Metrics: Towards a Unified Library for Experimenting Solvers

Thibault Falque, Romain Wallon, Hugues Wattez

Metrics: Towards a Unified Library for Experimenting Solvers

Thibault Falque[1], Romain Wallon[2], and Hugues Wattez[2]

[1] Exakis Nelite
thibault.falque@exakis-nelite.com
[2] CRIL, Univ Artois & CNRS
{wallon,wattez}@cril.fr

Abstract

When developing a SAT solver, one of the most important parts is to perform experiments so as to evaluate its performance. Most of the time, this process remains the same, so that everybody collects almost the same statistics about the solver execution. However, how many scripts are there to retrieve experimental data and draw scatter or cactus plots? Probably as many as researchers in the domain. Based on this observation, this paper introduces Metrics, a Python library, aiming to unify and make easier the analysis of solver experiments. The ambition of Metrics is to provide a complete toolchain from the execution of the solver to the analysis of its performance. In particular, this library simplifies the retrieval of experimental data from many different inputs (including the solver's output), and provides a nice interface for drawing commonly used plots, computing statistics about the execution of the solver, and effortlessly organizing them (e.g., in Jupyter notebooks). In the end, the main purpose of Metrics is to favor the sharing and reproducibility of experimental results and their analysis.

1 Introduction

In the context of computer science research (and more generally, in any field requiring the design of software programs), it is necessary to carry out experiments to ensure that the produced programs work as intended. In particular, one needs to ensure that the resources it uses remain reasonable. To do so, software solutions such as *runsolver* [8] have been developed to measure and limit the consumption of the temporal and spatial resources of the program under test. However, respecting these limits is generally not sufficient to assess the program behaviour. It is often necessary to collect additional statistics, which are usually provided by the program (e.g., via software logs) or by tools such as *runsolver*. The collected data has then to be aggregated to evaluate the quality of the results of the program through a statistical analysis.

Statistics provide many mathematical tools, and choosing one over another can introduce biases in the results or their analysis. Thus, over the years, many cases of erroneous analyses have been identified. One of the most famous such analysis is an article on economics by Reinhart and Rogoff [7], for which analytical errors were detected only three years later [4]. As a countermeasure to such errors, the principles of *transparent science* and *reproducibile results* are increasingly being applied. They have been the subject of an OECD recommendation [6], and scientists have introduced many approaches favoring reproducibility in the context of computer experiments [3,5]. Towards this direction, it is recommended to open the source of the software program (or, at least, provide its binaries), and to make available the data used to evaluate it (e.g., using software forges). It is also important to make the analysis of the results reproducible, which can be done using tools such as RMarkdown or Jupyter notebooks.

In the SAT research community (as well as in the CP, PB, QBF communities, and many more), there is not a great difference between how the different solvers are executed. Indeed,

solvers are often required to provide command line interfaces that meet the requirements of the environment in which they are being executed (e.g., during competitions). As such, the main difference between these programs are their actual implementation. Also, it appears that most of the data collected when running the solvers remains almost the same (e.g., runtime, memory usage, etc.). In this context, the creation of a tool that is able to run the program, collect the data it produces and analyze it would have multiple advantages: testing new features is simpler, both in terms of execution and analysis, and the reproducibility of the results is automatically ensured.

Based on these observations, this paper introduces *Metrics* (*mETRICS* stands for *rEproducible sofTware peRformance analysIs in perfeCt Simplicity*), a (work-in-progress) Python library, aiming to unify and make easier the analysis of solver experiments. The ambition of Metrics is to provide a complete toolchain from the execution of the solver to the analysis of its performance. Currently, this library contains two main components: `scalpel` (*sCAlPEL* stands for *extraCting dAta of exPeriments from softwarE Logs*) and `wallet` (*wALLET* stands for *Automated tooL for expLoiting Experimental resulTs*). On the one hand, `scalpel`, is designed to simplify the retrieval of experimental data. It is able to handle a wide variety of inputs, including CSV files or even the solver's output thanks to a description file provided by the user. This makes the tool easy to configure and highly flexible. On the other hand, `wallet` provides a nice interface for drawing commonly used plots (such as scatter or cactus plots) and computing statistics about the execution of the different solvers (in particular, their score using classical performance measures). The design of `wallet` makes easier the integration of the analysis in Jupyter notebooks which can easily be shared online (for instance, GitHub is able to render such files), which also favors the reproducibility of the analysis.

The rest of this paper is organized as follows. As preliminaries, we introduce different vocabulary notions that we use throughout the paper. We then present the different plots and statistics classically used when analyzing solvers, and the tools that are generally used to generate them. Next, we present an overview of the *Metrics* library and of its components before considering a use case of *Metrics* on the results of the last SAT competition. We finally conclude with some perspectives of improvement for *Metrics*, which would allow to make it a unified library for experimenting solvers.

