performance is consistent and therefore the scenario is homogeneous. For heterogeneous scenarios we provide advice in [Section 10.4](#).

Finally **irace** outputs the best configuration found and a list of the elite configurations. The elite configurations are configurations that did not show statistically significant difference during the race; they are ordered according to their mean performance on the executed instances.

## 9.2 Data file output

The R data file created by **irace** (**logFile**) contains an object called **iraceResults**. You can load this data in the R console by:

```
> load("irace-output.Rdata")
```

The **iraceResults** object is a list, the elements of a list can be accessed in R by using the **\$** or **[[]]** operators:

```

> iraceResults$irace.version
# [1] "2.0.1397M"
> iraceResults[["irace.version"]]
# [1] "2.0.1397M"

```

The `iraceResults` list contains the following elements:

- **scenario**: The scenario R object containing the **irace** options used for the execution. See [Section 11](#) and the help of the **irace** package; open an R console and type: `?defaultScenario`. See [Section 11](#) for more information.
- **parameters**: The parameters R object containing the description of the target algorithm parameters. See [Section 5.1](#).
- **allConfigurations**: The target algorithm configurations generated by **irace**. This object is a **data frame**, each row is a candidate configuration; the first column (`.ID.`) indicates the internal identifier of the configuration; the following columns correspond to the parameter values; each column is named as the parameter name specified in the parameter object. The final column (`.PARENT.`) is the identifier of the configuration from which model the actual configuration was sampled.

```

> head(iraceResults$allConfigurations)

#   .ID. algorithm localsearch  alpha  beta  rho ants nnls
# 1    1      as              2 2.9953 0.6188 0.7023 75  6
# 2    2     mmas              0 4.6876 8.2611 0.1948 44 NA
# 3    3     mmas              1 3.6487 6.7212 0.2986 91 44
# 4    4      as              0 4.5230 3.3080 0.3026 58 NA
# 5    5      as              2 2.6748 3.2815 0.4874  5 25
# 6    6     mmas              0 3.0051 5.1321 0.5918 24 NA
#   q0  dlb rasrank elitistants .PARENT.
# 1 NA   1    NA      NA      NA
# 2 NA <NA>    NA      NA      NA
# 3 NA   1    NA      NA      NA
# 4 NA <NA>    NA      NA      NA
# 5 NA   0    NA      NA      NA
# 6 NA <NA>    NA      NA      NA

```

- **allElites**: A list that contains one element per iteration. Each element contains the internal identifier of the elite candidate configurations of the corresponding iteration (identifiers correspond to `allConfigurations$.ID.`).

```

> print(iraceResults$allElites)

# [[1]]
# [1] 8 18 15

```

```

#
# [[2]]
# [1] 47
#
# [[3]]
# [1] 47 67 70 69
#
# [[4]]
# [1] 47 118 70 96 95
#
# [[5]]
# [1] 118 154 47 119 95
#
# [[6]]
# [1] 118 47 95 164 156
#
# [[7]]
# [1] 95 47 164 156 118

```

The configurations are ordered by mean performance, that is, the id of the best configuration corresponds to the first id. To obtain the values of the parameters of all elite configurations found by **irace** use:

```

> getFinalElites(irace.logFile="irace-output.Rdata", n=0)

#   .ID. algorithm localsearch  alpha  beta  rho ants nnls
# 1   95      acs             3 2.1078 3.5827 0.399  44  30
# 2   47      acs             3 1.5946 2.6973 0.7878  37  31
# 3  164      acs             3 2.0925 3.7981 0.9098  18  34
# 4  156      acs             3 1.4171 5.3167 0.5322  23  31
# 5  118      acs             3 1.5666 5.7256 0.6368  27  43
#      q0 dlb rasrank elitistants .PARENT.
# 1 0.3813  1      NA           NA      47
# 2 0.2983  1      NA           NA      15
# 3 0.2981  1      NA           NA      47
# 4 0.3379  1      NA           NA     118
# 5 0.1491  1      NA           NA      70

```

- **iterationElites**: A vector containing the best candidate configuration internal identifier of each iteration. The best configuration found corresponds to the last one of this vector.

```

> print(iraceResults$iterationElites)

# [1] 8 47 47 47 118 118 95

```

Obtain the full configuration with:

```

> last <- length(iraceResults$iterationElites)
> id <- iraceResults$iterationElites[last]
> getConfigurationById(irace.logFile="irace-output.Rdata",
+                       ids=id)

#   .ID. algorithm localsearch  alpha  beta  rho ants nnls
# 95   95      acs              3 2.1078 3.5827 0.399  44  30
#      q0 dlb rasrank elitistants .PARENT.
# 95 0.3813  1      NA              NA      47

```

- **experiments**: A matrix with configurations as columns and instances as rows. Column names correspond to the internal identifier of the configuration (`allConfigurations$.ID.`). To obtain the experiment results of a particular configuration use:

```

> # As an example, we use the best configuration found
> best.config <- getFinalElites(iraceResults=iraceResults,
+                               n=1)
> id <- best.config$.ID.
> # Obtain the configurations using the identifier
> # of the best configuration
> all.exp <- iraceResults$experiments[,as.character(id)]
> all.exp[!is.na(all.exp)]

#      1      2      3      4      5      6
# 23143448 22959710 23284140 22858335 23226582 23264551
#      7      8      9     10     11     12
# 23439606 23108045 23388942 23097102 23069267 23300829
#      13     14     15
# 23079207 23384244 23104481

```

When a configuration was not executed on an instance there is a `NA` value in the corresponding matrix cell. A configuration is not executed on an instance for three different reasons: 1) because it was not created yet when the instance was used (only for the non elitist **irace**) or 2) because it was discarded by the statistical test or 3) the race was terminated before the instance could reach the execution of the instance.

The row names correspond to the identifier of the `<instance,seed>` pairs defined in `state$.irace$instancesList`. To obtain the instance and seed used for a particular experiment use:

```

> # As an example, we get seed and instance of the experiments
> # of the best candidate.
> # Get index of the instances
> pair.id <- names(all.exp[!is.na(all.exp)])
> index <-
+   iraceResults$state$.irace$instancesList[pair.id,"instance"]
> # Obtain the instance names
> iraceResults$scenario$instances[index]

```

```

# character(0)

> # Get the seeds
> iraceResults$state$.irace$instancesList[index,"seed"]

# NULL

```

- **experimentLog**: A matrix with columns:

`<iteration,instance,configuration>`.

This matrix contains the log of all the experiments that **irace** performs during its execution. The instance column refers to the index of the `state$.irace$instancesList` data frame.

