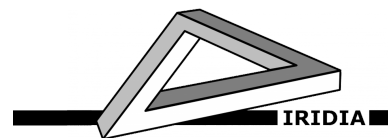




ECOLE
POLYTECHNIQUE
DE BRUXELLES



The crace Package: User Guide

Yunshuang Xiao, Leslie Pérez Cáceres,
Manuel López-Ibáñez and Thomas Stützle

Version 1.4, Last updated: September 14, 2026

IRIDIA - Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle
CoDE, Université Libre de Bruxelles, Brussels, Belgium

Contents

1	General information	5
1.1	Background	5
1.2	Version	5
1.3	License	5
2	Before starting	5
3	Installation	6
3.1	System requirements	6
3.2	crace installation	7
3.2.1	Install automatically with conda	7
3.2.2	Install automatically with pip	7
3.2.3	Manual download and installation from source	7
3.2.4	Testing the installation and invoking crace	8
3.2.5	Checking the installation path of crace	8
4	Running crace	9
4.1	Step-by-step setup guide	9
4.2	Setup example for ACOTSP	13
5	Defining a configuration scenario	14
5.1	Target algorithm parameters	14
5.1.1	Parameter types	14
5.1.2	Parameter domains	15
5.1.3	Conditional parameters	15
5.1.4	Forbidden parameter configurations	15
5.1.5	Parameter file format	16
5.1.6	Load parameters	17
5.1.7	Parameters python format	18
5.2	Target algorithm runner	19
5.2.1	Target runner as an executable program	19
5.2.2	targetRunnerLauncher as an executable command line	20
5.2.3	targetRunnerRetries as the times to repeat targetRunner	20
5.2.4	Load targetRunner	20
5.3	Training instances	21
5.3.1	Load training instances	22
5.3.2	Instances python format	22
5.4	Initial configurations	24
5.4.1	Load initial configurations	24
5.4.2	Configurations python format	24
6	Crace options	25
6.1	Crace options python format	26
6.2	Tuning budget	27
6.3	Heterogeneous scenarios	27
6.4	Choosing the elimination test	28
6.4.1	Statistical test	28
6.4.2	Aggressive test	29
6.5	Tuning for minimizing computation time	29

<i>CONTENTS</i>	3
7 Parallelization	31
8 Testing (Validation) of configurations	32
9 Recovering crace runs	33
10 Output and results	34
10.1 Text output	34
10.2 Log files	37
10.2.1 Experiments python format	38
10.2.2 Models python format	38
10.2.3 Crace results python format	39
10.3 Simple analysis of results	46
11 List of command-line and scenario options	47
11.1 Utility options	47
11.2 General options	48
11.3 Scenario configurations	49
11.4 Instances options	49
11.5 Recovery options	50
11.6 Budget options	50
11.7 Elimination Options	50
11.8 Parallelization options	51
11.9 Internal crace options	51
11.10 Elitist options	52
11.11 Capping options	52
12 FAQ (Frequently Asked Questions)	53
12.1 Is crace minimizing or maximizing the output of my algorithm?	53
12.2 Are experiments with crace reproducible?	53
12.3 My program works perfectly on its own, but not when running under crace . Is crace broken?	54
12.4 crace seems to run forever without any progress, is this a bug?	54
12.5 My program may be buggy and run into an infinite loop. Is it possible to set a maximum timeout?	54
12.6 When using the mpi option, crace is aborted with an error message indicating that a function is not defined. How to fix this?	54
12.7 How are relative filesystem paths interpreted by crace ?	55
12.8 My parameter space is small enough that crace could generate all possible configurations; however, crace generates repeated configurations and/or does not generate some of them. Is this a bug?	56
12.9 On Windows and using target-runner.py (a Python file), I get the error target- runner.py is not executable	56
13 Resources and contact information	56
14 Acknowledgements	57
Bibliography	58

Appendix A	Installing Python	59
A.1	GNU/Linux	59
A.2	macOS	59
A.3	Windows	59
Appendix B	Using a virtual environment	60
Appendix C	Attachments	62
Appendix D	Changelog	66

1 General information

1.1 Background

The **crace** package implements a *continuously racing* procedure, which is an alternative to **irace** that performs in each iteration a single race. The continuously racing configurator (**crace**) evaluates, removes and generates new configurations asynchronously, granting a high level of flexibility regarding the configuration process when compared to the previous iterative scheme. The main use of **crace** is the automatic configuration of decision and optimization 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 **irace** package by López-Ibáñez and it is implemented in Python.

<https://pypi.org/project/crace/>

More information about **crace** is available at <https://race-autoconfig.github.io/crace>.

1.2 Version

This current major version of the **crace** package is 1.4. Previous versions of the package can also be found in the [PyPI website](#).

1.3 License

The **crace** package is Copyright © 2026 and distributed under the GNU General Public License version 3.0 (<http://www.gnu.org/licenses/gpl-3.0.en.html>).

This program 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.

This program 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.

Please be aware that the fact that this program is released as Free Software does not excuse you from scientific propriety, which obligates you to give appropriate credit! If you write a scientific paper describing research that made substantive use of this program, it is your obligation as a scientist to (a) mention the fashion in which this software was used in the Methods section; (b) mention the algorithm in the References section. The appropriate citation is:

Yunshuang Xiao, Leslie Pérez Cáceres, Manuel López-Ibáñez, and Thomas Stützle. 2023. Algorithm Configuration via Continuously Racing: Preliminary Results. In Proceedings of the Companion Conference on Genetic and Evolutionary Computation (GECCO '23 Companion). Association for Computing Machinery, New York, NY, USA, 1744–1752. [10.1145/3583133.3596408](https://doi.org/10.1145/3583133.3596408)

2 Before starting

The **crace** package provides an automatic configuration tool for tuning optimization algorithms, that is, automatically finding 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 **crace** works. Similar to **irace**, **crace** receives as input a *parameter space definition* corresponding to the parameters of the target algorithm that will

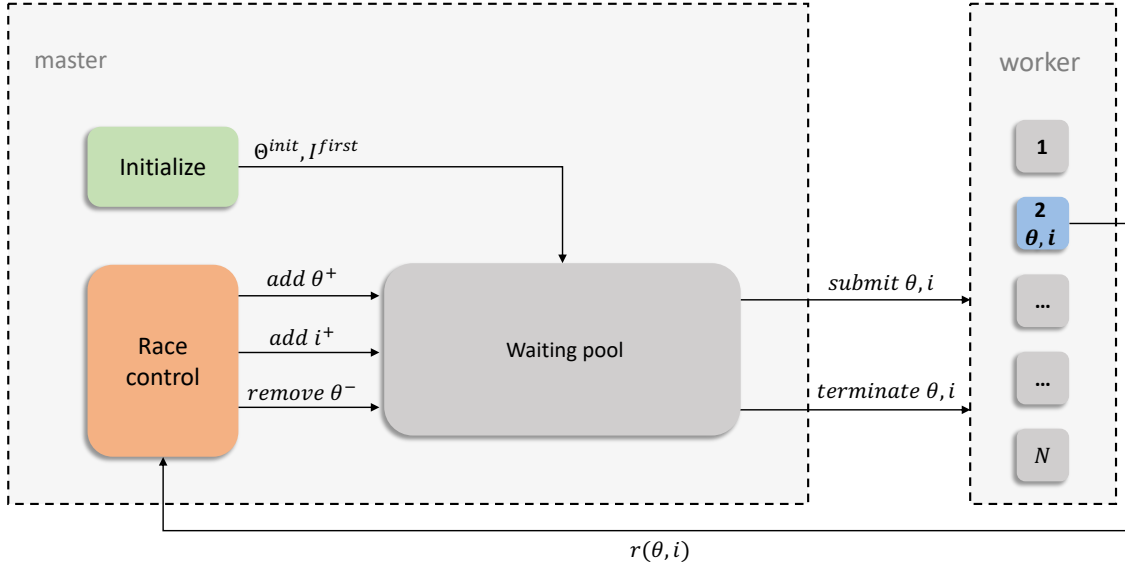


Figure 1: Scheme of **crace** flow of information.

be tuned, a set of *instances* for which the parameters must be tuned for and a set of options for **crace** that define the *configuration scenario*. Then, **crace** searches in the parameter space for well performing algorithm configurations by executing the target algorithm on different instances with different parameter configurations. A **targetRunner** must be provided to execute the target algorithm with a specific parameter configuration (θ) and instance (i). The **targetRunner** function (or program) acts as an interface between the execution of the target algorithm and **crace**: It receives the instance and configuration as arguments and must return the evaluation of the execution of the target algorithm. The major difference between **crace** and **irace** is that an asynchronous type of executing is implemented in **crace**. Although **irace** can do a parallel execution of the experiments, this parallel execution is limited to do a synchronous type of execution on the same instance. Hence, if at any time in the procedure of **irace** the parallel cores are more than the parallel executions that have to be done, then these cores which do not executed anything will be free. This waste of computational resources is large when a group of experiments that are parallel executed on a same instance vary a lot from the computation time consumption. Therefore, it is expected that the **crace** will be doing potentially better with many more cores than **irace**.

The following user guide contains guidelines for installing **crace**, defining configuration scenarios, and using **crace** to automatically configure your algorithms.

3 Installation

3.1 System requirements

- Python (version $\geq 3.6.0$) is required for running **crace**, but you don't need to know the Python language to use it. Python is freely available and you can download it from the Python project website (<https://www.python.org/>). See [Appendix A](#) for a quick installation guide of Python.
- For GNU/Linux and macOS, the executable command `crace parallel` requires `bash`. Individual examples may require additional software.



Note: This package is currently tested with Python versions up to 3.14.3.

3.2 **crace** installation

The only prerequisite for installing **crace** is Python itself. If you don't have Python yet, see [Appendix A](#) for a quick installation guide of Python. The **crace** package can be installed automatically with conda, with pip or from the source. We advise to use the automatic installation unless particular circumstances do not allow it. The instructions to install **crace** with the three mentioned methods are the following:

3.2.1 Install automatically with conda

If you use conda, execute the following line at the shell to install the package:

```
# Best practice, use an environment rather than install in the base env
conda create -n my-env
conda activate my-env
# If you want to install from conda-forge
conda config --env --add channels conda-forge
# The actual install command
conda install crace
```

Alternatively, within the conda graphical interface (Anaconda Navigator), you may search and install **crace**.

3.2.2 Install automatically with pip

If you use pip, execute the following line at the shell to install the package:

```
pip3 install crace
```

Also when using pip, it's good practice to use a virtual environment and here is the [guide](#)¹ for using the virtual environment. See [Appendix A](#) for a quick installation guide of Python.

3.2.3 Manual download and installation from source

If the previous installation instructions fail because of insufficient permissions and you do not have sufficient admin rights to install **crace** system-wide, then you need to force a local installation. From the **crace** package PyPI website (<https://pypi.org/>), search and download one of the three versions available depending on your operating system:

- `crace-1.4.tar.gz` (Unix/Linux/BSD/macOS)
- `crace-1.4-py3-none-any.whl` (python3, any platform)

Alternatively, you can use `wget` or `curl` from the command line to download the file:

- From PyPI website:

```
wget https://files.pythonhosted.org/packages/source/c/crace/crace-1.4.tar.gz
```

¹How to use virtual environment in pip, see <https://dev.to/bowmanjd/python-tools-for-managing-virtual-environments-3bko#howto>

- From github:

```
wget https://github.com/race-autoconfig/crace
```

To install the package on GNU/Linux, macOS or Windows using the formats .tar.gz, .tar or .zip, you should first extract the source files and then navigate to the directory containing the extracted files and run the following command to install the package:

```
pip3 install .
```

To install the package on any platform using the Wheel format, you should run the following command to install the package:

```
pip3 install /path/to/crace-1.4-py3-none-any.whl
```



Before installing **crace** using pip, it's better to upgrade packages **pip** and **setuptools** at first:

```
pip3 install --upgrade pip setuptools
```



Once **crace** is installed, executable command, **crace** will be added to the active environment. You can call **crace** to execute a single crace run, **crace parallel** to execute multiple crace runs and **crace doc** to check the user guide in Bash shell (Linux and MacOS) as well as in Powershell (Windows).

3.2.4 Testing the installation and invoking crace

Once **crace** has been installed, it can be listed by using command line at the shell or load the package and test that the installation was successful by opening a Python console and executing:

```
# List installed packages at the shell
pip3 list
# Show the information of crace at the shell
pip3 show crace
```

```
# Load and test at the Python console
import crace
print(crace.__version__)
```

3.2.5 Checking the installation path of crace

To check the installation path of **crace** using command line at the shell:

```
# Unix/macOS
which crace
```

```
# Windows
where crace
```

The output of this line should be \$PYTHON_HOME/bin/crace, which means the path before /bin/crace is \$PYTHON_HOME. Here, \$CRACE_HOME should be:

```
CRACE_HOME=$PYTHON_HOME/lib/python3.X/site-packages/crace
```

Also, you can check the installation path of **crace** opening a Python console and executing:

```
import importlib.util
print(importlib.util.find_spec('crace').submodule_search_locations)
```

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

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

```
# Replace <CRACE_HOME> with the crace installation path
export CRACE_HOME=<CRACE_HOME>
# Tell operating system where to find crace
export PATH=${PYTHON_HOME}/bin/:$PATH
```

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

```
crace --help
```

Alternatively, you may directly invoke **crace** within the Python console by executing:

```
import crace
crace.run("--help")
```

4 Running crace

Before performing the tuning of your algorithm, it is necessary to define a tuning scenario that will give **crace** 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 5.3](#))
4. **crace** options (see [Section 11](#)).
5. *Optional*: Initial configurations (see [Section 5.4](#)).

These scenario elements should be provided as plain text files. This user guide provides examples.

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 **crace** using the **ACOTSP** scenario go to [Section 4.2](#).

4.1 Step-by-step setup guide

This section provides a guide to setup a basic execution of **crace**. The template files provided in the package (`$CRACE_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 macOS, you can do this as follows:

```
mkdir ~/tuning
cd ~/tuning
```

2. Initialize the tuning directory with template config files. On GNU/Linux or macOS, you can do this as follows:

```
$PYTHON_HOME/bin/crace --init
# If $PYTHON_HOME/bin is in the system PATH
# This command line evaluates:
crace --init
```

Using this command line, a folder named ‘templates’ could be copied to the current directory. Two scenarios and some script files are provided in this folder.

3. Define the target algorithm parameters to be tuned by following the instructions for the file `parameters.txt`. Available parameter types and other guidelines can be found in [Section 5.1](#).
4. *Optional*: Define the initial parameter configuration(s) of your algorithm, which allows you to provide good starting configurations (if you know some) for the tuning. Follow the instructions in `configurations.txt` and set `configurationsFile="configurations.txt"` in `scenario.txt`. More information in [Section 5.4](#). If you do not need to define initial configurations remove this file from the directory.
5. 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 5.3](#) for guidelines.
6. Uncomment and assign in `scenario.txt` only the options for which you need a value different from the default. Some common options that you might want to adjust are:

execDir (`--exec-dir`): the directory in which **crace** will execute the target algorithm; the default value is the current directory. All log files are stored in directory `execDir/logDir`.

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

maxTime (`--max-time`): maximum total execution time in seconds for the executions of `targetRunner`. In this case, `targetRunner` must return two values: cost and time.



Note that you must provide at least one of `maxExperiments` or `maxTime`.

For setting the tuning budget see [Section 6.2](#). For more information on **crace** options and their default values, see [Section 11](#).