2 Preliminaries

In this section, we introduce some of the vocabulary used when talking about *Metrics*. We also describe the main figures and statistics used to compare the performance of different solvers, and make a tour of the existing libraries allowing to compute them.

2.1 Description

First, let us define the vocabulary used in this paper. The main object handled by *Metrics* is a *campaign*, which contains all the experimental data that has been collected, and defines the experimental setup (time and memory limit, computer configuration, etc.). During a campaign, *experimentwares* are being experimented. We use experimentware as a generic name to characterize software with the ability of being experimented (as a more general notion than *solvers*). Different experimentwares may be either different programs or the same program with different configurations or settings. The campaign is also characterized by the *input-sets* it uses, i.e., the *inputs* that are given to the different experimentwares. Note that, in this context, all experimentwares are given exactly the same input-set. Finally, an *experiment* depicts the execution

of a particular experimentware on a particular input, so that the set of experiments corresponds to the cartesian product of the input and experimentware sets. Experiments are characterized by the data that are relevant for our analyses, such as the runtime and memory usage of the experimentware (and many other statistics, depending on the user configuration).

Example 1. *Let us consider a* campaign *in which we would like to compare the two solvers* Sat4j *[2] and* Glucose *[1]. These solvers are our* experimentwares. *Suppose that we want to compare them on two* inputs*: a sudoku instance* `sudoku.cnf` *and a pigeonhole problem* `pigeonhole.cnf`*. The* input-set *we consider is composed of these two instances. The* experiments *of this campaign are thus:*

- *the execution of* Sat4j *on* `sudoku.cnf`*,*
- *the execution of* Sat4j *on* `pigeonhole.cnf`*,*
- *the execution of* Glucose *on* `sudoku.cnf`*, and*
- *the execution of* Glucose *on* `pigeonhole.cnf`*.*

2.2 Statistics

After executing such a campaign, we need tools to analyze the collected results. Even though a large variety of analyses exists, most of the time, the prefered way of comparing solvers is to consider the number of solved inputs and the time taken to solve them, as shown in this section.

First, let us describe the relevant statistics for *Metrics*. As we said above, one of the most important and trivial ways of comparing experimentwares is to count the number of solved inputs. This allows to have a quick overview of the behaviour of each experimentware. That is not the only way to compare their efficiency, though. Indeed, an experimentware could solve the highest number of inputs while having taken a time close to the timeout to solve each of them. These experimentwares could thus beat another solver that solves the same inputs twice as fast, except one that is not solved at all (which we admit is quite rare in practice). That is why counting the number solved inputs is not a sufficient measure to assess the performance of a solver, and we need to consider another criterion: time.

In this case, there exist also many ways to compare the runtime of the different solvers. One of them is the PARx method, computed with the cumulated sum of the runtime of the solver on each solved input, to which the runtime of unsolved inputs (equal to the timeout) is added with a penalty (by being multiplied by x). Common PARxs are PAR1, PAR2 or PAR10, which correspond to the original timeout, the timeout multiplied by 2 or by 10, respectively.

However, such measures may be considered as *arbitrary*: the runtime of different solvers that do not solve a given input will be set to the same value for this instance (the timeout with a penalty), whereas their actual runtime would probably be different (if they had enough time to solve it). We thus also consider another measure, called *common solved inputs*. It consists in avoiding to take into account instances that were not solved by at least one solver or, otherwise said, to only consider instances solved by all solvers. The score of each solver is computed as with PARx methods on this subset of inputs, with the difference that, as all solvers solved these instances, there is no need for penalties. However, the main drawback of this measure is that the more experimentwares, the less inputs in the considered subset. As a symmetrical measure, we also consider the *uncommon solved inputs*. Such inputs are solved by at least one experimentware but unsolved by at least another one. This set allows to compare the different solvers on the inputs considered as "hard" (as opposed to the "easy" common solved inputs).

In all cases, considering only one measure is often not enough, and it is preferable to consider a combination of them instead.

2.3 Plots

In general, figures such as scatter or cactus plots are a good way to visualize statistics and to get a preview of solvers behaviour. In the following we present plots that may be used to display a wide variety of statistics, and especially the two main measures described above: the runtime and the number of solved inputs.

A first kind of plots that allows to consider an overview of all the experimentwares is the *cactus plot* (see, e.g., Figures 3 and 4). A cactus plot considers all solved inputs of each experimentware. Each line in the plot represent an experimentware. Inputs are ordered by solving time for each experimentware to build this figure: the x-axis corresponds to the rank of the solved input and the y-axis to the time taken to solve the input, so that the righter the line, the better the solver. Note that we can also cumulate the runtime of each solved inputs to get a smoother plot.

In addition to cactus plots, one may consider *box plots* (see, e.g., Figure 6) to get more detailed results about the runtime of each solver. A box in such a plot represents the distribution of each experiment time of a given experimentware. In particular, such plots allow to easily locate medians, quartiles and means for all experimentwares in a single figure. We can find a practical application of this plot in the case of randomized algorithms: it permits to visualize the variance and to simply compare the effect of changing the random function seed for a given fixed solver configuration using it.

