ACVIZ: A Tool for the Visual Analysis of the Configuration of Algorithms with irace

Marcelo de Souza[a,b,*], Marcus Ritt[b], Manuel López-Ibáñez[c] and Leslie Pérez Cáceres[d]

[a]*Santa Catarina State University, Brazil*
[b]*Federal University of Rio Grande do Sul, Brazil*
[c]*University of Málaga, Spain*
[d]*Pontificia Universidad Católica de Valparaíso, Chile*

ARTICLE INFO

Keywords:
algorithm configuration
parameter tuning
irace

ABSTRACT

This paper introduces acviz, a tool that helps to analyze the automatic configuration of algorithms with irace. It provides a visual representation of the configuration process, allowing users to extract useful information, e.g. how the configurations evolve over time. When test data is available, acviz also shows the performance of each configuration on the test instances. Using this visualization, users can analyze and compare the quality of the resulting configurations and observe the performance differences on training and test instances.

1. Introduction

Many algorithms have input parameters that allow adapting their behavior to the problem being solved. A specific parameter configuration often has an impact on the performance of the algorithm. The search for good configurations is a fundamental step of the algorithm design. There are different tools, called *configurators*, for configuring algorithms automatically. Examples of such configurators include ParamILS [1], SMAC [2], GGA [3], and irace [4]. They reduce the human effort required for comparing several parameter configurations on different problem instances, minimize the human bias and make the configuration process reproducible.

In this work, we focus on irace [4], a configurator written in R and widely used in different domains. The main application of irace is the automatic configuration [5] of optimization [6, 7, 8, 9] and decision [10, 11] algorithms. Besides that, irace was used to configure the parameters of the GCC compiler [12], the CPLEX optimization software package [13, 10, 11], machine learning models [14, 15, 16], and also for improving the anytime behavior of optimization algorithms [17]. Some works define a parameterized framework with algorithmic design choices, then use irace to automatically design algorithms [18]. This approach was applied to design algorithms for different problems, including permutation flowshop scheduling [19, 20, 21, 22], binary quadratic programming [23, 24], bin packing [25], and also to construct control software for robots [26, 27, 28]. The irace configurator was also used to configure and design algorithms to tackle multi-objective problems, including evolutionary approaches [29, 30, 31, 32], ant colony optimization [33, 34], hybrid local searches [35] and clustering algorithms [27]. Additional applications of irace include the optimization of traffic light programs [36] and the analysis of configurations through ablation [37].

The use of configurators like irace allows users to adjust algorithms for obtaining high performance without the need of vast expert knowledge about the algorithm or the problem. The configuration process implemented in irace often generates large volumes of algorithm performance data that is used to guide the search for good configurations. Additionally, the data produced by irace can be used to obtain insights about the configured algorithm and the configuration process. Despite the widespread use of irace, many users apply it without a careful analysis of its operation, i.e. they simply use the tool as a black-box method for configuring algorithms. Nevertheless, understanding how the configurator works and analyzing its execution is essential to obtain the best results from the configuration process and to ensure the efficient use of the computational resources. This understanding is essential when designing the configuration scenario, which defines the parameter space, the training instances, and the available computational resources. The configuration scenario can be setup inadequately, e.g. by using too little or too much computational effort may lead to poor results or the waste of available computational resources. Using training instances that are not representative of typical problem instances may lead to poor results on a separate set of test instances. Too much configuration effort and insufficiently diverse training instances will lead to overtuning [38], i.e. the algorithm performs increasingly worse on the test instances as more effort is spent on the configuration process. A detailed analysis of the configuration process helps to identify such cases and adjust the configuration scenario. Currently, there are no tools available to directly visualize the configuration process data and thus, promoting and simplifying the analysis of it.

We present acviz, a visual tool to analyze runs of irace

acviz is available at https://github.com/souzamarcelo/acviz.