7. Modify the target-runner script to run your algorithm. This script must execute your algorithm with the parameters and instance specified by **crace** 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 target-runner template is written in GNU bash scripting language, which can be executed easily in GNU/Linux and macOS systems. However, you may use any other programming language. We provide examples written in Python, MATLAB and other languages in `$CRACE_HOME/examples/`. 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 **crace** 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}
```

- (d) Obtain the result returned by the target algorithm. 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 print **only** one number. The line:

```
if ! [[ "$COST" =~ [^+-0-9.e]+$ ]] ; then
```

checks if the final returned result is a number or not. In the template example, the result is printed with `echo "$COST"` (assuming `maxExperiments` is used) and the generated files are deleted by line:

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

or you may remove that line if you wish to keep the files.



The `target-runner` script must be an executable file. unless you specify `targetRunnerLauncher`.

You can test the target runner from the terminal by checking the scenario as explained in the next itemize.

If you have problems related to the `target-runner` script when executing **crace**, see [Section 12](#) for a check list to help diagnose common problems. For more information about the `targetRunner`, please see [Section 5.2](#),

8. The **crace** 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, **crace** 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:

- From the command-line:

```
crace --scenario /path/to/scenarioFile --check
```

- From the Python console:

```
import crace
# All arguments are provided in a list
crace.run(['--scenario', '/path/to/scenarioFile', '--check'])
```

Using this option, **crace** can test the target algorithm three times on one randomly selected training instance and one testing instance if provided, respectively. You can verify the scenario and parameter definitions work correctly or not based on the results.

9. Once all the scenario elements are prepared you can execute **crace**, either using the command-line wrappers provided by the package or directly from the Python console:

- **From the command-line console**, call the command:

```
cd ~/tuning/
# By default, crace reads scenario.txt in current directory,
# you can specify a different file with --scenario.
crace
```

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 **crace** options can be specified in the command line or directly in the `scenario.txt` file. See the output of `crace -h`, `crace --help` or `crace --man [option_name]` in the terminal for quick information on additional **crace** options.

- **From the Python console**, evaluate:

```
import crace
import os
# Go to the directory containing the scenario files
os.chdir("~/tuning")
# Check current directory
os.getcwd()

crace.run()
```

This will perform one run of **crace**. Besides, you can see the output of `crace.run('-h')`, `crace.run('--help')` or `crace.run(['--man', [option_name]])` in Python console for quick information on additional **crace** options.

- **From the Python console**, evaluate:

```
import crace
import os
# Go to the directory containing the scenario files
os.chdir("~/tuning")
# Check current directory
os.getcwd()

# scenario_file is the default enabled parameter
# and './scenario.txt' is its default value
crace.main()
```

For more information about **crace** options, see [Section 11](#).



Command-line options provided by `crace.run()` in Python console or `crace` in terminal will override the same option values specified in the `scenario.txt` file.

4.2 Setup example for ACOTSP

The **ACOTSP** tuning example can be found in the package installation in the folder `$CRACE_HOME/examples/acotsp`. Other example scenarios can be found in the same 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 to the next section. For this example, we assume a GNU/Linux system such as Ubuntu with a working C compiler such as gcc. To execute this scenario follow these steps:

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
# $CRACE_HOME is the installation directory of crace.
cp $CRACE_HOME/examples/acotsp/* ~/tuning/
# or directly call:
crace --init
```

2. Download the training instances from <https://iridia.ulb.ac.be/supp/IridiaSupp2016-003/index.html> 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 executing the experiments and execute **crace**:

```
mkdir ~/tuning/acotsp-arena/
cd ~/tuning/
# when activate the python environment including crace:
crace
```

6. Or you can also execute **crace** from the Python console using:

```
import crace
import os
os.chdir("~/tuning")
# Default: load scenario.txt at current directory
crace.run()

# Or call crace using:
crace.main()
```

5 Defining a configuration scenario

Among all available **crace** options, parameters (provided by option `parameterFile`), target algorithm (provided by option `targetRunner`) and training instances (provided by option `trainInstancesDir` or option `trainInstancesFile`) must be provided for the tuning.

Option `scenarioFile` is optional. By default, **crace** reads `scenario.txt` in the current directory. You can specify a different file with argument `--scenario` or initialize a `Scenario` object at Python console as following:

```
from crace import Scenario
from crace import main as crace_main
scenario = Scenario(options, parameters, instances)

# Call crace
crace_main(scenario=scenario)
```

In this way, you must initialize objects `CraceOptions` (Section 6.1), `Instances` (Section 5.3.2), `Parameters` (Section 5.1.7) and `Instances` (Section 5.3.2) in advance. Optionally, as well as object configurations (Section 5.4.2).



If there is no such a scenario file provided, **crace** will set option `scenarioFile` to `None` with a warning. Then **crace** will try to load other necessary options from arguments.

5.1 Target algorithm parameters

The parameters of the target algorithm are defined by a parameter file as described in Section 5.1.5. Optionally, when executing **crace** from the Python console, the parameters can be specified directly as a Python object (see Section 5.1.7). To define your parameters, you can 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 **crace** 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 **crace** are as 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 the option `digits`. For example, given the default number of digits of 4, the value 0.12345 is rounded to 0.1235 while 0.12341 is rounded to 0.1234. Selected real-valued parameters can be optionally sampled on a logarithmic scale (base *e*).
- *Integer* parameters are numerical parameters that can take only integer values within the 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 range of integer parameters is specified as the range of real parameters and they can also be optionally sampled on a logarithmic scale (base *e*).
- *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.



Boolean (or logical) parameters are best encoded as categorical ones with just two values rather than integer ones with domain (0, 1). Some boolean parameters take an explicit value (0/1 or true/false) such as:

```
d1b "--d1b " c (0, 1)
```

Others are switches whose presence activates the parameter:

```
d1b "" c ("", "--d1b")
```

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 `algorithm` that takes values (`as`, `mmas`, `eas`, `ras`, `acs`) and another parameter `q0` that only needs to be set if the first parameter takes precisely the value `acs`. Thus, parameter `q0` is conditional on `algorithm == "acs"`.



Non-active parameter(s) in initial configuration (Section 5.4) will be changed to None with a warning.

5.1.4 Forbidden parameter configurations

A list of forbidden expressions is optionally provided by a forbidden file. Its filesystem path is the value of option `forbiddenFile`. Each line in this file is a logical expression (in Python syntax) containing parameter names as defined by the `forbiddenFile`, values and logical operators. For a list of Python logical operators, see:

<https://docs.python.org/3/library/stdtypes.html#truth>

If **crace** generates a parameter configuration that makes any of the logical expressions evaluate to `TRUE`, then the configuration is considered to be forbidden and it is never evaluated. 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.



Initial configuration (Section 5.4) that are forbidden will be discarded with a warning.

If the forbidden constraints provided are too strict, **crace** may produce the following error:



ERROR: crace tried 100 times to sample from the model a configuration not forbidden without success, perhaps your constraints are too strict?

In that case, it may be a good idea to reformulate the forbidden constraints as conditional parameters (Section 5.1.3). As an example, Figure 2 shows the forbidden file of **ACOTSP** scenario.

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

Figure 2: Forbidden file (forbidden.txt) for tuning **ACOTSP**.

5.1.5 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> <domain> [ | <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 label when invoking `targetRunner`, thus *any whitespace in the label 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’. Numerical parameters can be sampled using a natural logarithmic scale with ‘i, log’ and ‘r, log’ (without spaces) for integer and real parameters, respectively.
- `<domain>` 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 be 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 follow the same syntax as those for specifying forbidden configurations (see below), that is, it must be a valid Python logical expression². The condition may contain the name of other parameters as long as the dependency graph does not contain any cycle. Otherwise, **crace** will detect the cycle and stop with an error.

²For a list of Python operators, see: <https://docs.python.org/3/library/stdtypes.html#truth>



Categorical and ordinal parameters are always treated as strings. Given a parameter like:

```
a " " c (0, 5, 10, 20)
```

then, a condition like `a > 10` will be true when `a` is 5. Because comparisons between strings are lexicographic and "10" is sorted before "5". As a work-around, it is suggested to use condition `a == '20'`.

As an example, [Figure 3](#) shows the parameters file of the **ACOTSP** scenario.

# name	switch	type	values	[conditions (using Python 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)	
nnls	--nnls "	i	(5, 50)	localsearch in "(1, 2, 3)"
dlb	--dlb "	c	(0, 1)	localsearch in "(1, 2, 3)"
q0	--q0 "	r	(0.0, 1.0)	algorithm == "acs"
rasrank	--rasranks "	i	(1, 100)	algorithm == "ras"
elitistants	--elitistants "	i	(1, 750)	algorithm == "eas"

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

5.1.6 Load parameters

To provide parameters to **crace**, you can:

1. Use option `parameterFile` and `forbiddenFile` (if necessary) in the `scenario.txt` as shown in [Figure 4](#).

```
## File that contains the description of the parameters of the target
## algorithm.
parameterFile = "/path/to/parameterFile"

## File that contains a list of logical expressions that cannot be TRUE for
## any evaluated configuration. If empty or NULL, do not use forbidden
## expressions.
#forbiddenFile = ""
```

Figure 4: Add parameters in scenario file.

2. Add option `parameterFile` and `forbiddenFile` (if necessary) when calling **crace** using arguments.

- Call **crace** at the shell as follows:

```
crace \
  --scenario /path/to/scenarioFile \
  --parameter-file /path/to/parameterFile
# If forbiddenFile is provided
crace \
  --scenario /path/to/scenarioFile \
  --parameter-file /path/to/parameterFile \
  --forbidden-file /path/to/forbiddenFile
```

- Call **crace** at the Python console as follows:

```
import crace
arguments = ['--scenario', '/path/to/scenarioFile', \
            '--parameter-file', '/path/to/parameterFile']
# If forbiddenFile is provided
arguments = ['--scenario', '/path/to/scenarioFile', \
            '--parameter-file', '/path/to/parameterFile', \
            '--forbidden-file', '/path/to/forbiddenFile']
crace.run(arguments=arguments)
```

5.1.7 Parameters python format

The parameters are stored in a Python object Parameters and its structure is as follows:

all_parameters	Dictionary of ParameterEntry objects, keys are parameter names.
sorted_parameters	List of the parameter names in sampling order.
nb_parameters	Number of parameters.
nb_fixed	Number of fix-valued parameters.
n_variable	Number of variable-valued parameters.

For each ParameterEntry object, its structure is as follows:

name	Parameter name.
switch	Parameter switch. e.g., switches to be used for the parameters on the command line.
type	Parameter type, 'i', 'c', 'r', 'o'.
domain	List of parameter domain, where each list may contain two values [minimum, maximum] for real and integer parameters, or a list of values for categorical and ordinal parameters.
condition	Parameter condition (a code object showing logical expression and its result).
priority	Parameter priority.
depends	List of parameter names from which a parameter is conditional.
isFixed	Logical vector that specifies which parameter is fixed and, thus, it does not need to be tuned.
transform	String that contains the transformation of each parameter. Currently, it can take values "" (no transformation, default) of "log" (natural logarithmic transformation).

To check the details of object Parameters, you should load parameters first using the following command at Python console:

1. Initialize the object Parameters directly:

```
from crace import Parameters
# If neither parameters_file or text is provided,
# 'parameters.txt' in current directory would be loaded as the default
# exec_dir is optional and its default value is current directory
parameters = Parameters(parameters_file='/path/to/parameterFile')
# If forbiddenFile is provided
parameters = Parameters(parameters_file='/path/to/parameterFile', \
                        forbidden_file='/path/to/forbiddenFile')
```

For example, load parameters algorithm, ants and q0 of the ACOTSP scenario:

```
from crace import Parameters
# Read parameters from text
parameters = Parameters(text=' \
algorithm      "--"      c      (as,mmas,eas,ras,acs) \n \
ants           "--ants "  i      (5, 100) \n \
q0             "--q0 "    r      (0.0, 1.0) | algorithm == "acs"')
```

2. Or initialize object CraceOptions first:

```

from crace import Parameters, CraceOptions
# Initialize options using scenario file
options = CraceOptions(scenario_file='/path/to/scenarioFile')

parameters = Parameters(parameters_file=options.parameterFile.value)
# If forbiddenFile is provided
parameters = Parameters(parameters_file=options.parameterFile.value, \
                        forbidden_file=options.forbiddenFile.value)

```

5.2 Target algorithm runner

The evaluation of a candidate configuration on a single instance is done by means of a user-given auxiliary program. The function (or program name) is specified by the option `targetRunner`. The `targetRunner` must return the cost value (e.g., cost of the best solution found) of the evaluation or the time cost to find the solution.



The objective of **crace** is to minimize the cost value returned by the target algorithm. If you wish to maximize, you can multiply the cost by `-1` before returning it to **crace**.

5.2.1 Target runner as an 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> [bound] [boundMax] <configuration>
<code>id_configuration</code>	an alphanumeric string that uniquely identifies a configuration;
<code>id_instance</code>	an alphanumeric string that uniquely identifies an instance;
<code>seed</code>	seed for the random number generator to be used for this evaluation, provided with instances;
<code>instance</code>	string giving the instance to be used for this evaluation;
<code>bound</code>	optional execution time bound. Only provided when the <code>boundMax</code> option is set in the scenario, see Section 6.5 ;
<code>boundMax</code>	optional the default maximal execution time bound. Only provided when the <code>boundMax</code> option is set in the scenario, see Section 6.5 ;
<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 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 and/or when the `capping` option is enabled). 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 **crace** in parallel using different values for `execDir`, without the runs interfering with each other.

5.2.2 targetRunnerLauncher as an executable command line

When the script file provided by `targetRunner` cannot be executed directly, for example, you are using Windows, or this file `target-runner` is in `.bat`, `.py` or `.Rscript` format, you must use option `targetRunnerLauncher` instead of `targetRunner`. This option provides an executable command line that call a script file using specific executable software for each configuration that executes the target algorithm on a particular instance.

```
targetRunner = ""

## An executable command line that call a script file using specific executable
## software for each configuration that executes the target algorithm on a
## particular instance, when `targetRunner` cannot be executed directly
## (e.g., a Python script in Windows).
targetRunnerLauncher = "python /path/to/targetRunner.py"
```

Figure 5: Add `targetRunnerLauncher` in scenario file.

5.2.3 targetRunnerRetries as the times to repeat targetRunner

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

If the times to repeat `targetRunner` is exhausted but the execution is still unsuccessful, **crace** may produce the following error:



Error: Experiment [id] exceeded the number of retries

5.2.4 Load targetRunner

To provide the executable program to **crace**, you can:

1. Use option `targetRunner` and `targetRunnerRetries` (if necessary) in the `scenario.txt` as shown in Figure 6.

```
## Define a script that evaluates a configuration of the target algorithm on
## a particular instance. See templates.
targetRunner = "/path/to/targetRunner"

## Number of times to retry a call to targetRunner if the call failed.
# targetRunnerRetries = 0
```

Figure 6: Add `targetRunner` in scenario file.

2. Add option `targetRunner` and `targetRunnerRetries` (if necessary) when calling **crace** using arguments.

- Call **crace** at the shell as follows:

```
crace \
  --scenario /path/to/scenarioFile \
  --target-runner /path/to/targetRunner
# If targetRunnerRetries is necessary
crace \
  --scenario /path/to/scenarioFile \
  --target-runner /path/to/targetRunner \
  --target-runner-retries 3
```

- Call **crace** at the Python console as follows:

```
import crace
arguments = ['--scenario', '/path/to/scenarioFile', \
            '--target-runner', '/path/to/targetRunner']
# If forbiddenFile is provided
arguments = ['--scenario', '/path/to/scenarioFile', \
            '--target-runner', '/path/to/targetRunner', \
            '--target-runner-retries', 3]
crace.run(arguments=arguments)
```

5.3 Training instances

The **crace** options `trainInstancesDir` and `trainInstancesFile` specify where to find the training instances. Optionally, when executing **crace** from the Python console, the instances can be specified directly as a Python object (see [Section 5.3.2](#)). By default, the value of `trainInstancesFile` is empty. This means that **crace** 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. Within each line, elements separated by white-space will be parsed as separate arguments to be supplied to `targetRunner`. This allows defining instance-specific parameter settings. Quoted strings will be parsed as a single argument. The following example shows a training instance file for the **ACOTSP** scenario:

```
# 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 7: Training instances file (`instances_list.txt`) for tuning **ACOTSP**.

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:

```
target-runner 1 4 5718 ./Instances/100/100-1_100-2.tsp --time 1 --alpha 2.92 ...
```

Training instances do not need to be files, **crace** just passes the elements of each line as arguments to `targetRunner`, thus each line may denote the name of a benchmark function or a label, plus instance-specific settings, that the target algorithm understands. Each line may even be the command-line parameters required to call an instance generator within `targetRunner`. When the instances do not represent actual files, then `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
```

Figure 8: Example of training instances file (`instances_function.txt`) where instances do not correspond to files.

5.3.1 Load training instances

To provide the training instances to **crace**, you can:

1. For the example shown in [Figure 8](#), set the related part of scenario file as in [Figure 9](#).

```
## Directory where training instances are located; either absolute path or
## relative to current directory. If no trainInstancesFile is provided, all
## the files in trainInstancesDir will be listed as instances.
trainInstancesDir = ""

## File that contains a list of training instances and optionally additional
## parameters for them. If trainInstancesDir is provided, crace will search
## for the files in this folder.
trainInstancesFile = "/path/to/instances_function.txt"
```

Figure 9: Add training instance functions in scenario file.