Finally, to get a more detailed comparison of two experimentwares, one can use scatter plots (see, e.g., Figure 5). Each axis in this plot corresponds to an experimentware and displays its runtime (between 0 and the timeout). We can place each input in the plot as a point corresponding to the time taken by both experimentwares to solve this input. We can quickly observe if there exists a trend for one experimentware or the other in terms of efficiency.

Once again, it is important to note that a combination of all plots is often required before actually making conclusions.

2.4 Technologies

Many technologies have been proposed to implement the statistics and plots described above. We focus here on the most popular ones.

Created in 1986, *GNUPlot* is one of the oldest plot libraries which is still used today by the community. It has a great success among computer scientists, as it is a full command-line driven graphing utility. In our case, and for the many perspectives we give to *Metrics*, this library does not fit our needs, as we need a complete programming language allowing to do more than just plotting figures, for instance to parse the log data produced by the experimentware during its execution.

Another well-known technology is the statistic-oriented R language. This language not only allows to build all the possible figures and statistics we need, it also natively supports the reproducibility of experimental analyses thanks to RMarkdown, which allows to combine both textual format (Markdown) and code (R, Python, etc.). However, we believe that the use of a language that is not so widespread in the community will make the adoption of *Metrics* harder.

From these observations we decided to use Python. Indeed, the Python community has implemented a library that imitates the R standard library named `pandas`. This library provides R-like *data-frames*, which are a particular data structure allowing to simply filter and manage

data extracted from the considered experiments. Python also provides a library to create dynamic plots (`plotly`, or more classically `matplotlib`). All these tools may be integrated in *Jupyter notebooks*, the Python counterpart of RMarkdown. We also note that some tools have also been developed by the community, such as *mkplot*[1], but do not provide a complete toolchain as *Metrics* is intended to. Finally, Python is a well-known language, and its simple syntax will make easier the use of our library by end users.

3 Presentation of the Library

This section presents a quick overview of the design of *Metrics*.

3.1 *Metrics* at a Glance

The component diagram presented in Figure 1 shows an overview of *Metrics*. This library exposes two main components, `metrics-scalpel` and `metrics-wallet`, which rely on the same `metrics-core` component (which is not intended to be used by end users).

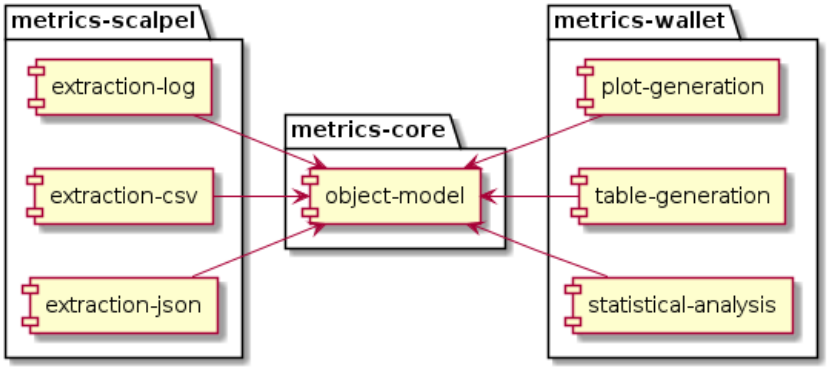

Figure 1: Component diagram of *Metrics*

The `metrics-core` component contains the definition of *Metrics*' object model. Figure 2 provides a description of how this model is designed, which is an object-oriented implementation of the vocabulary defined in Section 2.1.

The two other components, namely `metrics-scalpel` and `metrics-wallet` are independent. It is thus possible to use `metrics-scalpel` without using `metrics-wallet` and vice-versa. But it is not possible to use `metrics-scalpel` nor `metrics-wallet` without `metrics-core`, as it provides the internal representation of the campaign used by these components. Note that `metrics-scalpel` may be used to transform data to fit the `metrics-core` representation.

[1] https://github.com/alexeyignatiev/mkplot.git

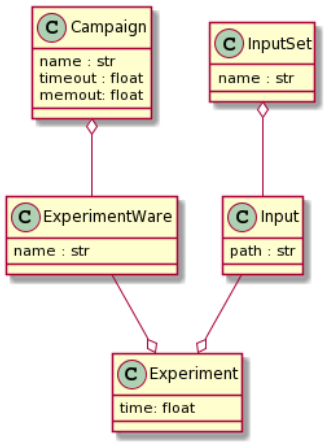

Figure 2: Class diagram of `metrics-core`

3.2 A Closer Look on `metrics-scalpel`

sCAlPEL ("extraCt dAta of exPeriments from softwarE Logs") allows to extract the data collected during a campaign into the internal representation defined by `metrics-core`.