- **softRestart**: A logical vector that indicates if a soft restart was performed on each iteration. If `FALSE`, then no soft restart was performed. For info about soft restart see [Section 11](#).
- **state**: A list that contains the state of **irace**, the recovery ([Section 8](#)) is done using the information contained in this object. The probabilistic model of the last elite configurations can be found here doing:

```

> # As an example, we get the model probabilities for the
> # localsearch parameter.
> iraceResults$state$model["localsearch"]

# $localsearch
# $localsearch$`118`
# [1] 0.0002285714 0.0002285714 0.0002285714 1.0000000000
#
# $localsearch$`47`
# [1] 0.0002285714 0.0002285714 0.0002285714 1.0000000000
#
# $localsearch$`95`
# [1] 0.0002285714 0.0002285714 0.0002285714 1.0000000000
#
# $localsearch$`164`
# [1] 0.0002285714 0.0002285714 0.0002285714 1.0000000000
#
# $localsearch$`156`
# [1] 0.0002285714 0.0002285714 0.0002285714 1.0000000000

> # The order of the probabilities corresponds to:
> iraceResults$parameters$domain$localsearch

# [1] "0" "1" "2" "3"

```

The example shows a list that has one element per elite configuration (id as element name). In this case, `localsearch` is a categorical parameter and it has a probability per each value.

- **testing**: A list that contains the testing results. The list contains the following elements:
  - **experiments**: Matrix of experiments in the same format as the tuning **experiment** matrix. The column names indicate the candidate configuration identifier and the row names contain the name of the instances.

```
> # Get the experiments of the testing
> iraceResults$testing$experiments

#           95          47          164          156          118
# 1000-1.tsp 23409880 23410576 23422748 23404330 23366881
# 1000-2.tsp 23126916 23144708 23212491 23100817 23196172
# 1000-3.tsp 23084684 23076230 23110653 23086263 23098047
# 1000-4.tsp 23251050 23232485 23232201 23228093 23234648
# 1000-5.tsp 23278985 23287118 23330620 23295181 23336133
# 1000-6.tsp 22983127 22959425 23073906 22989811 22951321
# 1000-7.tsp 23087699 23124945 23115738 23122621 23084600
# 1000-8.tsp 22893110 22850619 22863202 22865437 22913600
# 1000-9.tsp 23180624 23174994 23209675 23227996 23205438
# 1000-10.tsp 23367381 23333064 23405137 23341862 23356161
```

- **seeds**: The seeds used for the experiments, each seed corresponds to each instance in the rows of the test **experiments** matrix.

```
> # Get the experiments of the testing
> iraceResults$testing$seeds

# [1] 716498671 999094119 1613704058 978659676 1046072282
# [6] 1418237375 466610226 249199596 828014021 1647202103
```

In the example instance `1000-1.tsp` is executed with seed `1815573416`.

### 9.3 Analysis of results

The best configurations provided by **irace** are configurations that were found to be not statistically significantly different. The configurations are reported in average performance order, that is, the best by mean configuration is reported first.

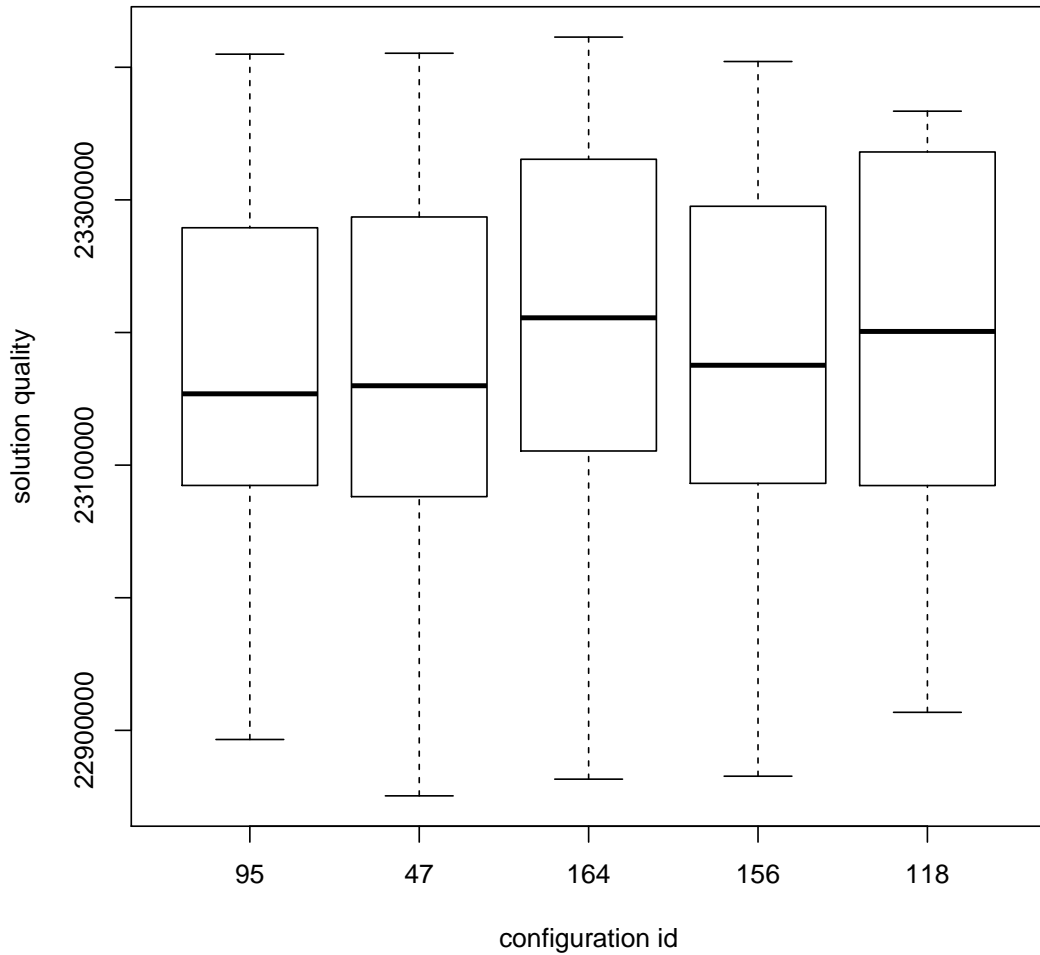
If testing is performed you can further analyze the resulting best configurations by performing statistical tests in R or just plotting the results:

```
> results <- iraceResults$testing$experiments
> # Wilcoxon paired test
> conf <- gl(ncol(results), #number of configurations
+          nrow(results), #number of instances
+          labels=colnames(results))
> pairwise.wilcox.test (as.vector(results),
+                       conf,
+                       paired=TRUE,
+                       p.adj = "bonf")
#
```



```
# Pairwise comparisons using Wilcoxon signed rank test
#
# data: as.vector(results) and conf
#
#      95    47    164    156
# 47 1.000 -      -      -
# 164 0.488 0.098 -      -
# 156 1.000 1.000 0.488 -
# 118 1.000 1.000 1.000 1.000
#
# P value adjustment method: bonferroni

> # Plot the results
> boxplot (iraceResults$testing$experiments,
+         ylab="solution quality",
+         xlab="configuration id")
```