*Corresponding author

marcelo.desouza@udesc.br (M. de Souza); marcus.ritt@inf.ufrgs.br (M. Ritt); manuel.lopez-ibanez@uma.es (M. López-Ibáñez); leslie.perez@pucv.cl (L. Pérez Cáceres)

ORCID(s): 0000-0002-0786-2127 (M. de Souza); 0000-0001-7894-1634 (M. Ritt); 0000-0001-9974-1295 (M. López-Ibáñez); 0000-0001-5553-6150 (L. Pérez Cáceres)

Algorithm 1: Iterated racing procedure

```
Input  : Configuration scenario $\langle \Theta, \Pi, c \rangle$ and
         computational budget $B$.
Output: Set of best configurations $\Theta^{\text{elite}}$.
1 $\Theta^{\text{elite}} \leftarrow \emptyset$
2 repeat
3     $\Theta' \leftarrow \texttt{sample}(\Theta, \Theta^{\text{elite}})$
4     $\Theta^{\text{elite}} \leftarrow \texttt{race}(\Theta' \cup \Theta^{\text{elite}}, \Pi, c)$
5 until budget $B$ is exhausted
6 return $\Theta^{\text{elite}}$
```

based on the graphical representation of the configuration process. The acviz tool provides two types of visualizations. The first shows the evolution of a single run of the configuration process performed by irace. The second visualizes the performance of the best found configurations on test instances and contrasts them with the performance on the training instances used by irace. This paper describes acviz in detail and presents examples that show how it can be used to understand the configuration process, and how it can provide useful information to design better configuration scenarios.

The remainder of this paper is organized as follows. Section 2 reviews the basic concepts about automatic algorithm configuration and explains the irace configurator. Section 3 introduces the acviz program and its functions. Section 4 presents examples of applying acviz to analyze the automatic configuration of algorithms. Finally, Section 5 gives some concluding remarks.

2. Automatic algorithm configuration

Let $\mathcal{A}$ be a target algorithm with parameters $p_1, p_2, \ldots, p_n$ and corresponding domains $\Theta_1, \Theta_2, \ldots, \Theta_n$. The parameter space Θ is a subset of $\Theta_1 \times \Theta_2 \times \cdots \times \Theta_n$, from which invalid parameter combinations are excluded. Parameters that determine the selection of algorithmic components (e.g. the neighborhood operator to be used in a local search) have usually categorical or ordinal domains. Parameters that control the behavior of algorithmic components (e.g. the perturbation size of an iterated local search) are usually numerical and have integer or real domains. A *configuration* of the algorithm $\theta \in \Theta$ is a valid assignment of values to all parameters.

Given a set Π of problem instances, the performance of a particular run of the target algorithm with configuration $\theta \in \Theta$ on instance $\pi \in \Pi$ is given by some function $c(\theta, \pi)$. For optimization scenarios, c is usually the cost of the best solution found after running the algorithm for a predefined time limit. For decision algorithms, c is usually the running time. If $\mathcal{A}$ is stochastic, then $c(\theta, \pi)$ is a random variable. The algorithm configuration task consists in finding at least one good configuration $\theta \in \Theta$ that optimizes the expected performance of running $\mathcal{A}$ under θ on instances Π.

The irace configurator [4] uses iterated racing [39, 40] for the automatic configuration of algorithms. The basic steps of irace are shown in Algorithm 1. Given the configuration scenario $\langle \Theta, \Pi, c \rangle$ and a computational budget B, irace iteratively samples a population of configurations Θ', and evaluates them using a racing procedure. The best found configurations form the elite set Θ^{elite}, which is used to guide the sampling of new configurations. The sampling process followed by the racing procedure are repeated while the budget B is not exhausted. The budget B can be defined as a maximum number of configuration evaluations or an execution time limit. The number of iterations is determined by irace at the beginning of the configuration process, based on the number of parameters to be configured. The budget of an iteration is determined at the start of the iteration, based on the remaining budget available and the number of iterations to be executed next.