These results may have already been collected within a CSV file (or any *SV file, actually), and will then be interpreted by *Scalpel* as a `Campaign` object. Results can also be extracted "as is" from the logs produced by the solver or by the environment in which it has been executed. In particular, `metrics-scalpel` provides a simplified way of describing how to extract relevant variables from such files. Finally, `metrics-scalpel` is designed to read JSON representations of the campaign to consider (for now, only JSON files that match exactly the representation defined by `metrics-core` are supported).

3.3 A Closer Look on `metrics-wallet`

wALLET ("Automated tooL for expLoiting Experimental resulTs") provides tools allowing to compute the statistics and draw the plots described in Sections 2.2 and 2.3. Under the hood, `metrics-wallet` converts a `Campaign` (as built by `metrics-scalpel`, for instance) into a data-frame of the `pandas` library (which thus also allows to use the classical `pandas` and `matplotlib` methods on this representation).

`metrics-wallet` is divided into two modules. The first one allows to draw static plots (as those described previously) and to compute tables showing different statistic measures. These figures can easily be exported in a format specified by the user, such as HTML or LaTeX for tables, or PNG images and vector graphics (such as SVG or EPS images). Static plots are highly configurable in order to fit in their final destination. This is why it is possible with these specific plots to set the font (family, size and color), using LaTeX commands in the different titles, mapping specific colors or shapes to experimentwares, customizing legend format, etc. This module is mainly based on the `matplotlib` library. The second module allows to build dynamic figures, using the `plotly` library. It makes possible to the user to interact with the plot, for example, by zooming in or out in the plots, selecting subparts of the legend, displaying additional data thanks to mouse hovering, etc. Users may also customize their plots through the interface given by Jupyter notebooks and with the parameters given to the plots at their creation.

4 Metrics in Action

To introduce the capabilities of *Metrics*, let us consider as an example the analysis of the results of the last SAT competition[2]. In this competition, 51 experimentwares (a.k.a. solvers) were experimented on 400 instances. Each experiment (i.e., execution of a particular solver on a particular instance) was set a timeout of 5000 seconds and the memory usage was limited to 128 GB.

As the competition is stiff between solvers, a tool as *Metrics* is welcome to observe what happened globally in the competition and more precisely between the best solvers of the event. If you are interested in reproducing the analysis proposed in this paper, you may find *Metrics* on GitHub[3].

4.1 Extracting Data with `metrics-scalpel`

In order to retrieve experimental data with `metrics-scalpel`, a YAML configuration file may be used to describe how to extract them from different files. First, this file may declare metadata about the campaign being analyzed, especially regarding the experimental setup.

```yaml
name: SAT Race 2019
date: July 12th, 2019
setup:
  timeout: 5000
  memout: 128000
```

This file also contains the informations about the experimentwares executed in the campaign (not all of them are listed in the example below for space reason). This is quite a strong requirement (and we plan to automatically discover the solvers in a future version of *Metrics*), but this approach has been designed to allow, when needed, to specify more informations about the solvers (such as their version, their command line, their fingerprint etc.).

```yaml
experiment-wares:
  - CCAnrSim default
  - ...
  - smallsat default
```

Similarly, the list of considered inputs must be specified in the YAML configuration. In the following example, the inputs are retrieved from a `hierarchy`. More precisely, `metrics-scalpel` explores the file hierarchy rooted at the given directory to discover each file it contains. It is also possible to give directly the list of the files, or to give a path to a file that contains this list. Once again, future versions of *Metrics* will be able to discover this list automatically.

```yaml
input-set:
  name: sat-race-2019
  type: hierarchy
  path-list:
    - /path/to/the/benchmarks/of/sat/race/2019/
```

To specify in which file *Scalpel* must look to retrieve the data, its `path` is set as follows.

[2] http://sat-race-2019.ciirc.cvut.cz
[3] https://github.com/crillab/metrics

```yaml
source:
  path: /path/to/the/results/of/sat-2019.csv
```

Now comes what is probably the most important section of the configuration file, which actually describes *how* to retrieve the data from the source file. In our example, as this file is a CSV file, there is not many thing to declare in the `data` field. The only relevant information to provide here is in the `mapping` section, which allows to retrieve from the CSV file (in this case) which columns correspond to the data expected by *Scalpel*. Note that the fields in this example are required as the column names do not match the naming convention of *Metrics*, and also that the identification of the experimentwares is spanned on two columns (the name of the solver and its configuration), which explains why a list is required there.

```yaml
data:
  mapping:
    input: benchmark
    experiment_ware:
        - solver
        - configuration
    cpu_time: solver time
```