During the tuning, **irace** iteratively updates sampling models for the parameters focusing on the best areas of the parameter search space. The frequency of the sampled configurations can provide insight on the parameter search space. We provide a function that allows to create plots that show the frequency of the sampling of a set of configurations. The following example plots the frequency of the parameters sampled during all the **irace** execution:

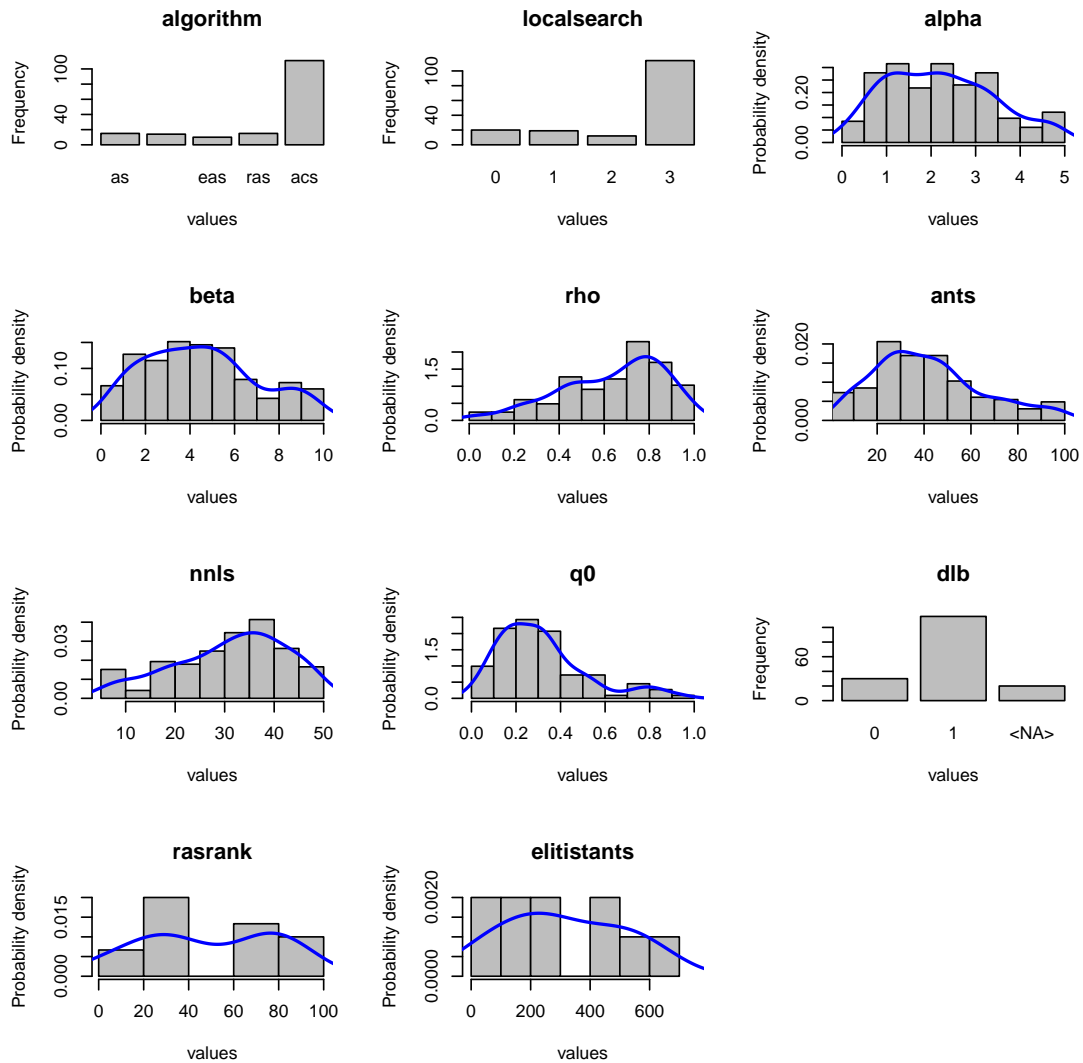
```
> parameterFrequency(iraceResults$allConfigurations,
+                    iraceResults$parameters)

# Plotting: algorithm
# Plotting: localsearch
# Plotting: alpha
# Plotting: beta
# Plotting: rho
```

```

# Plotting: ants
# Plotting: nnls
# Plotting: q0
# Plotting: dlb
# Plotting: rasrank
# Plotting: elitistants

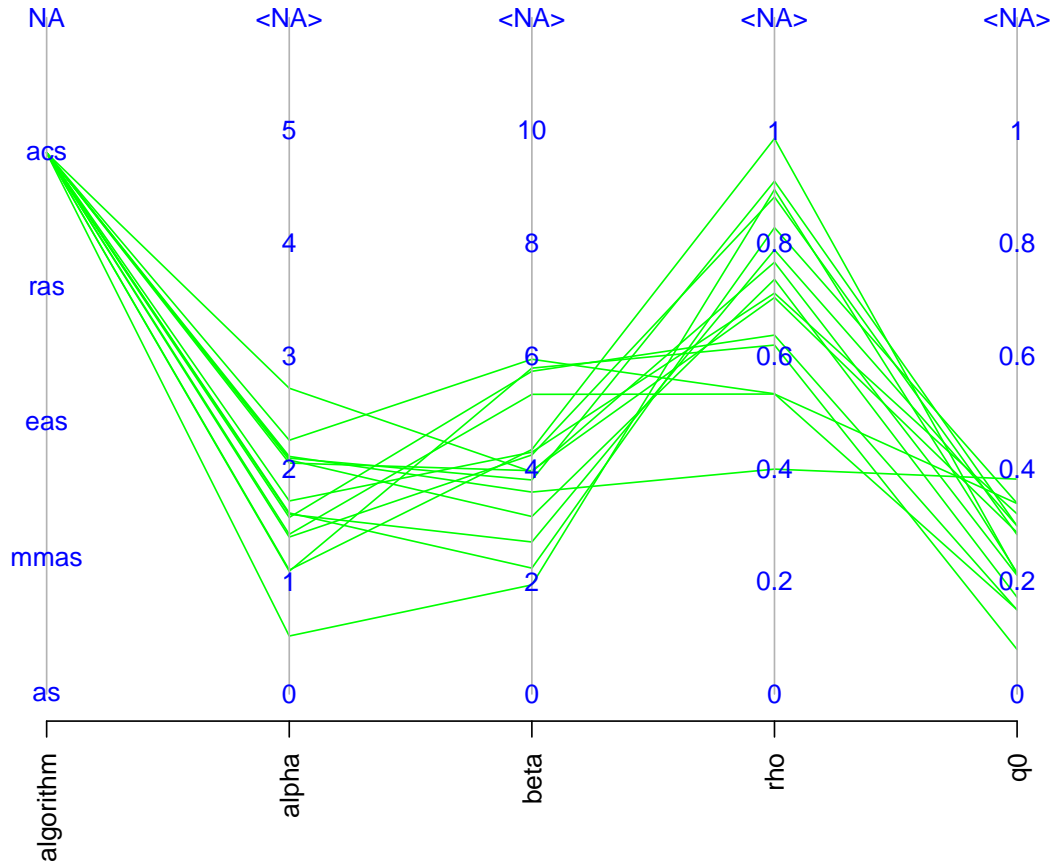
```