The sampling phase (line 3 of Algorithm 1) behaves as follows. At the beginning of the execution, irace samples the parameter space Θ uniformly, since Θ^{elite} is empty. In subsequent iterations, irace ranks the elite configurations according to their performance in previous evaluations, and iteratively selects one of them to generate each new configuration θ. Elite configurations of a higher rank have a higher probability of being selected. The value of each parameter of θ is determined based on probability distributions associated with its parent. Newly generated configurations inherit this set of probability distributions (one for each parameter) from their parents. The parameters of these distributions are updated at the beginning of each iteration to focus the sampling process around the best parameters values.

The racing phase (line 4 of Algorithm 1) evaluates the quality of the new and elite configurations (Θ' and Θ^{elite}, respectively) on a subset of the instances Π according to the performance metric c. After evaluating each configuration on a predefined number of initial instances, the configurations that perform statistically worse than the best one are discarded. The remaining configurations are evaluated on a new instance before performing a new statistical test. This process is repeated until the budget of the iteration is exhausted, or a minimum number of surviving configurations is reached. The surviving configurations become the elite set for the next iteration.

The updates of the probability distributions may lead to a premature convergence of the configuration process. In this case, the newly generated configurations are very similar to those already evaluated and the configuration process loses diversity. To avoid this, irace implements a convergence detection mechanism that compares each new configuration to the elite configuration used to generate it. The comparison is carried out by calculating their distance, based on the differences of the parameter values presented by both configurations. If this distance is less than a threshold, irace performs a soft restart that updates the parameters of the sampling distributions associated to that elite configuration, in order to increase the probability of generating different configurations.

Table 1
Arguments of acviz (default options are shown in bold).

Argument	Options	Description
`--iracelog`	<log file>	The irace log file (.Rdata)
`--typeresult`	{*aval*, *adev*, ***rdev***}	Which values are presented
`--bkv`	<bkv file>	The file containing reference values
`--imputation`	{***elite***, *alive*}	Imputation strategy for missing values
`--scale`	{***log***, *lin*}	Scaling of the y-axis
`--noelites`	–	Disables different markers for evaluations of elite configurations
`--noinstances`	–	Disables coloring evaluations on different instances
`--pconfig`	[**0**, 1]	Identifies the configurations of the best evaluations
`--overtime`	–	Presents the configuration time on the x-axis
`--alpha`	[0, **1**]	The opacity of the points
`--timelimit`	[**0**, ∞]	Time limit used to evaluate decision algorithms
`--testing`	–	Presents the plot of the test phase
`--testcolors`	{***instance***, *overall*}	The scheme for the color map
`--exportdata`	–	Exports the data of the configuration process to a csv file
`--exportplot`	–	Exports the produced plot to pdf and png files
`--output`	<prefix>	The prefix name of the exported files
`--monitor`	–	Monitors the irace log file and updates the plot after each iteration

3. The acviz program

Given a log file produced by running irace, the acviz program provides visualizations of the configuration process. Figure 1 gives examples of the configuration of two different algorithms. A point (i, v) shows the performance v obtained in the ith evaluation in the configuration process. Note that each evaluation is associated to a unique configuration-instance pair (θ, π). The first example shows the configuration of an optimization algorithm. In this case, the performance value v is the relative deviation of the best solution found in each evaluation from an instance-based reference performance value. These reference values can be provided by the user when, for example, there are best known solutions for the instances or there is a current default configuration and its performance can be used as reference. When no reference value is provided, the best values found by irace are used. The plot also shows the beginning of each iteration by a vertical dashed line, with the number of evaluations (bottom) and the number of different instances (top) used until that iteration. This vertical line is presented in red for iterations in which a soft restart was applied. Finally, evaluations on different instances are indicated by different colors, and evaluations of elite configurations are represented using different markers (● for elite configurations of the current iteration, ◆ for configurations that were elite in the final iteration, and ★ for the best found configuration, i.e. the first ranked elite configuration of the final iteration).