Let us now make an important side note here. In many cases, the experimental data has to be retrieved from the *log files* produced by the solver. To do so with *Scalpel*, all log files must be put in a single directory (if there is exactly one file per experiment), or in different directories (with one directory per experiment, which may itself contain multiple regular files). In both cases, the path specified as `source` is the directory at the root of all experiment files, and *Scalpel* will explore the entire hierarchy to retrieve all the relevant data. If some files of the hierarchy are raw log files, *Scalpel* will consider the description of the data they contain, which must also be set in the `data` section of the configuration file. The following example illustrate two different ways for describing the data to extract: either with a regular expression or with a simplified pattern. An interesting advantage of our simplified patterns is that they support many common expressions (`integer`, `real`, `word` or `any`) and that white spaces are interpreted as "any number of white spaces" (including tabulations). When using classical regular expressions, the data to extract must be properly identified with a group.

```yaml
data:
  raw-data:
    - log-data: memory
      file: mysolver.log
      regex: "c Memory usage: (\d+.\d+) Mo"
    - log-data: cpu_time
      file: mysolver.log
      pattern: "c CPU time: {real} seconds"
```

In all cases, it is the user's responsibility to ensure that all the data needed to perform the analysis may be retrieved by *Scalpel*, either with the corresponding `log-data` or `mapping`.

Once the YAML configuration file is properly set up, we can load the whole campaign it describes, corresponding here to the SAT competition.

```
from metrics.scalpel import read_yaml
campaign = read_yaml("/path/to/configuration.yml")
```

4.2 Exploiting Data with `metrics-wallet`

Now that we have extracted the relevant data from our campaign, we can start building figures. The first step consists in extracting a data-frame from the read campaign.

```
from metrics.wallet.dataframe.builder import CampaignDataFrameBuilder
campaign_df = CampaignDataFrameBuilder(campaign).build_from_campaign()
```

A campaign data-frame is a *Metrics* object encapsulating a `pandas` data-frame and allowing to simply manage experiments and apply operations such as filtering. Filtering operations permit to select a subset of experimentwares, inputs, and more generally making a subset of any column provided by the data-frame. With this data-frame, we may now build all the figures provided by *Metrics*. First, let us make a global overview of the solvers submitted to the competition with a cactus plot. To do so, one just needs to create a `CactusPlotly` figure from the campaign dataframe and call the method `get_figure()`:

```
from metrics.wallet.figure.dynamic_figure import CactusPlotly
cactus = CactusPlotly(campaign_df, cactus_col="cpu_time")
cactus.get_figure()
```

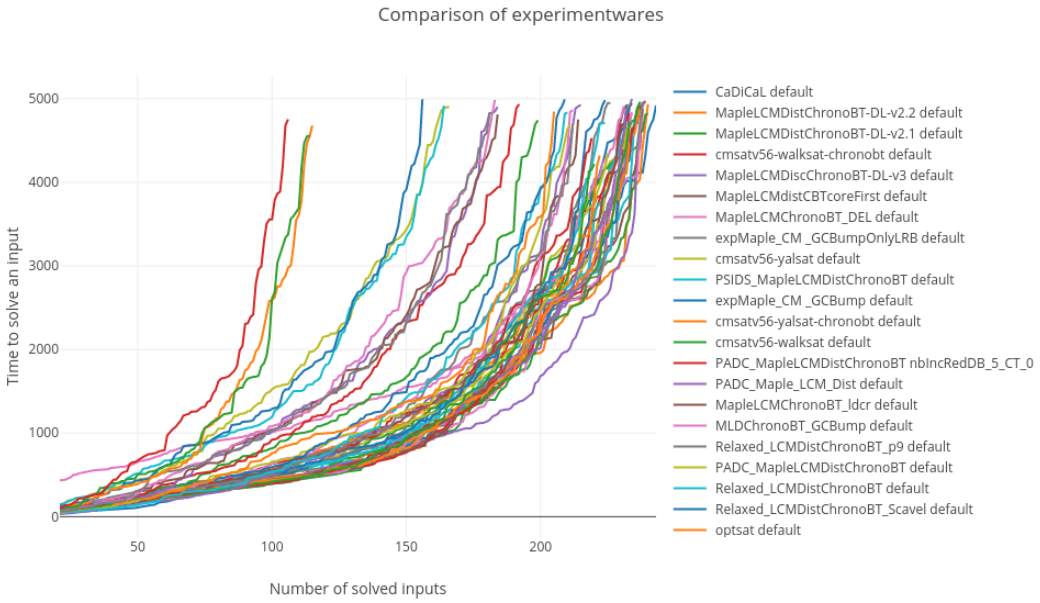

Figure 3: Cactus plot of the solvers submitted to the SAT Race 2019.

The cactus plot in Figure 3 is hard to interpret. Indeed, there are many plot lines with a lot of colors, but it is impossible to discriminate the solvers. Observe that solvers are ordered in the legend by decreasing number of solved instances to help reading the plot. Also, in this dynamic figure produced by *Metrics*, one may select solvers by clicking on their names in the

legend to highlight the corresponding line. Nevertheless, it remains hard to see something in the current view.