2. Add option `trainInstancesDir` and `trainInstancesFile` (if necessary) when calling **crace** using arguments.

- Call **crace** at the shell as follows:

```
crace \
  --scenario /path/to/scenarioFile \
  --train-instances-dir /path/to/trainInstancesDir
# If trainInstancesFile is provided
crace \
  --scenario /path/to/scenarioFile \
  --train-instances-dir /path/to/trainInstancesDir \
  --train-instances-file /path/to/trainInstancesFile
```

- Call **crace** at the Python console as follows:

```
import crace
arguments = ['--scenario', '/path/to/scenarioFile', \
            '--train-instances-dir', '/path/to/trainInstancesDir']
# If forbiddenFile is provided
arguments = ['--scenario', '/path/to/scenarioFile', \
            '--train-instances-dir', '/path/to/trainInstancesDir', \
            '--train-instances-file', '/path/to/trainInstancesFile']
crace.run(arguments=arguments)
```

5.3.2 Instances python format

The structure of class `Instances` is as follows:

<code>names</code>	List of instance names (file paths or strings).
<code>instances</code>	List of <code>InstanceEntry</code> objects that are used already in crace.
<code>instances_ids</code>	List of instance ids that are being using in crace, it's a subset of already used instances when option <code>instancePatience</code> is enable.
<code>instances_all</code>	List of all <code>InstanceEntry</code> objects initialized for training.
<code>instances_ids_all</code>	List of all instance ids initialized for training.
<code>instances_num</code>	Number of <code>InstanceEntry</code> objects used for training.
<code>fixsize</code>	Size of provided training instances.
<code>size</code>	Size of instances in current stream.
<code>_next_instance</code>	ID of the next instance to be used in crace (≥ 1).
<code>tnames</code>	List of test instance names (file paths or strings).
<code>tinstances</code>	List of <code>InstanceEntry</code> objects used for testing.

<code>tinstances_id</code>	List of test instance ids.
<code>tsize</code>	Number of test instances in the set.

Each `InstanceEntry` object is in the type of `collections.namedtuple` and its structure is as follows:

```
collections.namedtuple("InstanceEntry", ["instance_id", "path", "seed"])
```

To check the details of object `Instances`, you should initialize this object first. Training instances must be provided by one out of `instances_list`, `instances_dir` and `instances_file`.

1. Initialize the object `Instances` directly for the example shown in [Figure 9](#):

```
from crace import Instances
# If none of the three parameters is provided,
# './Instances' in current directory would be loaded as the default
# value of instances_dir
# exec_dir is optional and its default value is current directory
instances = Instances(instances_dir="", instances_file="/path/to/instances_function.txt")
```

The previous example would be equivalent to:

```
from crace import Instances
instances = Instances(instances_list=[
    "rosenbrock_20 --function=12 --nvar 20", \
    "rosenbrock_40 --function=12 --nvar 30", \
    "rastrigin_20 --function=15 --nvar 20", \
    "rastrigin_40 --function=15 --nvar 30"])
```

2. Or initialize object `CraceOptions` first:

```
from crace import Instances, CraceOptions
# Initialize options using scenario file
options = CraceOptions(scenario_file='/path/to/scenarioFile')

instances = Instances(instances_dir=options.trainInstancesDir, \
    instances_file=options.trainInstancesFile)
```

By default, **crace** assumes that the target algorithm is stochastic, thus, the same configuration can be executed more than once on the same instance and obtain different results. In this case, **crace** generates pairs (instance, seed) by generating a random seed for each instance. In other words, configurations evaluated on the same instance use the same random seed. This is a well-known variance reduction technique called *common random numbers* [4]. If all available pairs are used within a run of **crace**, new pairs are generated with different seeds, that is, a configuration evaluated more than once per instance will use different random seeds.

If option `seed` is provided, the seed used to generate random seeds for provided instances will be based on option `seed`. Otherwise, these seeds are generated totally at random.

crace can randomly re-orders the sequence of provided instances. 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 re-order instances to prevent biasing the tuning due to the instance order. See also [Section 6.3](#)

5.4 Initial configurations

The scenario option `configurationsFile` is optional. It allows specifying a text file that contains an initial set of configurations to start the execution of **crace**. If the number of initial configurations supplied in the file is less than the minimum number of alive configurations (option `nbConfigurations`) required by **crace**, additional configurations will be sampled uniformly. If no configuration is provided as the initial or all the provided ones are forbidden (Section 5.1.4), all required number of configurations will be sampled uniformly.

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 5.1.4), if any.

Figure 10 gives an example file that corresponds to the **ACOTSP** scenario.

```
## Initial candidate configuration for crace
algorithm localssearch alpha beta rho ants nnls dlb q0 rasrank elitistants
as          0          1.0  1.0  0.95 10  NA  NA  NA  NA  NA
```

Figure 10: 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 **crace** to start the search from those parameter values and attempt to improve their performance.

5.4.1 Load initial configurations

To provide initial configurations to **crace**, you can:

1. Use option `configurationsFile` in the `scenario.txt`.

```
## File that contains a table of initial configurations. If empty or NULL, all
## initial configurations are randomly generated.
configurationsFile = "/path/to/default.txt"
```

Figure 11: Add `configurationsFile` in scenario file for tuning **ACOTSP**.

2. Add option `configurationsFile` when calling **crace** using arguments.

- Call **crace** at the shell as follows:

```
$PATHON_HOME/bin/crace \
--scenario /path/to/scenarioFile \
--configurations-file /path/to/configurationsFile
```

- Call **crace** at the Python console as follows:

```
import crace
arguments = ['--scenario', '/path/to/scenarioFile', \
            '--configurations-file', '/path/to/configurationsFile']
crace.run(arguments=arguments)
```

5.4.2 Configurations python format

The structure of class `Configurations` is as follows:

<code>all_configurations</code>	Dictionary of ConfigurationEntry objects, keys are configuration ids.
<code>n_config</code>	Number of configurations in <code>all_configurations</code> .
<code>alive_ids</code>	List of ids which are considered alive.
<code>hash_configurations</code>	List of configuration hashes used mainly to find repeated configurations.
<code>parameters</code>	object Parameters.

Each ConfigurationEntry object includes attributes as follows:

<code>id</code>	configuration ID.
<code>parent_id</code>	parent configuration ID.
<code>model_id</code>	model ID.
<code>param_values</code>	Dictionary of parameter values (keys are parameter names).
<code>alive</code>	Boolean, indicates if the configuration is alive.
<code>cmd</code>	Command line string of the configuration.

In order to check the details of object Configurations, object Parameters must be initialized in advance (Section 5.1.7), then you can call command as following:

```
from crace import Configurations
# Initialize Configurations
configurations = Configurations(parameters=parameters)
# Load initial configurations
configurations.add_from_file(configurations_file='/path/to/configurationsFile')
```

6 Crace options

Apart from the options related to scenario mentioned in Section 5, **crace** provides some other options for the tuning. All **crace** options can be provided in the scenario file, whose filesystem path is the value of option `scenarioFile` (`--scenario`) or as arguments. The format of scenario file is one **crace** option per line. Within each line, value is assigned to the corresponding option using an equation

$$\langle \text{name} \rangle = \langle \text{value} \rangle$$

where $\langle \text{name} \rangle$ is the defined name of option. The format to add options via arguments depends on the type of options. In current **crace** version, the six basic types of **crace** options supported are the following:

- *Enabler* options: 'e'. They are signal execution options that are without any values. e.g.: `--help`, `--check`, `--version` and `--init`. These options can only be called via arguments.
- *Integer* options: 'i'. They are numerical options that can take only integer values within the given range. The range is specified as an interval ($\langle \text{lower bound} \rangle, \langle \text{upper bound} \rangle$). This interval is closed, that is, the parameter value may eventually be one of the bounds. e.g.: `debugLevel` (`--debug-level`), `firstTest` (`--first-test`), etc.
- *Real* options: 'r'. They are numerical options that can take floating-point values within a given range. The range is specified as an interval ($\langle \text{lower bound} \rangle, \langle \text{upper bound} \rangle$). This interval is closed, that is, the parameter value may eventually be one of the bounds. e.g.: `confidence` (`--confidence`).
- *String* options: 's'. They can only take specific string values within the given domain. These values may be specified as a set of allowed parameters, such as `testType` (`--test-type`) and `domType` (`--dom-type`), or they may be provided by a hidden list, for example, `man` (`-m`), whose value must be the name of an available **crace** option.

- *Boolean* options: 'b'. They can only take 0 as **False** and 1 as **True**. e.g.: `capping` (`--capping`), `mpi` (`--mpi`), etc.
- *File* options: 'p'. They are string options that can parse values to a absolute filesystem path. Some file options have default values, such as `scenarioFile` (`--scenario`) with default value `./scenario.txt` and `trainInstancesDir` (`--train-instances-dir`) with default value `./Instances`, while some options do not have default values, such as `testInstancesDir` (`--test-instances-dir`).

To check more information of all available **crace** options, you can call command `crace -h` for the short or `crace --help` for the more. Also, use command `crace --man <name>` to see details of a specified option. In the remainder of this guide, we use the `<name>` to refer to an option.

6.1 Crace options python format

All **crace** options are stored in a Python object `CraceOptions` and its structure is as following:

<code>options</code>	List of option names.
<code>arguments</code>	List of arguments calling crace .
<code><name></code>	A specific <i>TypeOption</i> object of <code><name></code> .

Based on the type of **crace** options, there are `EnablerOption`, `IntegerOption`, `RealOption`, `StringOption`, `BooleanOption` and `FileOption` objects. Each *TypeOption* object has the same attributes as listed:

<code>name</code>	Name the selected option.
<code>type</code>	Short type of the selected option.
<code>short</code>	Short command to call the selected option.
<code>long</code>	Long command to call the selected option.
<code>value</code>	Current value of the selected option.
<code>default</code>	Default value of the selected option.
<code>domain</code>	Domain of the selected option.
<code>description</code>	Brief description of the selected option.
<code>vignettes</code>	Details of the selected option.

To check the details of object `CraceOptions`, you scan use the following command at Python console:

```
from crace import CraceOptions
# If neither scenario_file or arguments is provided,
# 'scenario.txt' in current directory would be loaded as the default
options = CraceOptions()
```

For example, at Python console, you can use the following command to check the details of option `scenarioFile`:

```
options.scenarioFile.value
# Output
'/path/to/scenario.txt'
```

6.2 Tuning budget

Before setting the budget for a run of **crace**, please consider the number of parameters that need to be tuned, available processing power and available time. The optimal budget depends on the difficulty of the tuning scenario, the size of the parameter space and the heterogeneity of the instances. Typical values range from 1 000 to 100 000 runs of the target algorithm, although smaller and larger values are also possible.

Crace provides two options for setting the total tuning budget (`maxExperiments` and `maxTime`). The option `maxExperiments` limits the number of executions of `targetRunner` performed by **crace**. The option `maxTime` limits the total time of the `targetRunner` executions. When this latter option is used, `targetRunner` must return the evaluation cost together with the execution time ("cost time").



When the goal is to minimize the computation time of an algorithm, and you wish to use `maxTime` as the tuning budget, `targetRunner` must return the time also as the evaluation cost, that is, return the time twice as "time time".



When both `maxTime` and `maxExperiments` are used, **crace** could be terminated if either budget is exhausted. However, the standard output is splitted based on `maxTime`.

6.3 Heterogeneous scenarios

We classify a scenario as homogeneous when the target algorithm has a consistent performance regarding the instances; roughly speaking, good configurations tend to perform well and bad configurations tend to perform poorly on all instances of the problem. By contrast, in heterogeneous scenarios, the target algorithm has an inconsistent performance on different instances, that is, some configurations perform well for a subset of the instances, while they perform poorly for a different subset.

When facing a heterogeneous scenario, the first question should be whether the objective of tuning is to find configurations that perform reasonably well over all instances, even if that configuration is not the best one in any particular instance (a generalist). If this is not the goal, then it would be better to partition instances into more similar subsets and execute **crace** separately on each subset. This will lead to a portfolio of algorithm configurations, one for each subset, and algorithm selection techniques can be used to select the best configuration from the portfolio when facing a new instance.

To make sure **crace** is not misled by results on few instances, it may be useful to increase the number of instances executed before doing a statistical test using the option `firstTest`, e.g., `--first-test 10` (default value is 5 when `capping` is disabled and otherwise 1), in order to see more instances before discarding configurations.

The option `eachTest` (default value is 1) defines the number of instances evaluated between elimination tests. The value of `eachTest` must not be greater than that of `firstTest`.



The number of used instances in the first elimination test is only determined by `firstTest`. From the second test, the number of used instances $\text{len}(\text{Instances.instances}) - \text{options.firstTest.value}$, (Section 5.3.2) must be a multiple of `eachTest`.

When option `eachTest` is greater than 1, each time **crace** samples `eachTest` new instances for the tuning.

If finding an overall good configuration for all the instances is the objective, then we recommend that instances are randomly sampled (option `sampleInstances`), unless one can provide

the instances in a particular order that does not bias the tuning towards any subset.

Otherwise, **crace** provides an option `instancePatience` to give the opportunity to sample more instances. This integer option means the number of configurations that an instance can undergo without yielding any improvement.

6.4 Choosing the elimination test

As a continuous racing procedure, **crace** tries to do elimination test once one experiment has finished its execution (the number of used instances meets `firstTest` and `eachTest`). In order to decrease the frequency to call elimination test and save runtime, **crace** provides an option `testExperimentSize`. When `testExperimentSize` is enabled, the status of experiments finishing the execution will be set as 'testing' (Section 10.2.1) and waiting to be tested.

6.4.1 Statistical test

The statistical test used in **crace** identifies statistically bad performing configurations that can be discarded from the race in order to save budget. Different statistical tests use different criteria to compare the cost of the configurations, which has an effect on the tuning results.

Crace provides two types of statistical tests (option `testType`). Each test has different characteristics that are beneficial for different goals:

- Friedman test (F-test): This test uses the ranking of the configurations to analyze the differences between their performance. This makes the test suitable for scenarios where the scale of the performance metric is not as important to assess configurations as their relative ranking. This test is also indicated when the distribution of the mean performances deviates greatly from a normal distribution. For example, the ranges of the performance metric on different instances may be completely different and comparing the performance of different configurations using the mean over multiple instances may be deceiving. We recommend to use the F-test (default when `capping` is not enabled) when tuning for solution cost and whenever the best performing algorithm should be among the best in as many instances as possible.
- Student's t-test (t-test): This test uses the mean performance of the configurations to analyze the differences between the configurations.³ 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, in particular, when the target algorithm is minimizing computation time and, in general, whenever the best configurations should obtain the best average solution cost.

Besides, two adjustment methods are provided for t-test:

- Bonferroni (t-test-bonferroni): p-values are multiplied by the number of comparisons. It is used to reduce the chances of obtaining false-positive results (Type I errors) when performing multiple comparisons.
- Holm (t-test-holm): Iteratively compare the smallest p-value with $\alpha/(n - i + 1)$, where α is the significance level, i is the rank in the sorted p-values and n is the total number of tests. It is an improved version of the Bonferroni correction that is less conservative while still controlling the family-wise error rate.

³The t-test does not require that the performance values follow a normal distribution, only that the distribution of sample means does. In practice, the t-test is robust despite large deviations from the assumptions.

The confidence level of the tests may be adjusted by using the option `confidence`. Increasing the value of `confidence` leads to a more strict statistical test. Keep in mind that a stricter test will require more budget to identify which configurations perform worse. A less strict test discards configurations faster by requiring less evidence against them and, therefore, it is more likely to discard good configurations.

6.4.2 Aggressive test

The aggressive test is used in **crace** to aggressively discard bad performing configurations in order to save budget and resource. Derived from trajectory-preserving capping and aggressive capping [2], **crace** provides two types of aggressive tests (option `domType`):

- Dominance test (`dom`): This test compares the mean results of the best configuration on all used instances (I_{best}) with the mean results of the new configuration on the used instances (I_θ), where:

$$|I_{best}| \geq |I_\theta|, I_\theta \in I_{best} \quad (1)$$

- Adaptive dominance test (`adaptive-dom`): This test compares the mean results of the best configuration with the new configuration on the same selected instances. We recommend to use `adaptive-dom` (default when `capping` is enabled) when tuning for minimizing the computation time.

Observe that all comparisons between configurations using methods from statistical test (Section 6.4.1) in **crace** are to find the differences. For some homogeneous scenarios, there may be no significant difference in configuration performance, which requires the application of aggressive test. In this case, if the new configuration performs worse than the best configuration, it will be discarded directly. While the new configuration performs better than the best, **crace** will sample new instance to do one more elimination test for these two configurations.

6.5 Tuning for minimizing computation time

When using **crace** for tuning algorithms that only report computation time to reach a target, `targetRunner` should return the execution time of a configuration instead of solution cost. When using `maxTime` as the budget, this means that `targetRunner` must return twice the execution time since the first value is the minimization objective and the second value is used to track the budget consumed.

Crace implements **capping** and **adaptive capping** mechanisms. Capping and adaptive capping [2] are configuration techniques that avoid the execution of long runs of the target algorithm, focusing the configuration budget in the evaluation of the best configurations found. This is done by bounding the execution time of each configuration based on the best performing candidate configurations.

Capping is to calculate the bound for new configurations based on the best configuration on all used instances and to do elimination test using dominance test (Section 6.4.2), while adaptive capping is to adaptively update the bound for new configurations based on the best configuration and use adaptive dominance test (Section 6.4.2) to discard the bad configurations.