For more information of this function please see the R help, type in the R console: `?parameterFrequency`.  
Using parallel coordinates plots it is possible to analyze how the parameters interact with each other. The following example shows how to create a parallel coordinate plot of the candidates of the last two iterations of **irace**.

```
> # Get last iteration number
> last <- length(iraceResults$iterationElites)
> lasts <- c(last-1, last)
> # Get last iterations candidates
> conf <- getConfigurationByIteration(iraceResults = iraceResults,
+                                     iterations = lasts)
> parallelCoordinatesPlot (conf,
+                           iraceResults$parameters,
+                           param_names=c("algorithm",
+                                           "alpha",
+                                           "beta",
+                                           "rho",
+                                           "q0"),
+                           hierarchy=FALSE)
```

## Parameters parallel coordinates



For more information of this function please see the R help, type in the R console: (?parallelCoordinatesPlot).

## 10 Advanced topics

### 10.1 Tuning budget

**irace** provides two options for setting the total tuning budget (`maxExperiments` and `maxTime`). Consider the number of parameters that need to be tuned, available processing power and available time before setting the budget for the tuning. The option `maxExperiments` limits the number of executions of `targetRunner` performed by **irace**. The option `maxTime` limits the total time of the `targetRunner` executions, when this option is used `targetRunner` must return the evaluation cost together with the execution time (`<cost><time>`).



When tuning computation time and using `maxTime` as tuning budget just return the time as the evaluation cost (`< time >< time >`).



When using `targetEvaluator` and using `maxTime` as tuning budget just return the time (`< time >`).

When using `maxTime`, `irace` estimates the execution time of each `targetRunner` execution before the configuration. The amount of budget used for the estimation is set with the option `budgetEstimation` (default is 2%). The obtained estimation is adjusted after each iteration using the obtained results and is used to estimate the number of experiments that can be executed. Internally, `irace` uses the number of remaining experiments to adjust the number of configurations tested in each race.

## 10.2 Multi-Objective tuning

`irace` performs the automatic configuration of an algorithm optimizing only one objective that can be solution quality, computation time or any other objective and that is returned to `irace` through the `targetRunner`.

If you wish to tune your algorithm with `irace` for more than one objective there are two alternatives:

- Aggregate the objectives in one resulting number.
- Use a unary (e.g., the hypervolume) indicator for evaluating the quality of the configurations.

The first option is simple, it requires to devise a formula that can aggregate the objectives in a way that balances the importance of all of them. This might not be an easy task in some scenarios, and therefore using a more adequate indicator to evaluate the performance of a multi-objective optimizer, such as the hypervolume, is strongly advised.

For setting up the multi-objective tuning you must not return the evaluation of the experiment when finalizing the execution of `targetRunner` (see Section 5.2) and specify a `targetEvaluator` in which the reference points are obtained and the hypervolume is calculated. For more information about defining a `targetEvaluator` see Section 5.3. For the hypervolume calculation we suggest the following implementation:

<http://lopez-ibanez.eu/hypervolume>

Examples of a multi-objective tuning using the hypervolume can be found in the templates:

```
$IRACE_HOME/examples/hypervolume
$IRACE_HOME/examples/moaco
```

## 10.3 Tuning computation time

`irace` was developed primarily for tuning solution quality after a given amount of computation time. To use `irace` for tuning computation time, the execution time of a configuration must be returned as result by the `targetRunner`. Even though `irace` can be used for tuning computation time, it may not be the best method to do so as in its current version it does not use an “adaptive capping” mechanism as do ParamILS or SMAC. These latter two configurators may be obtained in the following links:

<http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/>  
<http://www.cs.ubc.ca/labs/beta/Projects/SMAC/>

We are currently extending **irace** by an adaptive capping mechanism and it will likely be included in the forthcoming version 2.1.

## 10.4 Heterogeneous scenarios

We classify a scenario as homogeneous when the target algorithm has a consistent performance regarding the instances that is, good configurations tend to be good for all instances. On the contrary, for heterogeneous scenarios the target algorithm has an inconsistent performance on different instances, that is, some configurations are good for one or a subset of instances while are very bad for another subset of instances.

If you know your scenario has heterogeneous characteristics, the first question you should ask yourself is if the tuning objective is to find configurations that are good for all instances. If this is not the case, then separating executions of **irace**, one per instance type, is the best choice. You may combine in this case configurations obtained by **irace** with algorithm selection techniques that help to find the right configuration for each class of instances.

If finding a good configuration for all the instances is the objective, then we recommend to always sample the instances initially (option `sampleInstances`) unless you provide an instance order that does not bias the search. For example, assume you have an scenario that has two kinds of instances, if the ten first instances belong to only one class, the search will be biased to obtain configurations that are good for those instances. The best order in this case would be to intercalate different types of instances to avoid bias.

Another advice is to increase the number of instances executed per iteration, as an heterogeneous scenario will need to gather more information about the different instances before discarding configurations. Use the option `elitistInstances` (default value is 1) when the elitist **irace** version (option `elitist`) is used to increase the number of new instances executed in each iteration (e.g., `-elitist-instances 5`). When using the non elitist **irace** version you can indirectly increase the number of instances by increasing the `firstTest` (default is 5) option (e.g., `-first-test 10`).

When executing **irace** you can analyze the homogeneity of the scenario by observing the results of the Kendall W and Spearman's rho in the text output of **irace**. See [Section 9.1](#) for more information.

## 10.5 Choosing the statistical test

The statistical test identifies statistically bad performing configurations that **irace** can discard from the race in order to save budget. The criterion that is used to assess the quality of the configurations might have an effect on the tuning results.

**irace** provides two kinds of statistical tests. Both test have different characteristics that could be beneficial for certain scenarios:

- **Friedman test (F-test)**: This test uses the ranking of the configurations to analyze the difference between their performance. This makes the test suitable for scenarios where the numerical results and their scale are not significant to assess the quality of the configurations. For example, if the results for different instances have high numerical differences and evaluating the performance of the configurations using the mean could be deceiving. We recommend to use the **F-test** (default) when tuning for solution quality and whenever the best performing algorithm must solve as well as possible all the instances.