Using the same kind of plots, we want to apply a filter to compare only the best solvers. To do so, we create the subset of solvers we are interested in and give it to the filter method of the campaign data-frame while specifying on which column the filter must be applied. We get thus a new campaign data-frame as follows:

```
subset = {
    "CaDiCaL default",
    "MapleLCMDistChronoBT-DL-v2.2 default",
    "MapleLCMDistChronoBT-DL-v2.1 default",
    "MapleLCMDiscChronoBT-DL-v3 default",
    "cmsatv56-walksat-chronobt default"
}
campaign_df_best = campaign_df.sub_data_frame("experiment_ware", subset)
```

We may now, using the same process as described above, create a cactus plot displaying only the specified solvers. To do so, we use the new campaign data-frame to create it, and add two new parameters. The first one, `show_marker`, specifies that we want to show the markers for each solved instance of each solver. The second one, `min_solved_inputs`, specifies that we want to start showing plot lines after 200 solved instances:

```
cactus = CactusPlotly(
    campaign_df_best,
    show_marker=True,
    min_solved_inputs=200
  )
cactus.get_figure()
```

Figure 4 shows now clearer results. Indeed, we see now that *CaDiCaL* is leading the race in terms of number of solved instances. As said in Section 2.2, it is however also important to consider the runtime of the different solvers. We can also observe in this figure that *CaDiCaL* takes more time to solve the inputs ranked between the 200th and 235th instances compared to *MapleLCMDistChronoBT-DL* solvers. That is why it is interesting to now consider the runtime to compare these solvers.

For this time analysis, let us introduce another capability of *Metrics*: computing *Virtual Best Solvers* (VBS). A VBS selects the best experiment for each input from a selection of real solvers. We now want to create two different VBSs, simply called `vbs1` and `vbs2`. To do so, we call the method `add_vbew(...)`, to which we specify the set of solvers to consider, the column from which the VBS is computed and the name of the VBS.

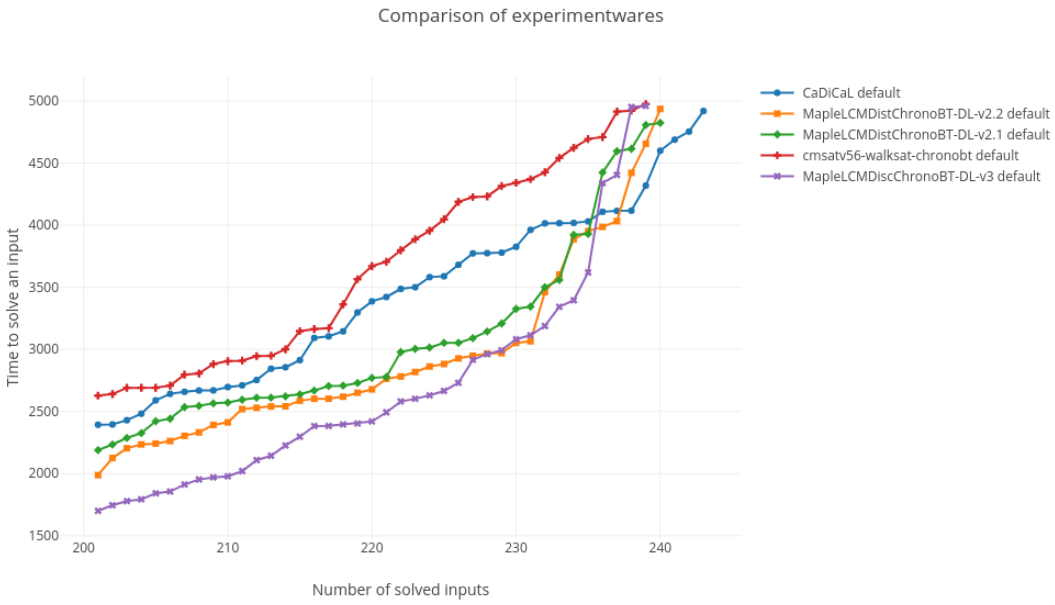

Figure 4: Cactus plot of the leading solvers of the SAT Race 2019.

```
vbs1 = {
    "CaDiCaL default",
    "MapleLCMDistChronoBT-DL-v2.2 default"
}
vbs2 = {
    "CaDiCaL default",
    "MapleLCMDiscChronoBT-DL-v3 default"
}

campaign_df_best_plus_vbs = campaign_df_best\
    .add_vbew(vbs1, "cpu_time", vbew_name="vbs1")\
    .add_vbew(vbs2, "cpu_time", vbew_name="vbs2")
```

We may now create a table with these VBSs and other time statistics using these three lines of code:

```
from metrics.wallet.figure.static_figure import StatTable
stat = StatTable(campaign_df_best_plus_vbs)
stat.get_figure()
```

Table 1 shows, from left to right, the solvers ordered by number of solved inputs. Obviously, we can observe that VBSs have best ranks. They are followed by *CaDiCaL* and *MapleLCMDistChronoBT-DL* solvers. Thanks to this view, we can also remark that time (PAR1) is not continuously increasing. Indeed, we can see that, in spite of a lower number of solved instances and the penalty of the PAR1, *Maple*-based solvers have lower results in terms

of time than *CaDiCaL*. This is also the case when considering the cumulated time of common solved instances (common_sum is computed on 202 instances).

	vbs1	vbs2	CaDiCaL	MapleLCM...-v2.2	MapleLCM...-v2.1	MapleLCM...v3	cmsatv56-walksat-chronobt
count	264	262	244	241	241	240	240
sum (PAR1)	224776	240614	408361	385357	385468	365054	449371
common_count	202	202	202	202	202	202	202
common_sum	106496	101773	198214	182257	185539	140238	202954
uncommon_count	62	60	42	39	39	38	38
total	400	400	400	400	400	400	400

Table 1: Table of solved instances and runtime statistics of the leading solvers of the SAT Race 2019.