The horizontal lines present the estimated performance of the elite (purple line) and non-elite (orange line) configurations in each iteration. The estimated performance is determined by the median of the results obtained by all configurations of the current iteration on all instances evaluated so far, considering evaluations in the current and previous iterations. Some of the non-elite configurations may not be evaluated on a subset of the instances, e.g. when the configuration is discarded in the middle of an iteration. For the calculation of the estimated performance, we replace missing values by the worst result of the elite configurations, since the eliminated configuration is not better than the worst elite configuration (called *elite* imputation strategy). An alternative approach is to use the worst result of the configurations being evaluated in the current iteration (called *alive* imputation strategy).

Table 1 details the input arguments of acviz. The command to produce the first visualization shown in Figure 1 is:

```
python3 acviz.py --iracelog irace.Rdata --bkv bkv.txt
```

which provides the irace log file to be used, in this case `irace.Rdata`, and the file containing the reference values used to compute the relative deviations (`bkv.txt`). Additional options control the elements of the visualization, like presenting the absolute performance values or the absolute deviations from the reference values (option `--typeresult`), or changing the imputation strategy. Users can also disable the coloring of instances and the markers of elite configurations, or tell acviz to show the ID of the configurations associated with the $p\%$ best performing evaluations of each iteration (option `--pconfig`). The opacity of the points can be changed and the default logarithmic scale of the y-axis can be disabled.

The second visualization in Figure 1 shows the configuration of a decision algorithm, where the performance of each evaluation is the running time used to solve the corresponding instance. In this case, the configuration budget is a time limit, then users can opt to plot the starting time of evaluations on the x-axis (option `--overtime`), making it possible to observe how the configuration time is distributed over the iterations, and identify evaluations that took a long time. To produce this visualization, we select to show absolute performance values in the y-axis and disable the logarithmic scale.

During the configuration process, evaluations that reach the running time limit without solving the instance are penalized by returning to irace the time limit multiplied by a penalization factor [10]. If we inform the time limit to acviz, each evaluation with a result that exceeds this limit is presented in the upper border of the plot, indicating that these evaluations did not solve the instance (see those cases in the second visualization shown in Figure 1). The following command produces this visualization (observe that argument `--iracelog` can be omitted):

```
python3 acviz.py irace.Rdata --typeresult aval
    --scale lin --timelimit 10 --overtime
```

A second plot provided by acviz presents the results obtained by the best found configurations on the set of test instances (this requires the testing feature to be enabled when running irace). Figure 3 shows an example, presenting the results of the best elite configurations of each iteration and all elite configurations of the last iteration. Each column in the plot is associated with a configuration. The acviz tool presents its ID and, in parenthesis, the iterations in which it was the first ranked elite configuration (e.g. 251 $(3, 4)$ means configuration 251 was the best ranked elite configuration in iterations 3 and 4). For the final iteration, we also present the rank of the corresponding configuration in the elite set in a subscript (e.g. 9_1 means that the configuration was ranked first in the 9th iteration). The instance name is black if the instance has been used during training and testing, and blue, if it has been used only for testing. The subplot on the left shows the mean relative deviations from the reference values that, as for the previous plot, can be provided using the `--bkv` option, or are determined by acviz based on the best values found during the execution of irace (in both training and test phases). The subplot on the right presents the ranking of each configuration on each instance, allowing us to compare the performance of different configurations across instances. The command to produce the visualization shown in Figure 3 indicates that acviz should present the plot of the test phase:

```
python3 acviz.py irace.Rdata --bkv bkv.txt --testing
```

In the visualization of the test phase, we can also use option `--typeresult` to present the mean absolute values or the mean absolute deviations from the reference values. The colors in the plot help to differentiate the performance obtained by the resulting configurations. In Figure 3, the color map is calculated according to the results obtained within each instance. Worst values for each instance are in red while the best values are in green. Alternatively, the color map can be defined according to the whole range of values in all instances, thus visualizing the overall performance obtained in the test phase.