- Student's t-test (**t-test**): This test uses the mean performance of the configurations to analyze the differences between the configurations. This makes the test suitable for scenarios where the differences between values obtained for different instances are relevant to assess good configurations. We recommend using t-test when tuning for computation time, whenever the obtained configurations must solve the instances in the best average time.

Using the option **confidence**, is possible to set the statistical significance of the test. Increasing the value of **confidence** leads to a more strict statistical test. Keep in mind that a strict test will require more budget to identify which configurations perform worse. A less strict test discards configurations quickly by requiring less data against them and therefore it has more probability of discarding good configurations.

## 10.6 Complex parameters

Some parameters may have complex dependencies. We advice to always try to define the parameters in the way that is most suitable for the tuning objective. For example, when tuning a branch and bound algorithm one may have the following parameters:

- branching (**b**): This parameter can take the values  $\{0,1,2,3\}$ , 0 indicates no branching will be used and the rest are different types of branching.
- stabilization (**s**): This parameter can take the values  $\{0,1,2,3,4,5,6,7,8,9,10\}$ , of which for  $b=0$  only  $\{0,1,2,3,4,5\}$  are relevant.

In this case is not possible to describe the parameter space defining only two parameters for **irace**. An extra parameter must be introduced as follows:

```
# <name> <label> <type> <range> [ | <condition>]
b      "-b "  c      (0,1,2,3)
s1     "-s "  c      (0,1,2,3,4,5) | b == "0"
s2     "-s "  c      (0,1,2,3,4,5,6,7,8,9,10) | b != "0"
```

Parameters whose values depend on the value of other parameters also could be described using extra parameters or changing the parameters and processing them in the **targetRunner**. For example the following parameters:

- Population size (**p**): This parameter can take the integer values  $[1,100]$ .
- Selection size (**s**): This parameter can take as maximum the population size, that is  $[1,p]$ .

In this case is possible to describe the parameters  $p$  and  $s$  using surrogate parameters that represent a percentage of original interval as described below.

```
# <name> <label> <type> <range>
p      "-p "  r      (0.0,1.0)
s      "-s "  r      (0.0,1.0)
```

The parameter values must be processed in the **targetRunner**. For example, if the surrogate  $p$  parameter is  $p=0.5$ , transforming  $p$  to the interval  $[1,100]$  we obtain that the final value is  $p=50$ . Note that in this case more than one value of the surrogate parameter (e.g.,  $p=0.501, p=0.502$ ) can result in the same final value. Parameter  $s$  has an interval that depends on the final value



of parameter  $p$ , the percentage  $s=0.3$  must be calculated over the  $[1,50]$  interval, giving that the final value for  $s$  is  $s=15$ .

More complex value dependencies could be also expressed by mixing extra parameters and transformations. Keep in mind that the `targetRunner` can also process the parameters. You can also split parameters and join them in the `targetRunner`, for example assume the following parameters:

```
# <name> <label> <type> <range>
m      "-m "   i      (1,250)
e      "-e "   r      (0.0,2.0)
```

These parameters could be part of one parameter that has a multiplier and an exponent that has to be passed to your target algorithm as `--strength  $m \cdot 10^e$` . `targetRunner` can join the extra parameters `e` and `m` and provide them in the correct format.

## 10.7 Unreliable target algorithms

There are some situations in which the target algorithm may fail to execute correctly. This could be due to system problems or bugs for which no fix is available or fixing them is impossible because there is no access to the source code.

The `irace` option `targetRunnerRetries` indicates the number of times a `targetRunner` execution is repeated if it fails. Use this option if you know new repetitions could be successful.

When the program consistently fails using a particular set of configurations and repeating the execution will cause always the program to crash, you can use the `forbiddenFile` option to specify the configurations that must be avoided. On the other hand, if you do not know which configurations cause the problems, we advise you to handle this in the `targetRunner` script, when the program crashes you can use a penalty evaluation (very big number for minimization) that will allow `irace` to discard the configuration based on that result. Adjust the penalty according to your objective and the results you consider appropriate, for example, if a configuration crashes for an instance you might still consider it as a good configuration if it gives very good results for other instances.

## 11 irace options

Most of the `irace` options can be specified by command line using a flag, by setting them in the `irace` scenario file using the option name or by directly setting them in the scenario R object. This section describes the `irace` options that can be specified by the user.

### 11.1 General options

`scenarioFile` flag: `-s` or `--scenario` default: `./scenario.txt`

File that contains the scenario setup and other `irace` settings. All options listed in this section can be included in this file. See `$IRACE_HOME/templates/` for an example.

`debugLevel` flag: `--debug-level` default: 0

Level of information to display in the text output of `irace`. A value of 0 silences all debug messages. Higher values provide more verbose debug messages. To see details about the text output of `irace`, see [Section 9.1](#).

`seed` flag: `--seed` default: NA

Seed to initialize the random number generator. The seed must be a positive integer. If the seed is NA a random seed will be used.

`execDir` flag: `--exec-dir` default: ./

Directory where the target algorithm executions will be performed. The default `execDir` is the current directory.



`irace` will not attempt to create the execution directory so it must exist before calling `irace`.

`logFile` flag: `-l` or `--log-file` default: `./irace.Rdata`

File to save tuning results as an R dataset. The provided path must be either an absolute path or a relative to `execDir`. See [Section 9.2](#) for details on the format of the R dataset.

## 11.2 Elitist `irace`

`elitist` flag: `--elitist` default: 1

Enable/disable elitist `irace`.

In the **elitist `irace` version** elite configurations cannot be discarded from the race until the new configurations have executed the same instances as the elite configurations.

The race begins with a number of initial instances for which any configuration in race have been executed. This number of instances can be defined with the option `elitistInstances`. Once the new instances have been executed, the instances executed in previous iterations are executed. Elite configurations have already results for most of these instances and therefore do not need to be executed. Finally when the “previous instances” have all finished to be executed, new instances are used.

The statistical tests can be performed at any moment during the race according to the setting of the options `firstTest` and `eachTest`. The elitist rule forbids to discard elite configurations, even if the show bad performance, until the last “previous instance” has been executed.

The **non-elitist `irace` version** can discard the elite configurations from the race at any time. Instances are not re-used from one iteration to another, but new instances are always executed unless the `deterministic` option is active and all instances have already been used.

`elitistInstances` flag: `--elitist-instances` default: 1

Number of new instances to add to the execution list before “previous instances” in elitist `irace`.



If `deterministic` is TRUE then the number of `elitistInstances` will be reduced or set to 0 in case no more instances are available.