To use capping and adaptive capping, the `capping` option must be enabled and the `domType` **crace** option optionally be set (default value is `adaptive-dom`, means adaptive capping). When elite configuration list is updated, **crace** calculates a **priori bound** based on the execution times of the elitist configuration. Before evaluating candidate configurations on an instance, **crace** updates the execution bound (**post bound**) based on the **priori bound** and the execution times

on their used instances. Figure 12 shows examples of the two bounds for capping and adaptive capping, respectively.

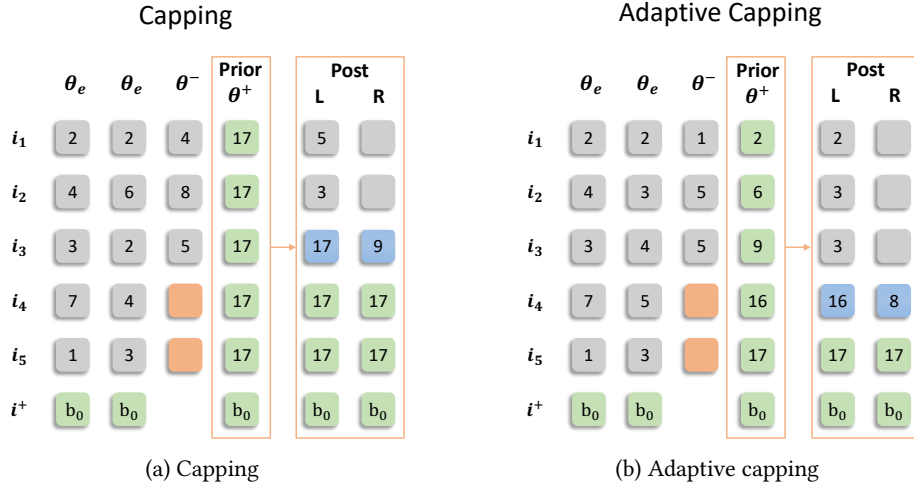


Figure 12: Capping and Adaptive Capping in **crace**. Here, all numbers are in cpu seconds. Solid boxes are experiments, among which the grey is the finished, the red is the discarded, the green is the newly sampled and the blue is the next. θ are configurations, among which θ_b are the best, θ_w are the worst and θ_{new} are the newly sampled. I are instances, among which I_{new} are the newly sampled. b_0 is the default time bound for target algorithm execution (option boundMax). Post-L and R are the experiments before and after updating the bound, respectively.

As shown in Figure 12a, for **capping**, the priori execution bound of new configurations (θ_{new}) is calculate by summing execution bounds of the elitist configuration (the first θ_b) on its used instances (I_{used}). Thus, the priori bound for new configurations on all instances (b^{priori}) is:

$$b^{priori} = \min\{\sum_{k=1}^n p_{k,best} + \epsilon, b_0\}, \quad n = |I^{used}| \quad (2)$$

While as shown in Figure 12b, for **adaptive capping**, the priori execution bound of the new configurations is calculated by summing execution bounds of the elitist configuration on the selected instances ($I_{selected}$). Thus, the priori bound of the new configuration on instance i (b_i^{priori}) is:

$$b_i^{priori} = \min\{\sum_{k=1}^i p_{k,best} + \epsilon, b_0\}, \quad i = (1, 2, \dots, |I^{used}|) \quad (3)$$

Both in Equation 2 and Equation 3, $p_{k,best}$ is the returned runtime of best configuration on instance k and ϵ is a constant that is taken as 0.001. This small constant is added to account for time measurements errors.

From Equation 2 and Equation 3, priori bound is only related to the best configuration and its used instances. Therefore, **crace** calculates the priori bound for all new configurations on the used instances only when the best configuration is updated.

The post execution bound for both capping mechanisms is updated based on the priori bound of the next experiment and the finished experiments of current configuration. Thus, the post bound of configuration j on instance i is:

$$b_{i+1,j}^{post} = b_{i+1,j}^{priori} - \sum_{k=1}^i p_{i,j} + \epsilon, \quad i = |I^{finished}| \quad (4)$$

When elite configurations dominate new configurations, the new ones are eliminated from the race. Otherwise, one more execution on a new instance will be taken to compare their performance again.



The default elimination test when **capping** is enabled is **adaptive-dom**. This test is more appropriate to configure algorithms for optimizing runtime (see [Section 6.4.2](#)).

The execution bound is constantly adjusted by **crace** based on the best configurations times, nevertheless, a maximum execution time (b_0) is never exceeded. This maximum execution time must be defined in the configuration scenario when **capping** is enabled. To specify the maximum execution bound for the target runner executions use the **boundMax** option. The final execution bound ($p_{i,j}$) of configuration j on instance i is calculated by:

$$p_{i,j} = \begin{cases} b_0 & \text{if } p'_{i,j} \geq b^{\max}, \\ \min\{b_i^{\text{post}}, b_0\} & \text{if } p'_{i,j} \leq 0, \\ p'_{i,j} & \text{otherwise;} \end{cases} \quad (5)$$

where $p'_{i,j}$ is real execution time of configuration j on instance i using the **boundDigits** option to define the precision of the time bound provided by **crace**, whose default setting is 0.

Timed out executions occur when the maximum execution bound (**boundMax**) is reached and the algorithm has not achieved successful termination or a defined quality goal. Bounded executions are executions that do not achieve successful termination or a defined quality goal in the execution bound ($p_{i,j}^{\text{post}}$) set by **crace**, which is smaller than **boundMax**. The **boundAsTimeout** option replaces the evaluation of bounded executions by the **boundMax** value.



In current **crace** version 1.4, option **boundMaxDEV** is provided to set the maximum execution bound especially for training phase when different execution boundaries for training and test phases are required.

7 Parallelization

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

1. **Parallel processes:** The option **parallel** executes multiple calls to **targetRunner** in parallel within a single computer, by means of the **parallel** option. For example, adding **--parallel N** to the command line of **crace** will launch in parallel up to N calls of the target algorithm. When using this option within a computing cluster, **crace** will be submitted as a *job* in some way that tells the cluster to “reserve” N CPUs (or tasks depending on the cluster) within a single cluster node (a single machine).
2. **MPI:** By enabling the option **mpi**, calls to **targetRunner** will be executed in parallel by using the message passing interface (MPI) protocol (requires a **MPI** library and the **mpi4py** Python package). In this case, the option **parallel** controls the resource applied and the number of slave nodes used by **crace**. 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. **MPICH**, **OpenMPI** and **Intel MPI** are available libraries suitable for different operating systems. MPI is commonly available in computing clusters and requires launching **crace** in some particular way. An example script for using MPI mode in a SLURM cluster is given at `$CRACE_HOME/examples/batchmode-cluster/parallel-crace-mpi`.



Integer option `testParallel` is provided since **crace** version 1.0.1, which is designed particularly for the parallelization in testing phase. 0 and 1 mean that the testing phase will be executed without parallelization. Note that `testParallel` must not be greater than `parallel`.

The best option will depend on the resources available to you. Option 1 is usually the fastest and simplest to setup. Running on a node (machine) with 128 CPUs will be faster than running on 8 nodes with 16 CPUs because the communication between nodes required by MPI can be slow depending on the cluster. Option 2 may be faster if **crace** has more alive configurations than the number of CPUs of a single node. However, depending on the configuration of your cluster, requesting many CPUs may require waiting in the queue a long time.

As a rule-of-thumb, if you only have access to a single machine, then you only need option 1. If you have access to a computing cluster with multiple machines, then use option 1 with the maximum number of CPUs that a single node has in your computing cluster. If that number is 64 or more, it should be enough unless a single run of **crace** evaluates thousands of experiments. Otherwise, investigate option 2.

In addition, **crace** provides an option “`bindCores`” to enable CPU pinning according to the physical topology of the underlying hardware. When this option is activated, **crace** can automatically detect the available cores and bind each worker to a dedicated core, ensuring that each worker has exclusive access to the assigned core and its associated resources, and each execution and its sub-processes on the worker will inherit the same CPU affinity. This mechanism can effectively decrease the contention for shared resources and improve the stability of the execution performance of the target algorithm.

8 Testing (Validation) of configurations

Once the tuning process is finished, **crace** returns a set of configurations corresponding to the elite configurations at the end of the run, ordered from best to worst. In order to evaluate the generality of these configurations without looking at their performance on the training set, **crace** 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 and `targetRunner`.

The test instances can be specified by the options `testInstancesDir` and/or `testInstancesFile`, or by setting directly the object `Instances` at the Python console. These options behave similarly to their counterparts for the training instances (Section 5.3). In particular, each test instance is assigned a different seed in the same way as done for the training instances. In principle, **crace** evaluates each configuration on each testing instance just once, because evaluating one run on n instances is always better than evaluating n' runs on n/n' instances [1]. However, if the number of instances is limited, one can always duplicate instances as needed in the `testInstancesFile`, and **crace** will assign a different random seed to each instance. An example of the output produced by **crace** when testing is shown in Figure 13.

```
# Initializing tester
# Start testing ...
# configurations: 2
110: --acs --localsearch 3 --alpha 4.4317 --beta 1.2857 --rho 0.5231 --ants 31 --q0 0.5771 --nnls 20
↳ --dlb 1
104: --acs --localsearch 3 --alpha 2.2388 --beta 6.5609 --rho 0.9484 --ants 68 --q0 0.8685 --nnls 8
↳ --dlb 0
# instances: 10
# Test 2 configuration(s): 110
# Experiment: 1      config: 110      instance: 1      result: 32873591.000000      time: 24.89728808403015
# Experiment: 9      config: 110      instance: 5      result: 32974440.000000      time: 24.85494899749756
# Experiment: 11     config: 110      instance: 6      result: 32646813.000000      time: 24.841660022735596
```

```

# Experiment: 7      config: 110      instance: 4      result: 32740514.000000      time: 24.93803596496582
# Experiment: 3      config: 110      instance: 2      result: 32502434.000000      time: 24.996917009353638
# Experiment: 13     config: 110      instance: 7      result: 32677242.000000      time: 24.959688186645508
# Experiment: 5      config: 110      instance: 3      result: 33054112.000000      time: 25.24346375465393
# Experiment: 4      config: 104      instance: 2      result: 32636788.000000      time: 24.61223292350769
# Experiment: 19     config: 110      instance: 10     result: 32186327.000000      time: 24.65356206893921
# Experiment: 2      config: 104      instance: 1      result: 32773590.000000      time: 24.670124053955078
# Experiment: 6      config: 104      instance: 3      result: 33158754.000000      time: 24.693687677383423
# Experiment: 15     config: 110      instance: 8      result: 32609176.000000      time: 24.74283528327942
# Experiment: 17     config: 110      instance: 9      result: 32673826.000000      time: 24.741806268692017
# Experiment: 8      config: 104      instance: 4      result: 32889300.000000      time: 24.717500925064087
# Experiment: 10     config: 104      instance: 5      result: 32940603.000000      time: 23.233488082885742
# Experiment: 12     config: 104      instance: 6      result: 32717392.000000      time: 23.234249114990234
# Experiment: 20     config: 104      instance: 10     result: 32246861.000000      time: 23.10284996032715
# Experiment: 16     config: 104      instance: 8      result: 32788257.000000      time: 23.23434615135193
# Experiment: 18     config: 104      instance: 9      result: 32678792.000000      time: 23.204938173294067
# Experiment: 14     config: 104      instance: 7      result: 32757248.000000      time: 23.427215814590454
Before closing
Pool closed
Master stopped
Shutdown complete
Stopped

```

Figure 13: Sample text output of **crace** when evaluating on test instances.

The option `testNbElites` controls the number of elite configurations finally evaluated during the testing phase. In particular, setting `testNbElites = 1` will test only the elitist configuration that returned at the end of the training phase.

The testing can be also (re-)executed independently by using the option `onlytest` as follows:

1. **From the command-line console**, call the command:

```

crace -o -i configurations.txt --scenario /path/to/scenarioFile
# or from an exsisting logDir
crace -o -i /path/to/logDir/elite.log --scenario /path/to/scenarioFile

```

2. **From the Python console**, evaluate:

```

import crace
crace.run(['-o', '-i', 'configurations.txt', '--scenario', '/path/to/scenarioFile'])
# or from an exsisting logDir
crace.run(['-o', '-i', '/path/to/logDir/elite.log', \
          '--scenario', '/path/to/scenarioFile'])

```

where `configurations.txt` and `/path/to/logDir/elite.log` both have the same format as the set of initial configurations (Section 5.4).

9 Recovering crace runs

Problems like power cuts, hardware malfunction or the need to use computational power for other tasks may occur during the execution of **crace**, terminating a run before completion. During the racing, **crace** saves all plain text log files in directory `execDir/logDir` that not only contains information about the tuning progress (Section 10.2), but also internal information that allows recovering an incomplete execution.

To recover an incomplete **crace** run, set the option `recoveryDir` to the `execDir` previously used, and **crace** will continue the execution from the last saved status. In current version, **crace** can restore the history of the racing except for the random generator.



External factors, such as CPU load and disk caches, may affect the target algorithm and that may affect the results. So even though **crace** executes with the same seed and scenario settings, the final returned configuration could be different.

You can specify the `recoveryDir` from the command-line or from the scenario file, and execute **crace** as described in [Section 4](#). For example, from the command-line use:

```
crace --recovery-file /path/to/execDir
```

Or from the Python console, evaluate:

```
import crace
crace.run(['--recovery-file', '/path/to/execDir'])
```

In both ways, all options are set as the same values to the previous run.



crace provides two options `execDir` (default value is current directory) and `logDir` (default value is `crace_log`) to customise the folder to save all plain text log files generated during the run.



When recovering a previous run, **crace** will try to load crace log files (`parameters.log`, `config.log` and `exps_fin.log` are necessary) from an existing `logDir` (without checking the name of `logDir`) in the provided folder `recoveryDir`.



When recovering a crace run, **crace** will save new log files in directory `execDir/logDir`. Thus, you must specify different directories for the combination of `execDir/logDir` from `recoveryDir/logDir`. Here is the command-line to recovery:

```
crace -r /path/to/old -e /path/to/new
```

For a finished **crace** execution, using `recoveryDir` and `testNbElites` can execute more elite configurations on the testing instances.

```
crace -r /path/to/old -e /path/to/new --test-num-elites 5
```



If `testNbElites` is greater than the number of finally returned elite configurations, all the returned configurations will be selected without a notice.



Only options `testNbElites`, `debugLevel`, `execDir` and `logDir` can be modified when recovering.

10 Output and results

During the execution, **crace** prints information about the progress of the tuning in the standard output. Additionally, some log files in plain text format are saved in `logDir` containing the state of **crace**.

10.1 Text output

[Section C](#) in [Appendix C](#) shows the output of a run of **crace** applied to the **ACOTSP** scenario with 1000 evaluations as budget.

First, **crace** prints the arguments user providing and gives a warning informing that it has found a file with the default scenario filename in current directory and it will use it. Then, information about **crace** options is printed. The options that are not introduced above are:

- `seed` is the number that was used to initialize the random number generator in **crace**.
- `debugLevel` is an integer number to control the level of debug information.
- `logLevel` is an integer number to control if generate log files or not.
- `testType` is to specify the method used for statistical elimination test.
- `eachTest` is the total number of evaluations available for the tuning.
- `firstTest` is the maximum execution time available for the tuning.
- `instancePatience` is the number of configurations that an instance can undergo without yielding any improvement.
- `confidence` is the confidence level for the elimination test.
- `maxExperiments` is the maximum budget of experiments for the tuning.
- `maxTime` is the maximum total execution time in second of experiments for the tuning.
- `nbConfigurations` is the number of alive configurations in **crace**.
- `maxNbElites` is the maximum number of elite configurations selected in **crace**.
- `elitist` is to enable/disable elitist **crace**.
- `modelUpdateByStep` is to control the frequency to update models used to sample new configurations.
- `sampleInsFrequency` is to control the probability for sampling new instance when there are more than one elite configurations.
- `softRestart` is the number of times of the soft-restart strategy applied to avoid premature convergence of the probabilistic model.
- `capping` is to enable/disable the technique designed for minimizing the computation time of the executions.

crace divides the output into multiple slices based on the option . At the beginning of each slice, information about the progress of the execution is printed as follows: `expectSliceBudget` is calculated as $\lceil B/(S^{model} + 1) \rceil$, where B is the total budget and S^{model} is the value of option `modelUpdateByStep` that is calculated as $S^{model} = 2 \times \lfloor 2 + \log_2 N^{param} \rfloor$.