When using the interactive presentation mode, acviz allows the user to control the visualization by moving the plot, zooming and controlling the margins of the figure. When positioning the cursor over a point, acviz shows a tooltip box with the corresponding evaluation number, the associated instance name and configuration ID. It is also possible to export the data and both of the plots. Finally, when option `--monitor` is enabled, acviz monitors the irace log file during the configuration process and updates the visualization after each iteration, allowing the user to analyze the evolution of the configuration process during its execution. All acviz options discussed above are summarized in Table 1.

4. Analyzing the configuration process with acviz

In this section we present three exemplary case studies of configurations with flaws that can be easily identified when using acviz. We use two scenarios from the Algorithm Configuration Library [41]. The first scenario considers the configuration of ACOTSP [42], a framework of ant colony optimization algorithms, applied to the symmetric traveling salesperson problem (TSP) [43]. ACOTSP has 11 parameters, 5 of which are conditional. We run ACOTSP with a time limit of 20 seconds of CPU time. In the first and second case studies, we use the Euclidean TSP instances with 2000 cities from a previous study [44]. For the third case study, we additionally use ten TSP instances with uniformly random distance matrices, generated with *portmgen* from the 8th DIMACS Implementation Challenge [45]. In this scenario, irace optimizes the cost of the best found solution.

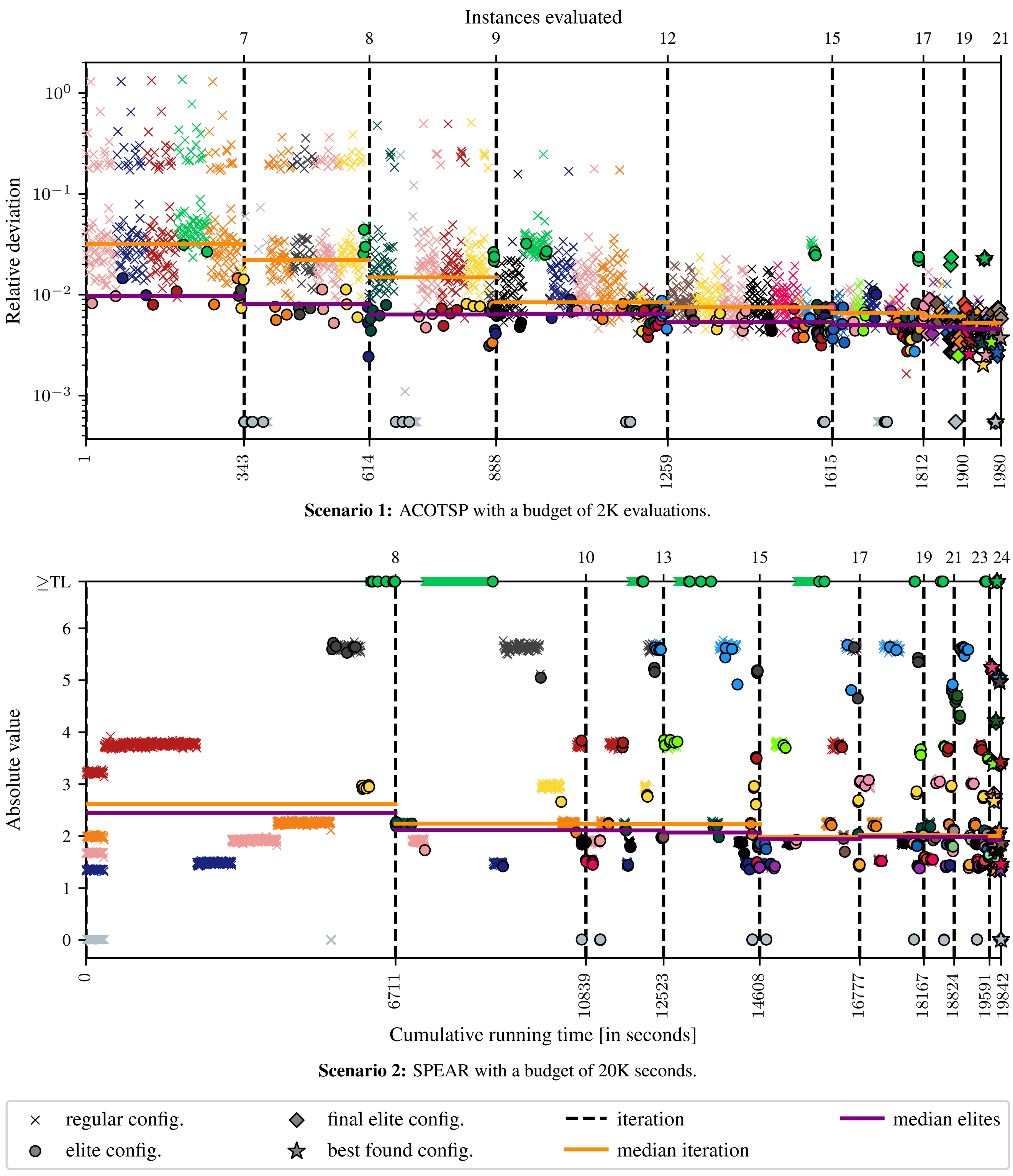
The second scenario considers the configuration of SPEAR [46], an exact solver for boolean satisfiability (SAT) problems. SPEAR has 26 parameters, 9 of which are conditional. We use the SAT-encoded instances of graph coloring from Gent et al. [47], and a limit of 10 seconds of wall-clock time. Here, irace minimizes SPEAR's solving time, where for evaluations in which the instance is not solved, a penalized performance value is returned.