`elitistLimit` flag: `--elitist-limit` default: 2

Limit for the elitist race that specifies the number of statistical tests performed without successful elimination. If it reaches `elitistLimit`, the race will be stopped. This limit has effect after all “previous instances” have been executed. Use 0 to disable the limit.

### 11.3 Internal irace options

**sampleInstances** flag: `--sample-instances` default: 1

Enable/disable the sampling of the training instances. If the option `sampleInstances` is disabled, the instances are used in the order provided in the `trainInstancesFile` or in the order they are read from the `trainInstancesDir` when `trainInstancesFile` is not provided. For more information about training instances see [Section 5.4](#).

**nbIterations** flag: `--iterations` default: 0

Number of iterations to be executed. By default **irace** calculates the number of iterations based on the scenario as described as follows ( $N^{\text{param}}$  is the number of non fixed parameters to be tuned).

$$N^{\text{iter}} = \lfloor 2 + \log_2 N^{\text{param}} \rfloor \quad (1)$$

We recommend to use the default value.

**nbExperimentsPerIteration** flag: `--experiments-per-iteration` default: 0

Number of experiments to execute per iteration. By default **irace** calculates the number of experiments per iteration based on the scenario as follows, where  $B_j$  is the budget for iteration  $j$ ,  $B$  is the total tuning budget (`maxExperiments`),  $B_{\text{used}}$  is the used budget and  $N^{\text{iter}}$  is maximum between the planned number of iterations (`nbIterations`) and the current iteration ( $j$ ).

$$B_j = \frac{(B - B_{\text{used}})}{(N^{\text{iter}} - j + 1)} \quad (2)$$

We recommend to use the default value.

**nbConfigurations** flag: `--num-configurations` default: 0

The number of configurations that should be sampled and evaluated at each iteration. By default **irace** calculates the number of configurations per iteration based on the scenario as follows, where  $N_j$  is the number of configurations that will be used in iteration  $j$ ,  $B_j$  is the budget for iteration  $j$  and  $\mu$  is the **irace** option `mu`. We recommend to use the default value.

$$N_j = \lfloor \frac{B_j}{(\mu + \min(5, j))} \rfloor \quad (3)$$

**mu** flag: `--mu` default: 5

This value is used to determine the number of configurations to be sampled and evaluated at each iteration.

**minNbSurvival** flag: `--min-survival` default: 0

The minimum number of configurations needed to continue the execution of an iteration.

**softRestart** flag: `--soft-restart` default: 1

Enable/disable the soft restart strategy that avoids premature convergence of the probabilistic model. When a sampled configuration is *highly similar* to its parent configuration the probabilistic model these configurations is soft restarted. The similarity of categorical and ordered parameters is given by the hamming distance, the option `softRestartThreshold` defines the similarity of numerical parameters.

**softRestartThreshold** flag: `--soft-restart-threshold` default: NA

Soft restart threshold value for numerical parameters. If NA, it is computed as  $10^{-\text{digits}}$ , where `digits` corresponds to the **irace** option explained in this section.

## 11.4 Target algorithm parameters

- parameterFile** flag: `-p` or `--param-file` default: `./parameters.txt`  
File that contains the description of the parameters of the target algorithm. See [Section 5.1](#).
- digits** flag: `--digits` default: 4  
Number of decimal places to be considered for the real parameters.
- forbiddenFile** flag: `--forbidden-file` default:  
File containing a list of logical expressions that cannot be true for any evaluated configuration. If empty or NULL, no forbidden configurations are considered. See [Section 5.6](#) for more information.

## 11.5 Target algorithm execution

- targetRunner** flag: `--target-runner` default: `./target-runner`  
This option defines a script or an R function that launches the program to be tuned for a particular experiment (configuration + instance). See [Section 5.2](#) for details.
- targetRunnerRetries** flag: `--target-runner-retries` default: 0  
Number of times to retry a call to `targetRunner` if the call failed.
- targetRunnerData** default: NULL  
Optional data passed to `targetRunner`. This is ignored by the default `targetRunner` function, but it may be used by custom `targetRunner` functions to pass persistent data around.
- targetRunnerParallel** default: NULL  
Optional R function to provide custom parallelization of `targetRunner`. See [Section 6](#) for more information.
- targetEvaluator** flag: `--target-evaluator` default: ""  
Optional script or R function that evaluates an experiment (configuration + instance), that is. The evaluation must consist of a numeric value. See [Section 5.3](#) for details.
- deterministic** flag: `--deterministic` default: 0  
Enable/disable deterministic algorithm mode. If the target algorithm is deterministic, configurations will be evaluated only once per instance. See [Section 5.4](#) for more information.



Note that if the number of instances provided is less than the value specified for the option `firstTest`, no statistical test will be performed.

- parallel** flag: `--parallel` default: 0  
Number of calls of the `targetRunner` to execute in parallel. A value of 0 means disabled. For more information on parallelization see [Section 6](#).
- loadBalancing** flag: `--load-balancing` default: 1  
Enable/disable load-balancing when executing experiments in parallel. Load-balancing makes better use of computing resources, but increases communication overhead. If this overhead is large, disabling load-balancing may be faster. See [Section 6](#).

**mpi** flag: `--mpi` default: 0  
Enable/disable MPI. Use **Rmpi** to execute the **targetRunner** in parallel. When **mpi** is enabled, the option **parallel** is the number of slaves. See [Section 6](#).

**sgeCluster** flag: `--sge-cluster` default: 0  
Enable/disable SGE cluster mode. Use `qstat` to wait for cluster jobs to finish (**targetRunner** must invoke `qsub`). See [Section 6](#).

## 11.6 Initial configurations

**configurationsFile** flag: `--configurations-file` default:  
File containing a list of initial configurations. If empty or NULL, **irace** will not use initial configurations. See [Section 5.5](#).



The provided configurations must not violate the constraints described in **parameterFile** and **forbiddenFile**.

## 11.7 Training instances

**trainInstancesDir** flag: `--train-instances-dir` default: `./Instances`  
Directory where tuning instances are located; either absolute path or relative to current directory. See [Section 5.4](#).

**trainInstancesFile** flag: `--train-instances-file` default:  
File containing a list of instances and optionally additional parameters for them. See [Section 5.4](#).



If **trainInstancesDir** is specified the path contained in **trainInstancesFile** must be relative to the directory. For having the absolute path or for defining instances that are not files set `trainInstancesDir=""`.

## 11.8 Tuning budget

**maxExperiments** flag: `--max-experiments` default: 0  
The maximum number of runs (invocations of **targetRunner**) that will be performed. It determines the maximum budget of experiments for the tuning. See [Section 10.1](#).

**maxTime** flag: `--max-time` default: 0  
The maximum time in seconds for the runs of **targetRunner** that will be performed. The mean execution time is estimated to calculate the maximum number of experiments. When this is used **targetRunner** must return the execution time as second output: `< cost > < time >`. See [Section 10.1](#).