Then a table shows the progress of the continuous execution of current slice. 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 of the elimination test:

- |x| No elimination was performed for this execution. The options `firstTest` and `eachTest` control on which instances the elimination test is performed.
- |-| Elimination test is performed and some configurations have been discarded. The column `Alive` gives an indication of how many configurations have been discarded.
- |. | All alive configurations are elite and no one is discarded after the elimination test.
- |c| **crace** samples new configuration(s) because there is not enough alive configurations or waiting experiments.

| i | **crace** samples new instance(s) because there is a challenge after elimination test, which means there are statistical similar configurations with the most instances when using statistical test or there is a new elite configuration with the most instances when using aggressive test.

| e | **crace** samples new instance(s) based on the option `expectInstances`.

| p | **crace** updates instances stream based on the option `instancePatience`.

Other columns have the following meaning:

Instances: Number of instances used for elimination test (number of total instances). See [Section 10.2](#) for more information.

Alive: Number of alive configurations in elimination test (number of alive configurations in **crace**).

Best: ID of the best configuration according to the instances seen in elimination test (ID of the best configuration according to the instances seen so far in this **crace**).

Mean best: Mean cost value of the best configuration across the instances seen so far in this race. When the global best configuration is different from the one on elimination test, this value is the mean cost of the best configuration from the elimination test.

Exp so far: Number of experiments performed so far.

W time: Wall-clock time spent on **crace**.

T time: Total time spent on all target algorithm executions.

E queue, C queue: Scheduled experiments and configurations that are waiting to be executed in **crace**.

At the end of each slice, there is a short summary:

- `experimentsTestedSoFar` is the number of experiments from the total budget that have been used up to the current iteration.
- `budgetUsedSoFar` is the execution budget used so far in the experiments. It is based on runtime when `maxTime` is enabled, otherwise is based on number of executions.
- `remainingBudget` is the budget of experiments that have not been used yet. It is based on runtime when `maxTime` is enabled, otherwise is based on number of executions.
- `nbNewConfigurations` is the number of new configurations sampled in current slice.
- `nbNewInstances` is the number of new instances sampled in current slice.
- `nbEliminatedConfigurations` is the number of configurations discarded in **crace** from elimination test, which includes statistical test and aggressive test.
- `Elite-so-far` configuration is the ID of elite configuration selected by **crace** so far.
- `mean value` is the mean quality or mean runtime on all instances of the elite-so-far configuration.

- Description of the elite-so-far configuration is the parameters and corresponding values of the elite-so-far configuration.

Finally, **crace** 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.



Option `debugLevel` (`--debug-level`) determines the details printed in the standard output file. With default value of **0**, results are printed only upon elite updates. While with default value of **5**, all results and some additional information, for example, related to restart, are printed.

10.2 Log files

During the racing, **crace** saves all plain text log files in the directory `execDir/logDir` and these files are as listed:

<code>asyncio_log.log</code>	The information/errors about asynchronous jobs.
<code>config_alive.log</code>	The IDs of alive configurations currently.
<code>config.log</code>	The information of all sampled configurations.
<code>execution_log.log</code>	The log information of the executions, enabled when option <code>logLevel > 0</code> .
<code>exps_disc.log</code>	The information of all discarded experiments.
<code>exps_fin.log</code>	The information of all finished experiments.
<code>exps_sub.log</code>	The information of all submitted experiments.
<code>instances_next.log</code>	The ID of next instance should be used in crace .
<code>instances.log</code>	The information of all training instances.
<code>logging_variables.log</code>	The brief information about current racing, enabled when option <code>logLevel ≥ 5</code> : <code>alive_ids</code> , <code>best_id</code> , <code>mean_best</code> , <code>exp_so_far</code> , <code>w_time</code> .
<code>model_disc.log</code>	The information of discarded models, enabled when option <code>logLevel ≥ 5</code> .
<code>models.log</code>	The information of models for currently alive configurations and the configuration IDs using each alive model.
<code>mpi_log.log</code>	The information of jobs on the master node when option <code>mpi</code> is enabled and option <code>logLevel ≥ 4</code> .
<code>mpi_worker_X.log</code>	The information of jobs on the <i>X</i> th slave node when option <code>mpi</code> is enabled and option <code>logLevel ≥ 4</code> .
<code>parameters.log</code>	The information of parameters for the provided scenario.
<code>race_log.log</code>	The information of racing about sampling, testing and eliminating.
<code>race.log</code>	The brief information about current race status, enabled when option <code>logLevel ≥ 5</code> : <code>first_test_done</code> , <code>n_alive</code> , <code>ranks</code> , <code>best_id</code> , <code>n_experiments</code> , <code>n_instances</code> , <code>n_time</code> , <code>_scheduled_jobs</code> , <code>terminated</code> .
<code>scenario.log</code>	Values of all crace options.
<code>slice.log</code>	The brief information of crace when updating the model.
<code>tinstances.log</code>	The information of all testing instances.

Option `readlogs` (`--read`) is used to read log files from a previous **crace** run and you can check all related data at the Python console using the following command:

```
import crace
results = crace.run(['--read', '/path/to/old'])
```

Here, `results` is a `CraceResults` object including attributes:

<code>parameters</code>	Object Parameters (Section 5.1.7).
<code>instances</code>	Object Instances (Section 5.3).
<code>configurations</code>	Object Configurations (Section 5.4).

options	Object CraceOptions (Section 6.1).
experiments	Object Experiments (Section 10.2.1).
models	Object ProbabilisticModel (Section 10.2.2).

10.2.1 Experiments python format

Object Experiments manages all experiments and their status in the continouse racing, including submitted, finished and discarded experiments. Its attributes are as listed:

all_experiments	Dictionary of all ExperimentEntry objects by experiment ID (key).
experiments_by_configuration	Dictionary of lists of experiments IDs by configuration ID (key).
experiments_by_instance	Dictionary of lists of experiments IDs by instance ID (key).
n_discarded	Number of discarded experiments.
discarded_ids	List of experiments that were discarded.
n_complete	Number of finished experiments (finished state).
n_complete_by_config	Dictionary of the number of finished experiments by configuration ID (key).
n_experiments	Number of experiments in the class (all states included).
n_experiments_by_instance	Dictionary of the number of experiments (all states) by instance ID (key).
total_time	Sum of reported times in the experiments (ExperimentEntry.time).

ExperimentEntry contains information about a experiment and its current state, its attributes are as listed:

experiment_id	Unique ID for the experiment.
configuration_id	ID of configuration.
instance_id	ID of the instance.
budget	Budget (runtime) assigned for the experiment.
bound_max	Default budget (runtime) assigned for experiments on new instance.
cmd_line	Part of execution line of the experiment.
param_line	Parameters command line.
quality	Result of the experiment (configuration objctive).
time	Time of the experiment reported by the user.
state	State of the experiment (possible values: pending/waiting/testing/finished/discarded). - pending: experiments waiting to be executed - waiting: experiments waiting to be tested when option <code>targetEvaluator</code> is enabled - testing: experiments waiting to be tested when option <code>testExperimentSize</code> ≥ 1 - finished: experiments finishing elimination test - discarded: experiments discarded from test or for illegal results
creation_time	Time stamp for when the experiment was created (does not imply execution).
start_time	Time stamp for when the experiment started execution.
end_time	Time stamp for when the experiment finished execution.

10.2.2 Models python format

Object ProbabilisticModel manages all LocalModel objects used to sample new configurations in the continouse racing. LocalModel manages a collection of parameter models associated to a configuration, including IntegerModel, IntegerLogModel, ContinuousModel, ContinuousLogModel, OrdinalModel and CategoricalModel classified based on the type of parameters ([Section 5.1.5](#)). The attributes of object ProbabilisticModel are as listed:

models	Dictionary of LocalModel objects with model IDs as keys.
alive_models	Dictionary of models used by alive configurations with model IDs as keys..
alive_model_config	Dictionary of alive configuration IDs with model IDs as keys..
disc_models	Dictionary of discarded models with model IDs as keys.

parameters	Parameters object.
size	Number of models in the object (equal to number of configurations).
hash_models	List of LocalModel object hashes used mainly to find repeated models.

Each LocalModel object is for a group of parameter models for sampling a total configuration. Based on the types of parameters ([Section 5.1.1](#)), there are six *typeModel* models: IntegerModel, IntegerLogModel, ContinuousModel, ContinuousLogModel, OrdinalModel and CategoricalModel.

Each *typeParameter* contains elements as listed:

name	Name the selected parameter.
model_type	Short type of the selected parameter.
domain	Domain of the selected parameter.
model_std	Standard deviation of the distribution of the selected parameter, valid except for CategoricalModel.
probabilities	Probabilities of each allowed value of the selected parameter, only valid for CategoricalModel.

10.2.3 Crace results python format

As mentioned above, **crace** can use option [readlogs](#) to load the data of parameters, instances, configurations, options, experiments and models of the previous run into object CraceResults. The CraceResults object contains the following Python objects:

parameters	A Python object containing the description of the target algorithm parameters. Each parameter is in a ParameterEntry object. See Section 5.1.7 .
instances	A Python object containing the instances used for training and testing (if provided) phases. Each instance is in an InstanceEntry object. See Section 5.3.2 .
configurations	A Python object including all target algorithm configurations generated by crace . It includes a dictionary of ConfigurationEntry objects, and some other elements. See Section 5.4.2 .
options	A Python object including scenario settings for the tuning. See Section 6.1 .
experiments	A Python object including all experiments generated in the racing procedure. Each experiment is in an ExperimentEntry object, including the experiment_id, configuration_id, instance_id and cmd_line to call the target algorithm. See Section 10.2.1 .
models	A Python object including all models used to sample new configurations in the continuous racing procedure. See Section 10.2.2 .
scenario	A dictionary shows necessary crace options and their values for this racing procedure.
all_elites	A list of configuration IDs that are selected as elite configurations at the end of each slice during the racing procedure.
elites	A list of configuration IDs that are returned at the end of racing procedure.
best_id	A configuration ID that finally returned as the elitist configuration of the racing procedure.
state	A dictionary shows brief information about the end of this racing procedure.
slice	A dataframe shows information of each slice in this racing procedure.
summarise	A dictionary shows brief information about the crace version used and this racing procedure.

Take **ACOTSP** shown in [Section C](#) as an example, you can check the details of these objects generating during the continuous racing procedure using the following command:

1. version: the version of crace.

```
results.version
```

2. elites: a list that contains the id(s) of elite configurations returned by **crace**:

```
results.elites
# Output
[110, 104]
```

The configurations are ordered by the mean performance, that is, the ID of the best configuration corresponds to the first ID. Or call using command:

```
results.best_id
# Output
110
```

3. `all_elites`: a list that contains the id(s) of elite configurations returned by **crace** during the whole tuning progress, which can be called using command:

```
results.all_elites
# Output
[2, 4, 6, 7, 17, 31, 32, 48, 53, 55, 72, 110]
```

as well as command `results.training.elites`.

4. `parameters`: The parameters Python object containing the description of the target algorithm parameters. Each parameter is in a `ParameterEntry` object. See [Section 5.1.7](#). At Python console, you can use the following command to check the details of object `Parameters`:

```
# Obtain all_parameters
results.parameters.all_parameters
# Output
{'algorithm': <crace.containers.parameters.CategoricalParameter object at ...
 'alpha': <crace.containers.parameters.RealParameter object at ...
 'ants': <crace.containers.parameters.IntegerParameter object at ...}

# Obtain the parameter names
results.parameters.sorted_parameters
# Output is the sorted parameter names based on the depth of condition.
['algorithm', 'localsearch', 'alpha', 'beta', 'rho', 'ants', 'q0', 'rasrank',
 ↪ 'elitistants', 'nnls', 'dlb']

# Obtain the depth
results.parameters._tree_level()
# Output
{'algorithm': 1, 'localsearch': 1, 'alpha': 1, 'beta': 1, 'rho': 1, 'ants': 1, 'q0': 2,
 ↪ 'rasrank': 2, 'elitistants': 2, 'nnls': 2, 'dlb': 2}
```

You can check each `ParameterEntry` object using the following command:

```
# Show attributes and corresponding values of parameters [parameter_name]
for x in vars(results.parameters.all_parameters[parameter_name]).items():
    print(x)
# Output of the ACOTSP scenario when parameter_name='dlb'
('name', 'dlb')
('switch', '--dlb ')
('type', 'c')
('domain', ['0', '1'])
('condition', <code object <module> at ..., file "localsearch in "(1,2,3)""", line 1>)
('priority', 2)
('depends', ('localsearch',))
('is_fixed', False)

# Or show selected attributes for all parameters
for _, v in results.parameters.all_parameters.items():
    print(v.name, v.switch, v.type, v.domain)
# Output of the ACOTSP scenario
algorithm -- c ['as', 'mmas', 'eas', 'ras', 'acs']
localsearch --localsearch c ['0', '1', '2', '3']
```

```
alpha --alpha r [0.0, 5.0]
beta --beta r [0.0, 10.0]
rho --rho r [0.01, 1.0]
ants --ants i [5, 100]
q0 --q0 r [0.0, 1.0]
rasrank --rasranks i [1, 100]
elitistants --elitistants i [1, 750]
nnls --nnls i [5, 50]
dlb --dlb c ['0', '1']
```

5. instances: The instances Python object containing the instances used for training and testing (if provided). Each instance is in an InstanceEntry object. See [Section 5.3.2](#). At Python console, you can use the following command to check the information of object Instances:

```
results.instances.names
# Output
['/path/to/2000-520.tsp', '/path/to/2000-521.tsp', '/path/to/2000-522.tsp', \
'/path/to/2000-523.tsp', '/path/to/2000-524.tsp', '/path/to/2000-525.tsp', \
'/path/to/2000-526.tsp', '/path/to/2000-527.tsp', '/path/to/2000-528.tsp', \
'/path/to/2000-529.tsp']

results.instances.fixsize
# Output
10

results.instances.instances_num
# Output
60

# Show all InstanceEntry objects
results.instances.instances_all
# Output
[InstanceEntry(instance_id=1, path='/path/to/2000-522.tsp', seed=9469788), \
 InstanceEntry(instance_id=2, path='/path/to/2000-521.tsp', seed=3963921), \
 ..., \
 InstanceEntry(instance_id=60, path='/path/to/2000-524.tsp', seed=8069403)]

# Show all used instances for training
results.instances.get_used_instances()
# Output
      instance_id  path                seed
...
49  50            /path/to/2000-529.tsp  8803211
50  51            /path/to/2000-528.tsp  4980953
```

Similarly, to check the instances used for testing:

```
results.instances.tsize
# Output
10
# Or results.instances.tnames, results.instances.tinstances...,
# that are beginning with 't'.
```

Or to print all instances both for training and testing:

```
results.instances.print_all()
# Output
Training instances
      instance_id  instance                seed
...
```

```

49 50          /path/to/2000-529.tsp 8803211
50 51          /path/to/2000-528.tsp 4980953

Test instances
  instance_id  instance              seed
  ...
8  9          /path/to/2000-728.tsp 850436
9  10         /path/to/2000-729.tsp 8784497

```

6. **configurations**: The target algorithm configurations generated by **crace**. This object includes a dictionary of ConfigurationEntry objects, and some other elements. See [Section 5.4.2](#). To obtain the values of the parameters of the elite configurations found by **crace** use:

```

for x in results.configurations.get_configurations(results.elites):
    print(x.param_values)
# Output
{'algorithm': 'acs', 'localsearch': '3', 'alpha': 4.4317, 'beta': 1.2857, 'rho': 0.5231,
↪ 'ants': 31, 'q0': 0.5771, 'rasrank': None, 'elitistants': None, 'nnls': 20, 'dlb':
↪ '1'}
{'algorithm': 'acs', 'localsearch': '3', 'alpha': 2.2388, 'beta': 6.5609, 'rho': 0.9484,
↪ 'ants': 68, 'q0': 0.8685, 'rasrank': None, 'elitistants': None, 'nnls': 8, 'dlb': '0'}

# Show attributes and corresponding values of parameters [parameter_name]
for x in vars(results.configurations.get_configuration(1)).items():
    print(x)
# Output of the ACOTSP scenario
('id', 1)
('parent_id', 0)
('model_id', 0)
('param_values', {'algorithm': 'as', 'localsearch': '0', 'alpha': 1.0, 'beta': 1.0, 'rho':
↪ 0.95, 'ants': 10, 'nnls': None, 'dlb': None, 'q0': None, 'rasrank': None,
↪ 'elitistants': None})
('alive', False)
('cmd', ' --as --localsearch 0 --alpha 1.0 --beta 1.0 --rho 0.95 --ants 10')

# Or show selected attributes for all parameters
for _, v in results.configurations.all_configurations.items():
    print(v.id, v.parent_id, v.cmd)
# Output of the ACOTSP scenario
1 0 --as --localsearch 0 --alpha 1.0 --beta 1.0 --rho 0.95 --ants 10
2 1 --mmas --localsearch 0 --alpha 3.4043 --beta 7.792 --rho 0.7502 --ants 50
3 1 --as --localsearch 3 --alpha 2.3227 --beta 9.5564 --rho 0.3509 --ants 51 --nnls 38
↪ --dlb 0
...