In all experiments, we use the default settings of irace. The detailed results of all experiments are available from De Souza et al. [48]. The source code of acviz, usage instructions, and further application examples are available on the project website (`https://github.com/souzamarcelo/acviz`).

4.1. Case study 1: Easy and hard instances

In this section we discuss two example scenarios and show how easy and hard instances can be identified. We configure ACOTSP with a budget of 2K evaluations, and SPEAR with a budget of 20K seconds. Figure 1 shows the visualizations produced by acviz. When configuring

Please cite this pre-print manuscript as: Marcelo De Souza, Marcus Ritt, Manuel López-Ibáñez, and Leslie Pérez Cáceres. ACVIZ: A Tool for the Visual Analysis of the Configuration of Algorithms with irace. *Operations Research Perspectives*, 2021. doi: 10.1016/j.orp.2021.100186



Scenario 1: ACOTSP with a budget of 2K evaluations.

Scenario 2: SPEAR with a budget of 20K seconds.

Figure 1: Example of configuring ACOTSP and SPEAR using instances of different hardness.

ACOTSP we observe the evolution of the configuration process, i.e. how the sampled configurations present better performance over the iterations. At the beginning of the configuration, there is a subset of the configurations with bad performance on almost all evaluated instances (points in the upper part of the figure in the four first iterations). The number of such bad performers decreases over the iterations, while the estimated performance of elite and non-elite configurations (the median values given by the horizontal lines) becomes better. We can also see that the instance selection strategy implemented in irace iteratively increases the number of instances on which the configurations are evaluated.

In the configuration of SPEAR, the estimated performance of the configurations also improves over the iterations. Besides that, we see that different configurations have a similar performance on each instance. Note that the evaluations of different configurations on a particular instance, represented by clusters of points of the same color, present a small variation of the running time. Nevertheless, we observe that elite configurations perform better than others, since they are often among the best in each cluster.

The visualizations shown in Figure 1 also provide some information about the configuration scenarios. We can see that both scenarios are quite homogeneous, i.e. a configuration with good performance on one instance often presents good performance on the others. For example, if we look at the elite configurations (● markers) of each iteration, we see that they present the best results for almost all instances. This contributes to irace easily identifying the best configurations in the racing phase, and consequently, using less evaluations than the budget available for the iterations. The saved budget is then used to perform more iterations than the five initially scheduled, as observed in the plot. Those additional iterations are increasingly shorter because they are consuming the remaining budget and fewer new configurations are sampled.