Another way to observe in details the behaviour of *CaDiCaL* and *MapleLCMDistChronoBT-DL-v2.2* is to use a scatter plot:

```
from metrics.wallet.figure.dynamic_figure import ScatterPlotly
stat = ScatterPlotly(
    campaign_df,
    "CaDiCaL default", "MapleLCMDistChronoBT-DL-v2.2 default",
    scatter_col="cpu_time"
  )
stat.get_figure()
```

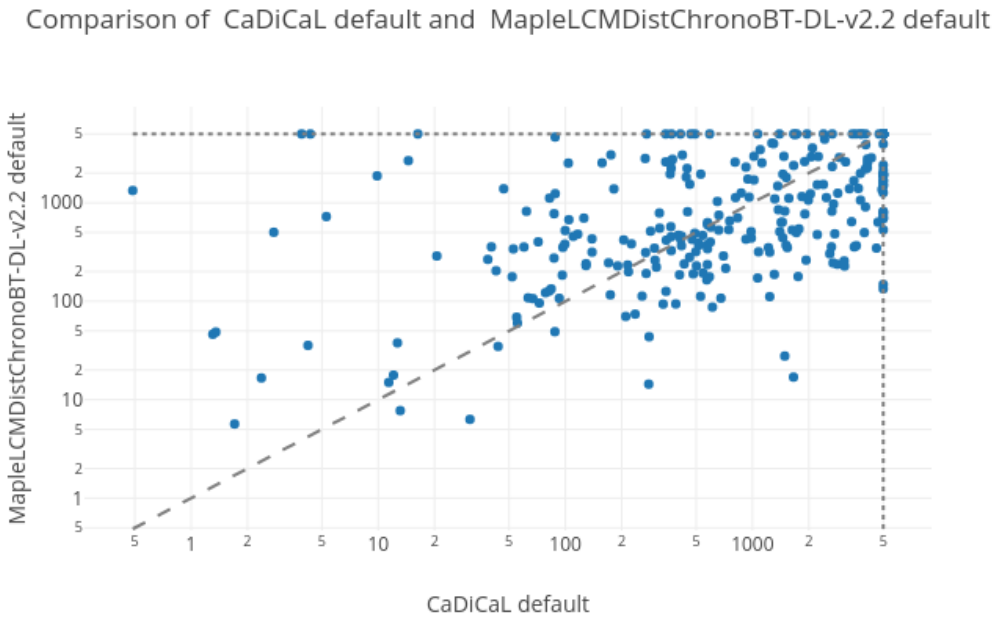

Figure 5: Scatter plot of two leading solvers of the SAT Race 2019.

The produced scatter plot, as shown in Figure 5 does not show any significative trend between the solvers, as we can see an equivalent repartition of points on both sides of the diagonal. However, the capacity to mouse hover the points and show metadata about the different instances could help understand the behaviour of each solver. Also, it is possible to give another column to plot (e.g. `scatter_col="memory_usage"`) permitting to possibly observe a new way to discriminate these two solvers, which are quite close.

Finally, let us introduce a new kind of plots that completes the information provided by the cactus plot: boxplots. Such plots display informations about mean and quartiles. As previously, we just need to instantiate a `BoxPlotly` object with the campaign we want to observe:

```
from metrics.wallet.figure.dynamic_figure import BoxPlotly
box = BoxPlotly(campaign_df_best)
box.get_figure()
```

The resulting boxplots are shown in Figure 6. Each boxplot is composed, from the bottom to the top, of the minimum value, the first and third quartile (the square) and the maximum. The maximum values, in this example, are overwritten by the third quartile. We can observe that *CaDiCaL* is the first solver starting to solve instances. In the box (between first and third quartiles), we see other informations: the full line corresponds to the median, and the dotted line to the mean. As exposed in Table 1, we can extract the information that the time is slightly better for *Maple*-based solvers: their median is lower than that of *CaDiCaL*.