```



parent_id = 0 means current configuration is provided by `configurationsFile`.

7. **options**: The **crace** options including scenario settings for the tuning and some other options. See [Section 6.1](#). At Python console, you can check the details of a particular option:

```

# Take modelUpdateByStep as an example
results.options.modelUpdateByStep.default
# Output
0
results.options.modelUpdateByStep.value
# Output
3

```



`modelUpdateByStep` is based on the number of parameters for the tuning provided by user, its default value 0 is not the real one calculated by **crace**.

8. experiments: A dataframe with ExperimentEntry object as rows. Column names correspond to the internal elements of the ExperimentEntry object. The results include experiments from both training and testing part from **crace**. Take training experiments as an example, `results.training` is a class includes the related information of training experiments. You can check all its experiments in dataframe using command:

```
results.training.data
```

and check the experiments of the particular configurations using:

```
exps = results.training.experiments
exps.get_experiments_by_configurations(results.configurations.get_alive_ids())
```

The output is in a dataframe as displayed in [Section ??](#).

```
# As an example, obtain the experiments of best configuration.
exps.get_experiments_by_configuration(results.best_id)
# Output
  experiment_id configuration_id instance_id ... quality    time    state ...
0  2353           110                1     ... 32775798.0 10.847117 finished ...
1  2354           110                2     ... 32893527.0 10.877225 finished ...
2  2355           110                3     ... 32634253.0 10.851721 finished ...
...

# Check mean performance of elite configurations
df = exps.get_experiments_by_configurations(results.elites).dropna()
df.groupby('configuration_id')['quality'].mean()
# Output
configuration_id
104    32775500.3
110    32766384.6
Name: quality, dtype: float64
```

The information of experiments for a particular instance can be obtained using:

```
# get the number of finished experiments by instance
exps.get_ncomplete_by_instance(50)
# Output
2
exps.get_ncomplete_by_instance(51)
# Output
0

exps.get_experiments_by_instance(50).dropna()
# Output: only elite configurations have finished the executions on instance 50
  experiment_id configuration_id instance_id ... quality    time    state ...
1  2686           104                50     ... 32676606.0 11.653946 finished ...
3  2688           110                50     ... 32689294.0 11.628259 finished ...
```

When a configuration was not executed on an instance, its value is NaN. A configuration may not be executed on an instance because: (1) it was discarded by the statistical test and not executed on subsequent instances, or (2) it was discarded for illegal result and not executed on subsequent instances, or (3) the race terminated before this instance was considered.

```
# Obtain all experiments
all_exps = exps.get_all_experiments()
# Obtain all incompletd experiments
all_exps[all_exps['quality'].isna()]
# Output
   experiment_id configuration_id ... quality time state      ... start_time end_time
...
44    45           9             ... NaN      NaN  discarded ... None      ...
...
2691 2692        104             ... NaN      NaN   pending ... None      None
...
```

The instance id and seed used for a particular experiment can be obtained with:

```
# As an example, we get instance id, seed and instance of the experiment
# of the best configuration on a particular instance.
df = exps.get_experiments_by_configuration(110)
line = df[df['instance_id'] == 1]
# cmd_line is part of the command line used to call the target algorithm,
# it includes configuration_id, instance_id, seed and instance.
line['cmd_line'].to_string(index=False).split(' ')
# Output
['110', '1', '9469788', '/path/to/2000-522.tsp']
```



If the path is not fully displayed, you could use command as following:

```
import pandas as pd
pd.set_option('display.max_colwidth', None)
```

To obtain the full command line of a particular experiment for executing the target algorithm:

```
exps.all_experiments[1].get_exec_line()
# Output
'1 1 9469788 /path/to/2000-522.tsp --as --localsearch 0 --alpha 1.0 --beta 1.0 --rho
↳ 0.95 --ants 10'
```

To check the similarity of configurations, the pairwise test can be done for the elite configurations that have the same instances:

```
results.training.pairwise_test()
```

also the Kendall's coefficient of concordance (W) and Spearman's rank correlation coefficient (rho) are provided:

```
results.training.concordance()
```

9. `models`: a dictionary of LocalModel objects used to sample new configurations. To check the details of parameter model, you can use commands as following:

```
results.models.models.keys()
# Output
dict_keys([2, 5, 6, 1, 0, 3, 4])

results.models.models[2].models['localsearch'].print()
# Output
model: localsearch, type: c, prob: [0.1875, 0.1875, 0.1875, 0.4375]
```

In this case, `localsearch` is a categorical parameter and it has a probability for each of its values.

```
results.models.alive_models[2].models['ants'].model_std
# Output
0.3585
```

To sample a new configuration:

```
# Select the best configuration as the parent
parent = results.configurations.get_configuration(results.best_id)
# sample a new configuration (ConfigurationEntry)
sampled = results.models.sample_configuration(parent)
# check parameters of the new configuration
sampled.param_values
# Output
{'algorithm': 'acs', 'localsearch': '3', 'alpha': 4.5462, 'beta': 1.0742, 'rho': 0.3395, \
 'ants': 46, 'q0': 0.4009, 'rasrank': None, 'elitistants': None, 'nnls': 21, 'dlb': '0'}
# check if the sampled configuration is totally a new one
results.configurations.add_from_model([sampled])
# Output if the sampled configuration is new
[116]
# Output if the sampled configuration already exists
[-2]
```



If there is no new configuration sampled after 100 tries, when `softRestart` is not `0`, `crace` will reset the models of selected elite configuration to initial version. Otherwise, an error occurs:



ERROR: No new configuration generated after 100 tries. Perhaps activate `softRestart`?

To check the state of `crace` when updating models, you can use command as following:

```
results.slice
# Output
.. used_budget best_so_far best_mean_so_far elites alive_model_ids end_time
0 .. 250 32 32902384.5 [32, 31] [0] 1729780089.3
1 .. 500 55 32875347.8 [55, 62] [0, 2] 1729780613.9
2 .. 750 72 32778634.8 [72, 104] [2] 1729781309.5
```

10. `state`: A string that contains the state of `crace`, the recovery (Section 9) is done using the information contained in this object. The probabilistic model of the last elite configurations can be found here by doing:

```
results.state
# Output
used budget: 1000
finished experiments: 1000
number of alive configurations: 6
number of scheduled experiments: 53
elite configurations: 110, 104
number of instances in stream: 51
```

11. `summarise`: A class that contains the details of the tuning procedure. You can check all information using the following command:

```
results.summarise.all()

# Get summary data from the logdir.
crs = results.summarise
# Get number of slices
```

```

crs.n_slice
# Get number of experiments (runs of target-runner) up to each slice
crs.model_step_update
# Get the mean value of the best configurations
crs.best_id
crs.best_mean

```

12. test: A TestResults object that contains the testing results, which is an Experiments object, and some methods used to analyze the results.

```

results.test.data.get_all_experiments()
# Output
  experiment_id configuration_id instance_id ... quality    time    state    ...
0 7            110                7         ... 32674572.0 10.935061 finished ...
1 1            110                1         ... 32846438.0 11.023709 finished ...
...

```

10.3 Simple analysis of results

The final configurations returned by **crace** are the elites of the total race procedure. They are reported in decreasing order of performance, that is, the best configuration is reported first.

If testing is performed, you can further analyze the resulting best configurations by performing statistical tests in Python:

```

# Load log files where testing at most 5 elite configurations
results = crace.run(['--read', '/path/to/new'])
test = results.test

test.experiments.get_all_experiments()
# Output
  experiment_id configuration_id instance_id ... quality    time    state    ...
...
8 19            110                10         ... 32186327.0 24.653562 finished ...
9 2            104                1         ... 32773590.0 24.670124 finished ...
...

# pairwise test
# default parameter test_name is wilcoxon
test.pairwise_test()
# Output
Doing pairwise test (selected: wilcoxon)..
Elite configurations: [110 104]
Test instances: 10
Comparison between configuration 110 and 104:
  statistic=21.0, p-value=0.556640625, adjusted p-value=0.556640625

```

The Kendall concordance coefficient (w) and the Spearman's ρ can be applied over data that has the characteristics of the data obtained in the testing, that is a full matrix where all configurations are executed in all instances. We can show if the configurations tested have a homogeneous performance on the used instances set. If evidence of a heterogeneous scenario is found we recommend to make some adjustments in the **crace** options as described in [Section 6.3](#).

```

test.concordance()
# Output
kendall.w: 0.24375 spearman.rho: -0.5125

```

Otherwise, a package named **cplot** is provided to do some more detailed analysis for the **crace** results. Here is the [user guide](#)⁴ for this package.

11 List of command-line and scenario options

Most **crace** options can be specified in the command line using a flag or in the **crace** scenario file using the option name (or setting their value in the scenario list passed to the various Python functions exported by the package). This section describes the various **crace** options that can be specified by the user in this way.



Relative filesystem paths (e.g., `./scenario/`) given in the command-line are relative to the current working directory (the directory at which **crace** is invoked). However, paths given in the scenario file are relative to the directory containing the scenario file. See also [Table 27](#).

11.1 Utility options

<code>.help</code>	<i>flag:</i> <code>-h</code> or <code>--help</code>	<i>default:</i> False
Show the list of command-line options of crace.		
1. brief information when call with <code>-h</code> and		
2. detail information when call with <code>-help</code> .		
3. more details when call <code>-man [option]</code> .		
<code>.man</code>	<i>flag:</i> <code>-m</code> or <code>--man</code>	<i>default:</i> False
Show details of the selected option. call with <code>python crace -man [option]</code> .		
<code>.version</code>	<i>flag:</i> <code>-v</code> or <code>--version</code>	<i>default:</i> False
Show the version of crace.		
1. brief information when call with <code>-v</code> and		
2. detail information when call with <code>-version</code> .		
<code>.doc</code>	<i>flag:</i> <code>doc</code>	<i>default:</i> False
Open the user guide of crace. This command must be the first argument.		
<code>.examples</code>	<i>flag:</i> <code>examples</code>	<i>default:</i> False
Open examples folder. This command must be the first argument.		
<code>.templates</code>	<i>flag:</i> <code>templates</code>	<i>default:</i> False
Open templates folder. This command must be the first argument.		
<code>.which</code>	<i>flag:</i> <code>which</code>	<i>default:</i> False
Open crace root folder. This command must be the first argument.		
<code>.where</code>	<i>flag:</i> <code>where</code>	<i>default:</i> False
Open crace root folder. This command must be the first argument.		
<code>.craceParallel</code>	<i>flag:</i> <code>parallel</code>	<i>default:</i> False

⁴User guide of craceplot, see <https://github.io/craceplot>

Execute parallel crace runs. This command must be the first argument.

NOTE: Arguments are separated by ‘--’: arguments before ‘--’ configure parallel crace runs, while arguments after ‘--’ are forwarded to crace.

Support arguments:

- h, -help: show this help message and exit.
- n, -repetitions: The number of parallel crace runs.
- p, -exec-dir: The directory path for saving the parallel crace results.
- t, -expname: The prefix of the folder name to save the parallel crace results.
- s, -seeds: A list of seeds used for parallel crace.

11.2 General options

<code>.check</code>	<i>flag:</i> -c or --check	<i>default:</i> False	
Check if the scenario and parameter definitions are correct and test the execution of the target algorithm 3 times on one training instance and one test instance (if provided), respectively. See Section 5.2 for more details.			
<code>.onlytest</code>	<i>flag:</i> -o or --only-test	<i>default:</i> False	
Run the configurations contained in the file provided (with option configurationsFile) as argument on the test instances. See Section 8 for more details.			
<code>.initialize</code>	<i>flag:</i> --init	<i>default:</i> False	
Initialize the tuning directory with template scenario files. With this option, two scenarios and some script files are copied to current folder.			
<code>.readlogs</code>	<i>flag:</i> --read	<i>default:</i> None	
Read data from log files of previous run and return a <code>craceResults</code> object, which includes objects <code>CraceOptions</code> , <code>Parameters</code> , <code>Configurations</code> , <code>Instances</code> , <code>Models</code> and <code>Experiments</code> .			
<code>debugLevel</code>	<i>flag:</i> -d or --debug-level	<i>default:</i> 0	<i>domain:</i> [0,5]
Level of information to display in the text output of crace. A value of 0 silences all debug messages. Higher values provide more verbose debug messages. See Section 10.1 for more details.			
<code>logLevel</code>	<i>flag:</i> -l or --log-level	<i>default:</i> 0	<i>domain:</i> [0,5]
Log level for generating log files. 0: only necessary files for the recovering and plots are included. 1: plus <code>execution_log</code> and <code>race_log</code> files. 2: plus <code>race</code> file. 4: plus <code>mpi</code> log files. 5: all files.			
<code>execDir</code>	<i>flag:</i> --exec-dir	<i>default:</i> ./	
Directory where the target algorithm executions will be performed. The default execution directory is the current directory. Notice: The execution directory must exist before executing crace, it will not be created automatically.			
<code>logDir</code>	<i>flag:</i> --log-dir	<i>default:</i> ./race_log	
Directory to save tuning results as plain text files. The provided path must be either an absolute path or relative to option <code>execDir</code> . See Section 10.2 for details on the format of the plain text files.			
<code>seed</code>	<i>flag:</i> --seed	<i>default:</i> None	<i>domain:</i> [1,9999999999]
Seed to initialize the random number generator. The seed must be a positive integer. If the seed is “” or NULL, a random seed will be generated.			

11.3 Scenario configurations

scenarioFile	flag: -s or --scenario	default: ./scenario.txt
File that contains the scenario setup and other crace options. All options listed in this section can be included in this file. See \$CRACE_HOME/templates for an example. Relative file-system paths specified in the scenario file are relative to the scenario file itself.		
parameterFile	flag: -p or --parameter-file	default: ./parameters.txt
File that contains the description of the parameters of the target algorithm. See Section 5.1.5 for more details.		
configurationsFile	flag: -i or --configurations-file	default: None
File containing a table of initial configurations. If empty or NULL, all initial configurations will be sampled randomly. The provided configurations must not violate the constraints described in option parameterFile and forbiddenFile . See Section 5.4 for more details.		
forbiddenFile	flag: -f or --forbidden-file	default: None
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 5.1.4 for more details.		
targetRunner	flag: -t or --target-runner	default: ./target-runner
Script file that evaluates a configuration of the target algorithm on a particular instance. See Section 5.2 for more details.		
targetRunnerLauncher	flag: --target-runner-launcher	default: None
An executable command line that call a script file using specific executable software for each configuration that executes the target algorithm on a particular instance, when targetRunner cannot be executed directly. See Section 5.2 for more details.		
targetRunnerRetries	flag: --target-runner-retries	default: 0 domain: [0,10]
Number of times to retry a call of the target algorithm (provided by option targetRunner) if failed.		

11.4 Instances options

trainInstancesDir	flag: --train-instances-dir	default: ./Instances
Directory where training instances are located; either absolute path or relative to current directory. If no trainInstancesFiles is provided, all the files in trainInstancesDir will be listed as instances. See Section 5.3 for more details.		
trainInstancesFile	flag: --train-instances-file	default: None
File that contains a list of instances and optionally additional parameters for them. If trainInstancesDir is provided, crace will search for the files in this folder. The list of instances in option trainInstancesFile is interpreted as file-system paths relative to option trainInstancesDir . When using an absolute path or instances that are not files, set option trainInstancesDir="". See Section 5.3 for more details.		
testInstancesDir	flag: --test-instances-dir	default: None
Directory where testing instances are located, either absolute or relative to the current directory. Instances provided for testing should be different from the training instances. See Section 8 .		
testInstancesFile	flag: --test-instances-file	default: None

File that contains a list of test instances and optionally additional parameters for them. The list of instances in option `testInstancesFile` is interpreted as file-system paths relative to option `testInstancesDir`. When using an absolute path or instances that are not files, set option `testInstancesDir=""`. Instances provided for testing should be different from the training instances. See [Section 8](#) for more details.

11.5 Recovery options

`recoveryDir` *flag: -r or --recovery-file* *default: None*
 Previously saved **crace** log file that should be used to recover the execution of **crace**; either absolute path or relative to the current directory. If empty or NULL, recovery is not performed. For more details about recovery, see [Section 9](#).

11.6 Budget options

`maxExperiments` *flag: -E or --max-experiments* *default: 0* *domain: [0,999999999]*
 The maximum number of runs (invocations of the target algorithm that is provided by option `targetRunner`) that will be performed. It determines the maximum budget of experiments for the tuning. See [Section 6.2](#) for more details.

`maxTime` *flag: -T or --max-time* *default: 0* *domain: [0,999999999]*
 The maximum total time in seconds for the runs of `targetRunner` that will be performed. When option `maxTime` is positive, the `targetRunner` must return the execution time as its second output. See [Section 6.2](#) for more details.