**budgetEstimation** flag: `--budget-estimation` default: 0.02  
The percentage of the budget used for estimating the mean execution time. Only used when **maxTime** > 0. See [Section 10.1](#).

## 11.9 Statistical test

`testType` flag: `--test-type` default: `F-test`  
Specifies the statistical test type:

- `F-test` (Friedman test)
- `t-test` (pairwise t-tests with no correction)
- `t-test-bonferroni` (t-test with Bonferroni's correction for multiple comparisons)
- `t-test-holm` (t-test with Holm's correction for multiple comparisons).

When selecting `t-test` we recommend to not use corrections for multiple comparisons. See [Section 10.5](#) to have more information about how to choose the statistical test.

`firstTest` flag: `--first-test` default: 5  
Specifies how many instances are executed before the first elimination test.



The value of `firstTest` must be a multiple of `eachTest`.

`eachTest` flag: `--each-test` default: 1  
Specifies how many instances are executed between elimination tests.

`confidence` flag: `--confidence` default: 0.95  
Confidence level for the elimination test.

## 11.10 Recovery

`recoveryFile` flag: `--recovery-file` default: ""  
Previously saved `irace` log file that should be used to recover the execution of `irace`, either absolute path or relative to the current directory. If empty or NULL, recovery is not performed. For more details about recovery see [Section 11.10](#).

## 11.11 Testing

`testNbElites` flag: `--test-num-elites` default: 1  
Number of elite configurations returned by `irace` that will be tested if test instances are provided. For more information about the testing, see [Section 7](#).

`testIterationElites` flag: `--test-iteration-elites` default: 0  
Enable/disable testing the elite configurations found at each iteration.

`testInstancesDir` flag: `--test-instance-dir` default:  
Directory where testing instances are located, either absolute or relative to the current directory.

`testInstancesFile` flag: `--test-instance-file` default:  
File containing a list of test instances and optionally additional parameters for them.

## 12 FAQ

### 12.1 Is irace minimizing or maximizing the output of my algorithm?

By default, **irace** considers that the value returned by `targetRunner` (or by `targetEvaluator`, if used) should be *minimized*. In case of a maximization problem, one can simply multiply the value by -1 before returning it to irace. This is done, for example, when maximizing the hypervolume (see the last lines in `$IRACE_HOME/examples/hypervolume/target-evaluator`).

### 12.2 Is it possible to configure a MATLAB algorithm with irace?

Definitely. There are two main ways to achieve this:

1. Edit the `targetRunner` script to call MATLAB in a non-interactive way. See the MATLAB documentation, or the following links<sup>3</sup>. You would need to pass the parameter received by `targetRunner` to your MATLAB script: <http://www.mathworks.nl/support/solutions/en/data/1-1BS5S/?solution=1-1BS5S>. There is a minimal example in:

```
$IRACE_HOME/examples/matlab/.
```

2. Call MATLAB code directly from R using the **R.matlab** package (<http://cran.r-project.org/package=R.matlab>). This is a better option if you are experienced in R. Define `targetRunner` as an R function instead of a path to a script. The function should call your MATLAB code with appropriate parameters.

### 12.3 My program works perfectly on its own, but not when running under irace. Is irace broken? Is irace broken?

Every time this was reported, it was a difficult-to-reproduce bug in the program, not in **irace**. We recommend that in `targetRunner`, you use `valgrind` to run your program. That is, if your program is called like:

```
$EXE ${FIXED_PARAMS} -i $INSTANCE ${CONFIG_PARAMS} \  
1> ${STDOUT} 2> ${STDERR}
```

then replace that line with:

```
valgrind --error-exitcode=1 $EXE ${FIXED_PARAMS} \  
-i $INSTANCE ${CONFIG_PARAMS} 1> ${STDOUT} 2> ${STDERR}
```

If there are bugs in your program, they will appear in `$STDERR`, thus do not delete those files.

### 12.4 My program may be buggy and run into an infinite loop. Is it possible to set a maximum timeout?

We are not aware of any way to achieve this using R. However, in GNU/Linux, it is easy to implement by using the `timeout` command in `targetRunner` when invoking your program.

<sup>3</sup><http://stackoverflow.com/questions/1518072/suppress-start-message-of-matlab>  
<http://stackoverflow.com/questions/4611195/how-to-call-matlab-from-command-line-and-print-to-stdout-before-exiting>

## 12.5 When using the `mpi` option, `irace` is aborted with an error message indicating that a function is not defined. How to fix this?

`Rmpi` does not work the same way when called from within a package and when called from a script or interactively. When `irace` creates the slave nodes, the slaves will load a copy of `irace` automatically. If the slave nodes are on different machines, they must have `irace` installed. If `irace` is not installed system-wide, R needs to be able to find `irace` on the slave nodes. This is usually done by setting `R_LIBS`, `.libPaths()` or by loading `irace` using `library()` or `require()` with the argument “`lib.loc`”. The settings on the master are not applied to the slave nodes automatically, thus the slave nodes may need their own settings. After spawning the slaves, it is too late to modify those settings, thus modifying the shell variable `R_LIBS` seems the only valid way to tell the slaves where to find `irace`.

If the path is set correctly and the problem persists, please check follow these instructions:

1. Test that `irace` and `Rmpi` work: run `irace` on a single machine (submit node), without calling `qsub`, `mpirun` or a similar wrapper around `irace/R`.
2. Test loading `irace` on the slave nodes. Note that `qsub/mpirun` may use a different mechanism than if you log into the node (e.g., with `ssh`). Thus, you need to write a little R program such as:

```
library(Rmpi)
mpi.spawn.Rslaves(nslaves = 10)
x <- mpi.applyLB(1:10, function(x) {
  library(irace)
  return(path.package("irace")) })
print(x)
```

submit this program to the cluster (using `qsub/mpirun`) like you would submit `irace`.

3. Check `bin/parallel-irace-mpi`. The function `irace_main()` creates an MPI job for our cluster. You may need to speak with the admin of your cluster and ask them how to best submit a job for MPI. There may be some particular settings that you need. Note also that `Rmpi` usually creates some log files; `irace` does not create those files unless `debugLevel > 0`.

Please contact us on the `irace` google group if you have further problems.

## 13 Resources and contact information

More information of the package can be found on the `irace` webpage:

<http://iridia.ulb.ac.be/irace/>.

For questions and suggestions please contact the development team through the `irace` package Google group:

<https://groups.google.com/d/forum/irace-package>

or by sending an email to:

[irace-package@googlegroups.com](mailto:irace-package@googlegroups.com)



## 14 Acknowledgements

We would like to thank all the people that directly or indirectly have collaborated in the development and improvement of **irace**.