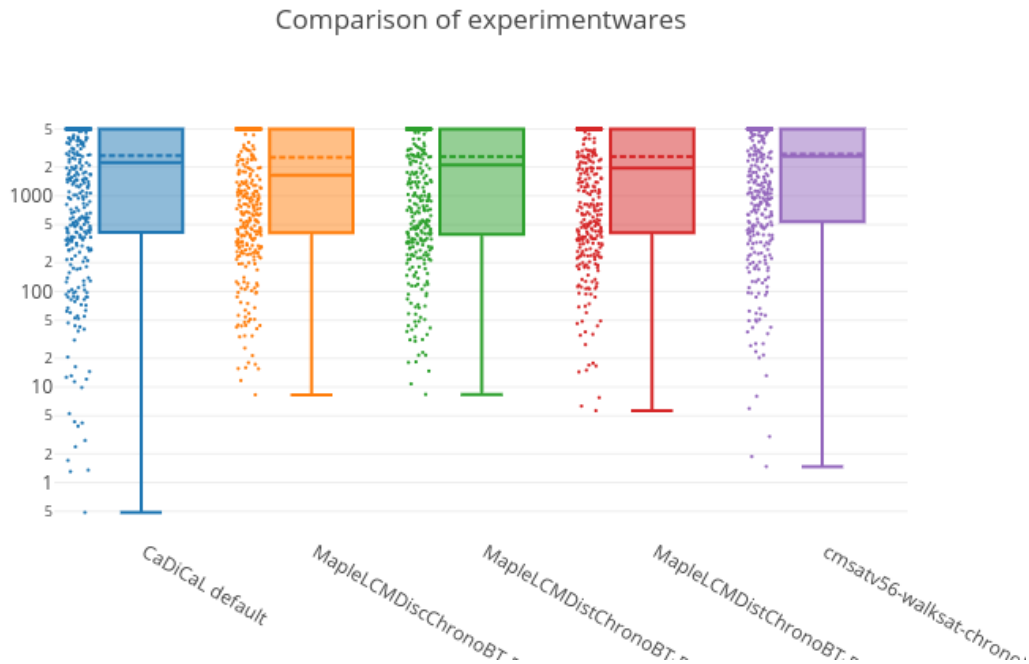

Figure 6: Boxplots comparing the runtime of the best solvers of the SAT Race 2019.

Finally, through the statistics and figures provided by *Metrics*, we observe that having a global overview of the campaign corresponding to the SAT Race 2019 is quite hard. Nevertheless, thanks to some filtering operations, we fastly identified the best solvers and compared

them to get more precise conclusions. Nevertheless, it is hard to find a good tie-break method to compare the performance of the different solvers, so as to take into account the number of solved inputs and the time taken to solve them. That is why a diversity of tools have been designed to clarify the situation.

In the actual competition, *MapleLCMDiscChronoBT-DL-v3* won the race by applying a scoring method based on PAR2 to rate solvers. This result is in accordance to our results as this solver takes place in the best ones, with the lowest PAR1 time and a common time, even though *CaDiCaL* solves more inputs within the time limit.

5 Conclusion

In this paper, we presented *Metrics*, a work-in-progress library providing an easy-to-use toolchain for retrieving experimental data and analyzing this data through the computation of statistics and the drawing of different kinds of plots. Users are free to organize and customize this analysis according to their needs thanks to the use of Jupyter notebooks. This allows to share the results of the conducted experiments and to make possible the reproducibility of their analysis.

Currently, *Metrics* is a "young" project, and we plan to add more features to this library, so as to make a deeper analysis of the results and draw different kinds of plots. Even though users can still use the data-frames of the campaign and make their analysis with the functions already provided by `pandas` and `matplotlib`, we believe that, to make the user's life easier, *Metrics* should provide such features by default.

Moreover, to become a *complete* toolchain, *Metrics* has the ambition to extend its capabilities by providing a command-line interface as well as a web application to perform standard operations, by automating the process of collecting and analyzing data. In particular, by properly configuring their *Metrics* installation, users would be able to submit their softwares directly from *Metrics*' interface, so that everything from the execution of the solver (including on remote machines, e.g., in the cloud or on clusters) to the writing of a complete report, with little effort required from the users.

References

[1] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018.

[2] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *J. Satisf. Boolean Model. Comput.*, 7(2-3):59–6, 2010.

[3] Juliana Freire, Norbert Fuhr, and Andreas Rauber. Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041). *Dagstuhl Reports*, 6(1):108–159, 2016.

[4] Thomas Herndon, Michael Ash, and Robert Pollin. Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. *Cambridge Journal of Economics*, 38(2):257–279, 2014.

[5] Yang-Min Kim, Jean-Baptiste Poline, and Guillaume Dumas. Experimenting with reproducibility: a case study of robustness in bioinformatics. *GigaScience*, 7(7):giy077, July 2018.

[6] Dirk Pilat and Yukiko Fukasaku. Oecd principles and guidelines for access to research data from public funding. *Data Science Journal*, 6:4–11, 06 2007.

[7] Carmen Reinhart and Kenneth Rogoff. Growth in a time of debt. *American Economic Review*, 100:573–78, 05 2010.

[8] Olivier Roussel. Controlling a Solver Execution: the runsolver Tool. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.