11.7 Elimination Options

`testType` *flag: --test-type* *default: None*
domain: ['f-test', 'ftest', 'friedman', 't-test', 'ttest', 't-test-bonferroni', 't-test-bon', 't-test-holm']
 Specifies the statistical test used for elimination:
 - 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.

We recommend to not use corrections for multiple comparisons because the test typically becomes too strict and the search stagnates. The default setting of parameter `testType` is F-test unless the option `capping` is enabled in which case, the default setting is defined as adaptive-dom. See [Section 6.4.1](#).

`domType` *flag: --dom-type* *default: None*
domain: ['dom', 'dominance', 'adaptive-dom', 'adaptive-dominance']
 Specifies the aggressive test used for elimination:
 dominance test is derived from capping,
 adaptive dominance test is derived from adaptive capping.

When `capping` is enabled, its default value is adaptive dominance test. See [Section 6.4.2](#).

`firstTest` *flag: --first-test* *default: 0*
 Specifies how many instances are evaluated before the first elimination test. The value of option `firstTest` must be a multiple of the value of option `eachTest`. When option `capping` is enabled, the default value is 1; while option `capping` is disabled, the default value is 5. See [Section 6.3](#).

eachTest	flag: --each-test	default: 1
Specifies how many instances are evaluated for each configuration between elimination tests. See Section 6.3 .		
testExperimentSize	flag: --test-experiment-size	default: 0
Specifies the number of finished target executions should be included in one elimination test. This option can decrease the frequency to call elimination test. The values 0 and 1 mean to call elimination test when one target execution is finished. The greater number, the lower frequency to call elimination test.		
confidence	flag: --confidence	default: 0.95
Confidence level for the elimination test.		
digits	flag: --digits	default: 4
Maximum number of decimal places that are significant for numerical (real) parameters.		

11.8 Parallelization options

mpi	flag: --mpi	default: False
Enable/disable the use of package mpi4py to execute the targetRunner in parallel using MPI protocol. When option mpi is enabled, the option parallel is the number of slave nodes. See Section 7 for more details.		
parallel	flag: --parallel	default: 0 domain: [0, 999999999]
Number of calls of the targetRunner to be executed in parallel in training phase. Values 0 or 1 mean no parallelization. See Section 7 for more details.		
testParallel	flag: --test-parallel	default: 0 domain: [0, 999999999]
Number of calls of the targetRunner to be executed in parallel in testing phase. Values 0 and 1 mean no parallelization and it must not be greater than parallel. See Section 7 for more details.		
bindCores	flag: -b or --bind-cores	default: False
Enable/disable the bindings of workers to corresponding cores when using parallel settings, especially on system allowed mpi tools. ONLY supported on linux.		

11.9 Internal crace options

nbConfigurations	flag: --num-configurations	default: 0
The minimum number of alive configurations that need to be tuned in the racing procedure. If the number of configurations alive in the race is not larger than this value, crace will sample new configuration. By default (when equal to 0), this value is calculated automatically as $\lfloor 2 + \log_2 N^{param} \rfloor$, where N^{param} is the number of non-fixed parameters to be tuned.		
maxNbElites	flag: --max-num-elites	default: 0
Maximum number of elite configurations selected in the racing procedure. By default (when equal to 0), this value will be set as 6.		
testNbElites	flag: --test-num-elites	default: None
Number of elite configurations returned by irace that will be tested if test instances are provided. For more information about the testing, see Section 8 .		
modelUpdateByStep	flag: -u or --model-update-by-step	default: 0

Number of times to update the models. The total budget is divided by this parameter and each small budget is used for a slice. By default (with 0), **crace** calculates this number as $S^{model} = 2 \times \lfloor 2 + \log_2 N^{param} \rfloor$, where N^{param} is the number of non-fixed parameters to be tuned. We recommend to use the default value.

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 5.3](#).

sampleInsStrict *flag: --sample-ins-strict* *default: False*
 Enable/disable the sampling of the training instances strictly. If the option **sampleInsStrict** is disabled, the challenge is discovered when there is no enough alive configuration. Otherwise, the challenge is discovered immediately.

sampleInsFrequency *flag: --sample-ins-frequency* *default: 1*
 The probability for sampling new instance when there is a challenge. The default value is 1, which means it will always sample a new instance when there is a challenge, which means there are statistical similar configurations when using statistical test and there is a new elite configuration wins the old with the most instances when using aggressive test.

instancePatience *flag: --instance-patience* *default: 0* *domain: [0, 5000]*
 The number of configurations that an instance can undergo without yielding any improvement. The default value is 0, which means there is no such a limitation for configurations.

softRestart *flag: --soft-restart* *default: 1* *domain: [-1, 100]*
 The number of times of the soft-restart strategy applied to avoid premature convergence of the probabilistic model. When there is no new configuration after 100 tries, the probabilistic model of alive configurations is soft restarted. The default value is -1, indicating an unlimited number of restart attempts and the racing procedure will be terminated when all budgets are exhausted. The soft-restart mechanism is explained in the irace paper [11].

11.10 Elitist options

elitist *flag: --elitist* *default: False*
 Enable/disable elitist irace. In the elitist **crace**, elite configurations are not discarded from the race until non-elite configurations have been executed on the same instances as the elite configurations. 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 discarding elite configurations, even if they show poor performance, until a new elite configuration wins the old on the same instances. The non-elitist **crace** can select **maxNbElites** configurations as the elite from the decreasing rank, where the selected configurations may have fewer instances.

globalModel *flag: --global-model* *default: False*
 Enable/disable global model. In the elitist version of crace, only global model is alive during the race procedure, which is used for sampling new configurations or updating the model. It can only be activated when elitist is enable.

11.11 Capping options

capping *flag: --capping* *default: False*
 Enable/disable the use of capping, a technique designed for minimizing the computation time of configurations. This option is only available when option **elitist** is activated. When enabling this option, crace provides an execution time bound to each target algorithm execution (see [Section 6.4.2](#)). The bound can be calculated using capping or adaptive capping. See [Section 6.5](#) for more details.

boundMax *flag: --bound-max* *default: 0*

Maximum execution bound for `targetRunner`. It must be specified when `capping` is enabled.

`boundMaxDEV` *flag*: `--bound-max-dev`

default: 0

Maximum execution bound for `targetRunner` especially for training phase. It could be specified when `capping` is enabled and you wish to use a different runtime bound from the test phase.

`boundDigits` *flag*: `--bound-digits`

default: 0

Precision used for calculating the execution time. It must be specified when `capping` is enabled.

12 FAQ (Frequently Asked Questions)

12.1 Is crace minimizing or maximizing the output of my algorithm?

By default, **crace** considers that the value returned by `targetRunner` should be **minimized**. In case of a maximization problem, one can simply multiply the value by -1 before returning it to **crace**.

12.2 Are experiments with crace reproducible?

Short answer: Yes, under some conditions.

Long answer: According to the terminology described by López-Ibáñez et al. [3], we define *repeatability* as “*exactly repeating the original experiment, generating precisely the same results*”. Following this definition, a run of **crace** is repeatable under the following conditions:

- Same version of **crace**.
- Same version of Python (different versions of Python may change the behavior of functions used by **crace**).
- The behavior of `targetRunner` is deterministic or exactly reproducible for the same instance, parameter configuration and random seed. Make sure that `targetRunner` uses the seed provided by **crace** to initialize all random number generators used. If the result of `targetRunner` depends on CPU-time, wall-clock time or system load in any way, then `targetRunner` is not reproducible and neither will be **crace**.
- Same random seed (`seed`) given to **crace**.
- Same scenario options (Section 11). Although some options should not affect reproducibility (e.g., `debugLevel`), maintaining a list of such options will be a huge effort, thus the safest assumption is that any change may break reproducibility.
- Same parameter space (Section 5.1), including types, domains, conditions and forbidden configurations. The order of the parameters may also affect reproducibility (the name of the parameters should not) because it affects the order in which random numbers are used.
- Same training instances provided and in the same order (Section 5.3). Even if the instances are sampled randomly (`sampleInstances`), a different initial order will produce a different sample even with the same random seed.
- Same initial configurations (Section 5.4), if any.

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

Every time this was reported, it was a difficult-to-reproduce bug, i.e., a [Heisenbug](#), in the program (target algorithm), not in **crace**. To detect such bugs, we recommend that you use, within [targetRunner](#), a memory debugger (e.g., [valgrind](#)) to run your program. For example, if your program is executed by [targetRunner](#) as:

```
${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. Memory debuggers will significantly slowdown your code, so use them only as a means to find what is wrong with your target algorithm. Once you have fixed the bugs, you should remove the use of valgrind.

12.4 crace seems to run forever without any progress, is this a bug?

Every time this problem was reported, the issue was in the target algorithm and not in **crace**. Some ideas for debugging this problem:

- Check that the target algorithm is really not running nor paused nor sleeping nor waiting for input-output.
- Use `debugLevel=3` to see how **crace** calls target-runner, run the same command outside **crace** and verify that it terminates.
- Add some output to your algorithm that reports at the very end the runtime and exit code. Verify that this output is printed when **crace** calls your algorithm.
- In target-runner, print something to a log file *after* calling your target algorithm. Verify that this output appears in the log file when **crace** is running.
- Set a maximum timeout when calling your target algorithm from target-runner (see [FAQ 12.5](#)).

12.5 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 Python. However, in GNU/Linux, it is easy to implement by using the `timeout` command⁵ in [targetRunner](#) when invoking your program.

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

mpi4py does not work the same way when called from within a package and when called from a script or interactively. When **crace** creates the slave nodes, the slaves will load a copy of **crace** automatically. If the slave nodes are on different machines, they must have **crace**

⁵<http://man7.org/linux/man-pages/man1/timeout.1.html>

installed. If **crace** is not installed system-wide, Python needs to be able to find **crace** on the slave nodes. This is usually done by exporting `$CRACE_HOME/bin/` to the system path or by executing **crace** using its full filesystem path, which is called using `$PYTHON_HOME/bin/crace` or `python3 $CRACE_HOME/scripts/main.py` or `python3 $CRACE_HOME/scripts/mpi.py`.

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 `PATH` or calling **crace** using the full filesystem path seem the valid ways to tell the slaves where to find **crace**.

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

1. Test that **crace** and **mpi4py** work. Run **crace** on a single machine (submit node), without calling `qsub`, `mpirun` or a similar wrapper around **crace** or Python.
2. Test loading **crace** on the slave nodes. However, jobs submitted by `qsub`/`mpirun` may load Python packages using a different mechanism from the way it happens if you log directly into the node (e.g., with `ssh`). Thus, you need to write a little Python program such as:

```
from mpi4py import MPI
import importlib.util

# initialize mpi
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

try:
    import crace
    crace_path = importlib.util.find_spec('crace').submodule_search_locations
    if rank == 0:
        print(f"Master {crace_path}")
    elif rank > 0:
        import importlib.util
        print(f"Worker {rank}: {crace_path}")
except ImportError:
    print(f"rank {rank} import crace failed.")

MPI.Finalize()
```

Submit this program to the cluster like you would submit **crace** (using `qsub`, `mpirun` or whatever program is used to submit jobs to the cluster).

3. In the script `bin/parallel-crace-mpi`, the function `crace.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. **mpi4py** normally creates log files; but **crace** suppresses those files unless `logLevel ≥ 4`.

Please contact us ([Section 13](#)) if you have further problems.

12.7 How are relative filesystem paths interpreted by **crace**?

The answer depends on where the path appears. Relative paths may appear as the argument of command-line options, as the value of options given in the scenario file, or within various scripts, functions or instance files. Table 27 summarizes how paths are translated from relative to absolute.

Table 27: Translation of relative to absolute filesystem paths.

Relative path appears as ...	...is relative to ...
a string within <code>trainInstancesFile</code>	<code>trainInstancesDir</code>
a string within <code>testInstancesFile</code>	<code>testInstancesDir</code>
code within <code>targetRunner</code> or <code>targetEvaluator</code>	<code>execDir</code>
<code>logDir</code>	<code>execDir</code>
the value of other options in the scenario file	the directory containing the scenario file
the value of other command-line options	invocation (working) directory of crace

12.8 My parameter space is small enough that crace could generate all possible configurations; however, crace generates repeated configurations and/or does not generate some of them. Is this a bug?

Typically, **crace** is applied to parameter spaces that are much larger than what can be explored within the budget given. In **crace**, there is a hash list to store all configurations, only a different configuration can be accepted as the new. If after 100 tries of sampling, no such a new configuration can be sampled, **crace** will restart the model of all alive configurations to the initial version in order to avoid premature convergence if option `softRestart` is not **0**. Thus, if the parameter space is actually very small, and the option `softRestart` is **0**, **crace** will be terminated. If you still want to use continuous racing, the recommended approach is to set a greater value of `firstTest` or `eachTest` or a smaller budget.⁶

12.9 On Windows and using `target-runner.py` (a Python file), I get the error “`target-runner.py` is not executable”

The issue is that `.py` files are not executable on their own and you need `python.exe` to read the `.py` file and execute it. Linux knows how to do this if the first line of the file is “`#!/usr/bin/python`”, however, Windows doesn’t know how to do it. In Windows you have 2 options:

- Create a `target-runner.bat` file that contains a line similar to :

```
\path\to\python.exe \path\to\target-runner.py %instance% %seed% \
%candidate_parameters% 1>%stdout% 2>%stderr%
```

- Or convert `target-runner.py` into an `.exe` file, for example, using `auto-py-to-exe`⁷, so that you do not need a `.bat` file.

13 Resources and contact information

More information about the package can be found on the **crace** webpage:

<https://race-autoconfig.github.io/crace/>

For questions and suggestions please contact the development team through the **crace** package Google group:

<https://github.com/race-autoconfig/crace/discussions>

or by report on the github:

⁶If you are interested in implementing this, please contact us!

⁷<https://pypi.org/project/auto-py-to-exe/>

<https://github.com/race-autoconfig/crace/issues>

or by sending an email to:

race.autoconfig@google.com

14 Acknowledgements

We would like to thank all the people that directly or indirectly have collaborated in the development and improvement of **crace**: [Leslie Pérez Cáceres](#), [Jonas Kuckling](#), [Pablo Contreras](#), [Yunshuang Xiao](#), [Thomas Stützle](#), and [Manuel López-Ibáñez](#).

Bibliography

- [1] M. Birattari. On the estimation of the expected performance of a metaheuristic on a class of instances. how many instances, how many runs? Technical Report TR/IRIDIA/2004-001, IRIDIA, Université Libre de Bruxelles, Belgium, 2004.
- [2] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, Oct. 2009. doi: [10.1613/jair.2861](https://doi.org/10.1613/jair.2861).
- [3] M. López-Ibáñez, J. Branke, and L. Paquete. Reproducibility in evolutionary computation. *ACM Transactions on Evolutionary Learning and Optimization*, 1(4):1–21, 2021. doi: [10.1145/3466624](https://doi.org/10.1145/3466624).
- [4] C. C. McGeoch. Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 24(2):195–212, 1992. doi: [10.1145/130844.130853](https://doi.org/10.1145/130844.130853).

Appendix A Installing Python

This section gives a quick Python installation guide that will work in most cases. The official instructions are available at <https://docs.python.org/3/using/index.html>

A.1 GNU/Linux

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

```
sudo apt-get install python3 python3-dev
```

Once Python is installed, you can use conda, pip or the source code to install the **crace** package (see [Section 3.2](#)).

A.2 macOS

For macOS 10.9 (Jaguar) up until 12.3 (Catalina) the operating system includes Python 2, which is no longer supported and is not a good choice for development. You should go to do the [downloads page](#) and download the installer.

For newer versions of macOS, Python is no longer included by default and you will have to download and install it. You can refer to the [Python documentation](#) for more details on the installation process and getting started.

You can install Python directly from a PyPI mirror.⁸ Alternatively, you can just brew the Python formula from the science tap (unfortunately it does not come already bottled so you need to have Xcode⁹ installed to compile it):

```
# install homebrew if you don't have it
/bin/bash -c \
  "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
# install python 3
brew tap homebrew/science
brew install python
```

Once Python is installed, you can use conda, pip or the source code to install the **crace** package (see [Section 3.2](#)).

A.3 Windows

You can install Python from the official website.¹⁰ We recommend that you install Python on a filesystem path without spaces, special characters or long names, such as C:\Python.



Note that Python 3.5 - 3.8 cannot be used on Windows XP or earlier and Python 3.9 - 3.13 cannot be used on Windows 7 or earlier.

Once Python is installed, you can use conda, pip or the source code to install the **crace** package (see [Section 3.2](#)).

⁸<https://pypi.org/>

⁹Xcode download webpage: <https://developer.apple.com/xcode/download/>

¹⁰<https://www.python.org/downloads/windows/>

Appendix B Using a virtual environment

This section gives a quick Python virtual environment usage guide that will work in most cases. The official instructions are available at the [official website](#).

1. Create a new virtual environment.

venv (for Python 3) allows you to manage separate package installations for different projects. It creates a “virtual” isolated Python installation. When you switch projects, you can create a new virtual environment which is isolated from other virtual environments. You benefit from the virtual environment since packages can be installed confidently and will not interfere with another project’s environment.

To create a virtual environment, go to your project’s directory and run the following command. This will create a new virtual environment in a local folder named `.venv`:

```
# Unix/macOS
python3 -m venv .venv
```

```
# Windows
py -m venv .venv
```

The second argument is the location to create the virtual environment. Generally, you can just create this in your project and call it `.venv`.

venv will create a virtual Python installation in the `.venv` folder.

2. Activate a virtual environment.

Before you can start installing or using packages in your virtual environment you’ll need to activate it. Activating a virtual environment will put the virtual environment-specific python and pip executables into your shell’s PATH.