- Prasanna Balaprakash
- Zhi (Eric) Yuan
- Franco Mascia
- Alberto Franzin
- Anthony Antoun

## Appendix A R installation

This section gives a quick R installation guide that will work in most cases. The official instructions are available at <http://cran.r-project.org/doc/manuals/r-release/R-admin.html>

### A.1 GNU/Linux

You should install R from your package manager. On a Debian/Ubuntu system it will be something like:

```
sudo apt-get install r-base
```

Once R is installed, you can launch R from the Terminal and from the R prompt install the **irace** package (see [Section 3.2](#)).

### A.2 OS X

You can install R directly from a CRAN mirror<sup>4</sup>. Alternatively, if you use homebrew, you can just brew the R formula from the science tap (unfortunately it does not come already bottled so you need to have Xcode<sup>5</sup> installed to compile it):

```
brew tap homebrew/science
brew install r
```

Once R is installed, you can launch R from the Terminal (or from your Applications), and from the R prompt install the **irace** package (see [Section 3.2](#)).

### A.3 Windows

You can install R from a CRAN mirror<sup>6</sup>. We recommend that you install R on a filesystem path without spaces, special characters or long names, such as `C:\R`. Once R is installed, you can launch the R console and install the **irace** package from it (see [Section 3.2](#)).

## Appendix B TargetRunner script check list

When the **targetRunner** script is not running properly it can be difficult to detect where the problem is. The more your script provides descriptive errors, the easier it will be to debug it. If you are using temporary files to redirect the output of your algorithm, check that these are created properly. We recommend to follow the structure of the example file (**target-runner**) provided in `$IRACE_HOME/templates`. The following examples are based on a file with that characteristics.

When you have problems with the **targetRunner**, you will see an error on the **irace** output that says that the execution of **targetRunner** was not successful.

Follow this list to detect where the problem is:

---

<sup>4</sup>Belgian CRAN mirror: <http://cran.freestatistics.org/bin/macosx/>

<sup>5</sup>Xcode download webpage: <https://developer.apple.com/xcode/download/>

<sup>6</sup>Belgian CRAN mirror: <http://cran.freestatistics.org/bin/windows/>

1. Make sure that your `targetRunner` script is the specified location. If you see an error as:

```
Error: == irace == run program runner '~/tuning/target-runner' does not exist
```

`irace` is not finding the script file. Check that the file is in the path specified by the error.

2. Make sure that your `targetRunner` script is an executable if you see an error as:

```
Error: == irace == run program runner '~/tuning/target-runner' is a directory, not a file
```

or

```
Error: == irace == run program runner '~/tuning/target-runner' is not executable
```

This error means that your `targetRunner` is not an executable file. In the first case, the script is a folder and therefore there must be a problem with the name of the script. In the second case, you must make the file executable, which in GNU/Linux can be done by:

```
cd ~/tuning/  
chmod +x target-runner
```

3. Make sure that your executable is in the location described in the script (variable `EXE` for the templates example). If you see an error as follows this is your problem:

```
Error: == irace == running command '~/tuning/target-runner'  
1 8 676651103 ~/tuning/Instances/1000-16.tsp --ras  
--localsearch 2 --alpha 4.03 --beta 1.89 --rho 0.02 --ants 37  
--nnls 48 --dlb 0 --rasranks 15 2>&1' had status 1
```

```
== irace == The call to target.runner.default was:  
~/tuning/target-runner 1 8 676651103 ~/tuning/Instances/1000-16.tsp  
--ras --localsearch 2 --alpha 4.03 --beta 1.89 --rho 0.02  
--ants 37 --nnls 48 --dlb 0 --rasranks 15
```

```
== irace == The output was:  
Tue May 3 19:00:37 UTC 2016: error: ~/bin/acotsp: not found  
or not executable (pwd: ~/tuning/acotsp-arena)
```

For testing your script you can copy the line of execution and execute it directly in the command-line. In this case, the line is:

```
~/tuning/target-runner 1 8 676651103 ~/tuning/Instances/1000-16.tsp --ras --localsearch  
2 --alpha 4.03 --beta 1.89 --rho 0.02 --ants 37 --nnls 48 --dlb 0 --rasranks 15
```

This line executes the `targetRunner` script as `irace` does. The output of this script must be only one number.

4. Check that your `targetRunner` script is actually returning one number as output. If you see an error as the following, this is your problem:

```
Error: == irace == The output of '~/tuning/target-runner
1 25 365157769 ~/tuning/Instances/1000-31.tsp --ras
--localsearch 1 --alpha 0.26 --beta 6.95 --rho 0.69
--ants 56 --nnls 10 --dlb 0 --rasranks 7' is not numeric!

== irace == The output was:
Solution: 24479793
```

For testing your script, copy the line of execution and execute it directly in the command-line:

```
~/tuning/target-runner 1 25 365157769 ~/tuning/Instances/1000-31.tsp --ras --localsearch
1 --alpha 0.26 --beta 6.95 --rho 0.69 --ants 56 --nnls 10 --dlb 0 --rasranks 7
```

This line executes the `targetRunner` script as `irace` does. The output of this script must be only one number. In this example, the output of the script is “Solution: 24479793”, which means that the regular expression used to obtain the result from the algorithm output file must be checked.

5. Check that your `targetRunner` script is creating the output files for your algorithm. If you see an error as:

```
== irace == The output was: Tue May 3 19:41:40 UTC 2016:
error: c1-9.stdout: No such file or directory
```

The output file of the execution of your algorithm has not been created (check permissions) or has been deleted before the result can be read.

6. Other errors can produce the following output:

```
== irace == The output was: Tue May 3 19:49:06 UTC 2016:
error: c1-23.stdout: Output is not a number
```

This might be due that your `targetRunner` script is not executing your algorithm correctly. To further investigate this issue comment the line that eliminates the temporary files where the output of your algorithm is redirected:

```
rm -f "${STDOUT}" "${STDERR}"
```

Execute the `targetRunner` command-line the error provides and search in your execution directory the files that are created. Check the `.stderr` file for errors and the `.stdout` file to see the output your algorithm produces.

## Appendix C Glossary

1. Parameter tuning: Process of searching good settings for the parameters of an algorithm under a particular tuning scenario (instances, execution time, etc.).
2. Scenario: Settings of a tuning scenario, these settings include the algorithm to be tuned (target), budget for the execution of the target algorithm (execution time, evaluations, iterations, etc.), set of problem instances and all the information that is required to perform the tuning.
3. Target algorithm: algorithm whose parameters will be tuned.
4. Target parameter: parameter of the target algorithm that will be tuned.
5. **irace** option: configurable option of **irace**.
6. Elite configurations: best configurations found from whose probabilistic models new configurations are sampled for the next iteration. All elite configurations are also included in the next iteration.