```
# Unix/macOS
source .venv/bin/activate
```

```
# Windows
.venv\Scripts\Activate.ps1
```

To confirm the virtual environment is activated, check the location of your Python interpreter:

```
# Unix/macOS
which python
```

```
# Windows
where python
```

While the virtual environment is active, the above command will output a filepath that includes the `.venv` directory, by ending with the following:

```
# Unix/macOS
.venv/bin/python
```

```
# Windows
.venv\Scripts\python
```

While a virtual environment is activated, pip will install packages into that specific environment. This enables you to import and use packages in your Python application.

3. Deactivate a virtual environment.

If you want to switch projects or leave your virtual environment, deactivate the environment:

```
deactivate
```

4. Reactivate a virtual environment.

If you want to reactivate an existing virtual environment, follow the same instructions about activating a virtual environment. There's no need to create a new virtual environment.

When a virtual environment is created, there are two installed packages, `pip` and `setuptools`. Before installing new package, it's better to upgrade the both packages at first.

```
pip3 install --upgrade pip setuptools
```

Appendix C Attachments

This section gives the attachments mentioned in the user guide above.

Sample text output of crace.

```
# Arguments: ['--parallel', '7', '-u', '3', '--log-level', '5']
# Reading crace options..
# Default scenario file was found in current directory
# Reading scenario file /Volumes/ssd/ysxiao/tmp/acotsp/scenario.txt
#-----
# crace: An implementation in python of a continuous Racing
# Version: 0.1.0
# Copyright (C) 2020
# Leslie Perez Caceres      <leslie.perez@pucv.cl>
# Franco Ardiles
# Jonas Kuckling
# Pablo Contreras
# Yunshuang Xiao
# Thomas Stutzle
#
# This is free software, and you are welcome to redistribute it under certain
# conditions. See the GNU General Public License for details. There is NO
# WARRANTY; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#-----
# Scenario options:
#   seed: 9864380
#   debugLevel: 0
#   logLevel: 5
#   execDir: /Volumes/ssd/ysxiao/tmp/acotsp/arena
#   scenarioFile: /Volumes/ssd/ysxiao/tmp/acotsp/scenario.txt
#   parameterFile: /Volumes/ssd/ysxiao/tmp/acotsp/parameters.txt
#   forbiddenFile: /Volumes/ssd/ysxiao/tmp/acotsp/forbidden.txt
#   configurationsFile: /Volumes/ssd/ysxiao/tmp/acotsp/default.txt
#   train instancesDir: /Volumes/ssd/ysxiao/race/ instances/tsp/RUE. instances.2000
#   train instancesFile: /Volumes/ssd/ysxiao/race/ instances/tsp/ instances/list_10_52
#   sample instances: True
#   test instancesDir: /Volumes/ssd/ysxiao/race/ instances/tsp/RUE. instances.2000
#   test instancesFile: /Volumes/ssd/ysxiao/race/ instances/tsp/ instances/list_10_72
#   testNbElites: 1
#   testType: F-test
#   eachTest: 1
#   firstTest: 3
#   testExperimentSize: 0
#   confidence: 0.95
#   targetRunner: /Volumes/ssd/ysxiao/tmp/acotsp/target-runner
#   targetRunnerRetries: 0
#   maxExperiments: 1000
#   maxTime: 0
#   digits: 4
#   parallel: 7
#   mpi: False
#   nb configurations: 5
#   maxNbElites: 6
#   elitist: True
#   modelUpdateByStep: 3
#   sampleInsFrequency: 1
#   softRestart: True
#   capping: False
#   boundMax: 0
#   boundDigits: 0
#   boundAsTimeout: True
#-----
# Creating log with log_level 5, debug_level 0 50
# Running crace configuration procedure

# 2024-10-24 16:18:48 CEST: Slice 1 of 4
# expectSliceBugdet: 250
# nbInitial configurations: 5
# Markers:
#   x No test is performed.
#   - The test is performed and some configurations are discarded
#   . All alive configurations are elite and nothing is discarded
#   c configurations added to the race by sampling
#   i instances added to the race based on challenge from elimination test
#   e instances added to the race based on option expect instance
# Each task:
#   instances: the number of selected instances for current task (sampled instances)
#   Alive: alive configurations in test (in crace)
#   Best: best configuration in test (in crace)
```

	instances	Alive	Best	Mean best	Exp so far	W time	E queue	C queue	Elites
	3(3)	5(5)	2(2)	33561381	10	24	8	4	2
	3(3)	4(4)	4(4)	33176761	12	24	6	3	4
	(4)					34	12	5	4,6
	(6)					57	15	6	7,6
	4(6)	5(5)	4(7)	33070759	37	68	6	4	7
	(7)					68	17	6	7,6
	4(7)	5(5)	4(4)	33070759	40	69	12	5	4,7,6
	(8)					79	14	5	7,6
	(10)					102	25	5	6,7
	5(10)	4(4)	4(7)	33095154	62	115	15	4	7,6
	10(11)	3(3)	17(17)	33082784	89	198	16	3	17
	(12)					258	30	5	17,21
	15(15)	4(4)	17(17)	33050083	162	368	21	3	17
	15(15)	4(4)	31(31)	32926836	186	413	44	3	31
	(16)					449	46	5	32,31

2024-10-24 16:28:09 CEST: Update models 1 of 3

experimentsTestedSoFar: 250

budgetUsedSoFar: 250

remainingBudget: 750

nbNew configurations: 38

nbNew instances: 14

nbEliminated configurations: 38

Elite-so-far configuration: 32 mean value: 32902384.47059

Description of the elite-so-far configuration 32:

--mmas --localsearch 2 --alpha 4.3788 --beta 2.9139 --rho 0.1755 --ants 57 --nnls 17 --dlb 0

2024-10-24 16:28:09 CEST: Slice 2 of 4

expectSliceBugdet: 250

Markers:

- x No test is performed.
- The test is performed and some configurations are discarded
- . All alive configurations are elite and nothing is discarded
- c configurations added to the race by sampling
- i instances added to the race based on challenge from elimination test
- e instances added to the race based on option expectet instance

	instances	Alive	Best	Mean best	Exp so far	W time	E queue	C queue	Elites
	4(17)	4(4)	32(32)	32937333	250	560	36	4	32,31
	(18)					605	40	5	32,45
	(19)					616	38	5	32,31
	3(19)	3(3)	32(32)	32860513	283	618	3	3	32,31,45
	(20)					628	44	5	31,32
	19(20)	3(3)	48(48)	32886341	309	672	41	3	48
	20(20)	4(4)	53(53)	32842226	336	727	60	3	53
	(21)					773	64	5	53,55
	3(22)	4(4)	55(55)	32853174	370	795	40	3	55
	(23)					795	69	5	53,55
	(25)					863	50	5	55,58
	5(25)	4(4)	55(53)	32787821	409	864	28	4	53,55,58
	25(25)	4(4)	55(55)	32857855	411	874	51	3	55
	(26)					930	56	5	55,62

2024-10-24 16:36:53 CEST: Update models 2 of 3

experimentsTestedSoFar: 500

budgetUsedSoFar: 500

remainingBudget: 500

nbNew configurations: 29

nbNew instances: 11

nbEliminated configurations: 29

Elite-so-far configuration: 55 mean value: 32875347.77778

Description of the elite-so-far configuration 55:

--as --localsearch 3 --alpha 4.0458 --beta 3.5557 --rho 0.9149 --ants 58 --nnls 26 --dlb 1

2024-10-24 16:36:53 CEST: Slice 3 of 4

expectSliceBugdet: 250

Markers:

- x No test is performed.
- The test is performed and some configurations are discarded
- . All alive configurations are elite and nothing is discarded
- c configurations added to the race by sampling
- i instances added to the race based on challenge from elimination test
- e instances added to the race based on option expectet instance

	instances	Alive	Best	Mean best	Exp so far	W time	E queue	C queue	Elites
	27(28)	3(3)	72(72)	32766943	526	1131	31	3	72

```

|i|          (29)|          |          |          |          | 1779|    62|    5| 72,104
+-----+-----+-----+-----+-----+-----+-----+-----+
2024-10-24 16:48:29 CEST:  Update models 3 of 3
experimentsTestedSoFar:    750
budgetUsedSoFar:          750
remainingBudget:          250
nbNew configurations:      32
nbNew instances:           1
nbEliminated configurations: 32
Elite-so-far configuration: 72          mean value: 32778634.79310

Description of the elite-so-far configuration 72:
--acs --localsearch 3 --alpha 3.7369 --beta 4.713 --rho 0.8554 --ants 39 --q0 0.5868 --nnls 21 --dlb 1

# 2024-10-24 16:48:29 CEST: Slice 4 of 4
# expectSliceBudget: 250
# Markers:
x No test is performed.
- The test is performed and some configurations are discarded
. All alive configurations are elite and nothing is discarded
c configurations added to the race by sampling
i instances added to the race based on challenge from elimination test
e instances added to the race based on option expect instance
+-----+-----+-----+-----+-----+-----+-----+-----+
| | instances | Alive| Best| Mean best| Exp so far | W time | E queue | C queue | Elites
+-----+-----+-----+-----+-----+-----+-----+-----+
|-| 3(29)| 3(3)| 104(72)| 32732663| 754| 1788| 5| 3| 72,104
|i| (31)| | | | | 1941| 39| 5| 110,104
+-----+-----+-----+-----+-----+-----+-----+-----+
2024-10-24 16:56:07 CEST:  Update models 3 of 3
experimentsTestedSoFar:    1000
budgetUsedSoFar:          1000
remainingBudget:           0
nbNew configurations:      11
nbNew instances:           22
nbEliminated configurations: 10
Elite-so-far configuration: 110          mean value: 32766384.6

Description of the elite-so-far configuration 110:
--acs --localsearch 3 --alpha 4.4317 --beta 1.2857 --rho 0.5231 --ants 31 --q0 0.5771 --nnls 20 --dlb 1

# BUDGET IS EXHAUSTED (maxExperiments)!
# nbUsed instances:          51
# nbUsed configurations:    115
# nbEliminated configurations (elimination test): 109
# mean value of the final best configuration 110 on 51 instances: 32766384.6

# Description of the final best configuration 110:
--acs --localsearch 3 --alpha 4.4317 --beta 1.2857 --rho 0.5231 --ants 31 --q0 0.5771 --nnls 20 --dlb 1

Before closing
Pool closed
Master stopped
Shutdown complete

```

Sample experiments of crace.

experiment_id	configuration_id	instance_id	budget	bound_max	cmd_line	quality	time	state	creation_time	start_time	end_time
0	1042	100	1	0	100 1 1243305 / ...	32632061.0	6.958269	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
1	1043	100	2	0	0 100 2 803830 / ...	32873645.0	7.094306	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
2	1044	100	3	0	0 100 3 8963029 / ...	32403294.0	7.065587	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
3	1045	100	4	0	0 100 4 931333 / ...	32637974.0	7.121079	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
4	1046	100	5	0	0 100 5 4191029 / ...	32933886.0	6.528276	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
5	1047	100	6	0	0 100 6 2938541 / ...	32857634.0	7.332312	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
6	1048	100	7	0	0 100 7 9558676 / ...	32670393.0	7.151243	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
7	1049	100	8	0	0 100 8 7208262 / ...	32974948.0	14.397675	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
8	1050	100	9	0	0 100 9 9145846 / ...	32989126.0	13.079033	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
9	1051	100	10	0	0 100 10 4273262 / ...	32794547.0	6.647842	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
10	1052	100	11	0	0 100 11 2717779 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
11	1053	100	12	0	0 100 12 8520197 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
12	1054	100	13	0	0 100 13 3563410 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
13	1055	100	14	0	0 100 14 4902126 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
14	1056	100	15	0	0 100 15 498321 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
15	1057	100	16	0	0 100 16 2275198 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
16	1058	100	17	0	0 100 17 320431 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
17	1059	100	18	0	0 100 18 5745781 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
18	1060	100	19	0	0 100 19 7199291 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
19	1061	100	20	0	0 100 20 1821593 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
20	1062	100	21	0	0 100 21 5940458 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
21	1063	100	22	0	0 100 22 4686866 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
22	1064	100	23	0	0 100 23 4714892 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
23	1065	100	24	0	0 100 24 1008629 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
24	1066	100	25	0	0 100 25 3378366 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
25	1067	100	26	0	0 100 26 1217027 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
26	1068	100	27	0	0 100 27 973883 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
27	1100	100	28	0	0 100 28 2180667 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
28	1133	100	29	0	0 100 29 8227644 / ...	32520566.0	6.929185	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
29	1225	100	30	0	0 100 30 1959562 / ...	32862803.0	7.483648	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
30	1231	100	31	0	0 100 31 550032 / ...	33010842.0	12.416237	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
31	1299	100	32	0	0 100 32 5346150 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
32	1165	104	1	0	0 104 1 1243305 / ...	33549688.0	7.177655	finished	(2024, 10, 23, ...)	(2024, 10, 23, ...)	(2024, 10, 23, ...)
33	1166	104	2	0	0 104 2 803830 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
34	1167	104	3	0	0 104 3 8963029 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
35	1168	104	4	0	0 104 4 931333 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
36	1169	104	5	0	0 104 5 4191029 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
37	1170	104	6	0	0 104 6 2938541 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
38	1171	104	7	0	0 104 7 9558676 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
39	1172	104	8	0	0 104 8 7208262 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
40	1173	104	9	0	0 104 9 9145846 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
41	1174	104	10	0	0 104 10 4273262 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
42	1175	104	11	0	0 104 11 2717779 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
43	1176	104	12	0	0 104 12 8520197 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
44	1177	104	13	0	0 104 13 3563410 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None
45	1178	104	14	0	0 104 14 4902126 / ...	NaN	NaN	pending	(2024, 10, 23, ...)	None	None

Appendix D Changelog

```
# Changelog

## crace 1.0

### crace 1.1.0

> Based on tag: v1.0.1
>
> - src:
>   - options: add integer option testParallel
>     - value 0: disable parallelization in testing phase
>     - value > 1: enable parallelization in testing phase (must be <= parallel)
>
>   - main.py
>     - add `print_affinity_info()` when parallel > 1 (mpi disabled)
>
> - pkg:
>   - add changelog.md
>
> (Yunshuang Xiao)

### crace 1.0.1

> Based on tag: v1.0.0.post1
>
> - src:
>   - main.py, mpi.py, utils.py:
>     - improve error handling in `start_cmdline()`, `start_mpi()` and
>     - `_enforce_single_thread_binding()`
>     - support single cpu binding in `_enforce_single_thread_binding()`
>
> (Yunshuang Xiao)

### crace 1.0.0.post1

> Based on commit 0686d5
>
> - src:
>   - Experiments, ProbabilisticModel:
>     - add 'debug_level' as argument to initialize mentioned class
>
>   - scripts.mpi:
>     - add try-except for mpi execution to ensure exiting
>     - add `_check_mpi_env()` to check mpi environment
>       - installed mpi4py or not
>       - installed MPI implementation or not
>       - mpi.py called by MPI launcher or not
>     - improve the output of cpu affinity for mpi executions
>
> (Yunshuang Xiao)

### No tag (**commit**: 0686d51)

> modified gh-pages.yml
>
> (Yunshuang Xiao)

### crace1.0.0.post0

> Based on v1.0.0
>
> - src:
>   - execution: improve pool management and add support for different python versions
>
>   - race:
>     - bug fix for log_state: support for 'n_instances'
>     - bug fix for slice budget: min -> max
>
>
```

```
> - package:
>   - makefile: support pyproject, build, tag
>     - pyproject: generate pyproject.toml from generate_pyproject.py
>     - build: build package
>     - tag: add tag for the current version based on (provided) base tag
>
>   - improve generate_pyproject.py
>     - add urls
>     - improve classifiers
>
>   - delete setup.py
>
>   - base information:
>     - modify package description
>
> (Yunshuang Xiao)
```

crace 1.0.0

```
> The first public version: v1.0.0
>
> (Leslie Pérez Cáceres, Jonas Kuckling, Pablo Contreras and Yunshuang Xiao)
```