We included in both scenarios two additional instances: one that is easy to solve, shown in gray, and another that is hard to solve, shown in green. Figure 1 shows an interesting behavior of the configurations on those instances. In ACOTSP, we can see that almost all configurations perform very well on the easy instance. Besides that, there is no variation of different configurations on this instance. Therefore, the evaluations on this instance do not help to determine the quality of different configurations and decide which one is better. In the case of the hard instance, we observe that it helps to differentiate the quality of the configurations in the first iterations. However, as for the easy instance, from the fifth iteration on, it stops being useful for the configuration process.

In the configuration of SPEAR it is even more evident that the easy and hard instances do not contribute to the configuration process. We observe that all configurations immediately solve the easy instance, while no configuration solves the hard instance. In this case, we could exclude the easy instance from the configuration scenario, since it does not help to evaluate the configurations. We could also exclude the hard instance, or increase the time limit, trying to find configurations that can solve it. In this context, acviz helps to identify such cases by visualizing and comparing how the configurations perform on the training instances and which ones are actually contributing to the configuration process.

4.2. Case study 2: Unnecessarily large budget

Choosing an adequate configuration budget can be difficult. A small budget may not be sufficient to find good configurations. A common practice is to use the highest possible budget, according to time constraints and the available computational resources. However, even after running irace, it may not be clear if the chosen budget was appropriate. In this second experiment, we configure both ACOTSP and SPEAR with very large budgets of 100K evaluations and 500K seconds, respectively. Figure 2 shows the resulting visualizations. Since the budget is larger, irace samples more configurations and uses more instances to evaluate them.

In ACOTSP, the observed behavior is similar to the first case study. We can see a fast evolution of the configuration process in the first iterations, producing configurations with better performance compared to those obtained in the first case study. From the fifth iteration on, after approximately 20K evaluations, the quality of the sampled configurations stagnates. Note that the estimated performance of both elite and non-elite configurations (orange and purple horizontal lines) does not improve from that point until the end of the configuration process. We can also see that irace performs a soft restart (red dashed line) in almost all subsequent iterations, which indicates that the sampling models converged. The same behavior is observed in the configuration of SPEAR, where soft restarts are present after about 400K seconds.

In both scenarios, if we needed to repeat the process, we could decrease the budget to about 20K$\sim$30K evaluations (ACOTSP) or 300K$\sim$ 400K seconds (SPEAR), for example. Alternatively, if we have the time for a large budget, we could tell irace to sample more configurations at each iteration to increase diversification (parameter `nbConfigurations`). For heterogeneous scenarios, the additional budget could be better spent in increasing the number of instances evaluated before the first and between each elimination test (parameters `firstTest` and `eachTest` of irace, respectively).

4.3. Case study 3: Unrepresentative instances

A common mistake when configuring algorithms is to choose training instances that are not representative of the instances to be used in production. Suppose, for example, we use an algorithm that has been configured on a certain class of instances $\mathcal{I}$. Now, we want to solve additional instances of class $\mathcal{I}'$. In order to get the best performance, we may want to configure the algorithm again to reflect the new instance distribution. If the goal is to obtain a configuration that performs well for both instance classes, then starting the configuration process from the current configuration (tuned for $\mathcal{I}$) and training only on instances of $\mathcal{I}'$ would be a methodological mistake.

In this experiment, we reproduce the above situation using ACOTSP to analyze how the configuration process behaves. In a first step, we select ten TSP instances with random distances and tune ACOTSP on them to obtain a set of initial configurations. Then, we select ten Euclidean TSP instances, and use them as training instances for an irace run with a budget of 3K evaluations. We provide the configurations from the first step as initial configurations. We use the testing options of irace to evaluate the resulting configura-

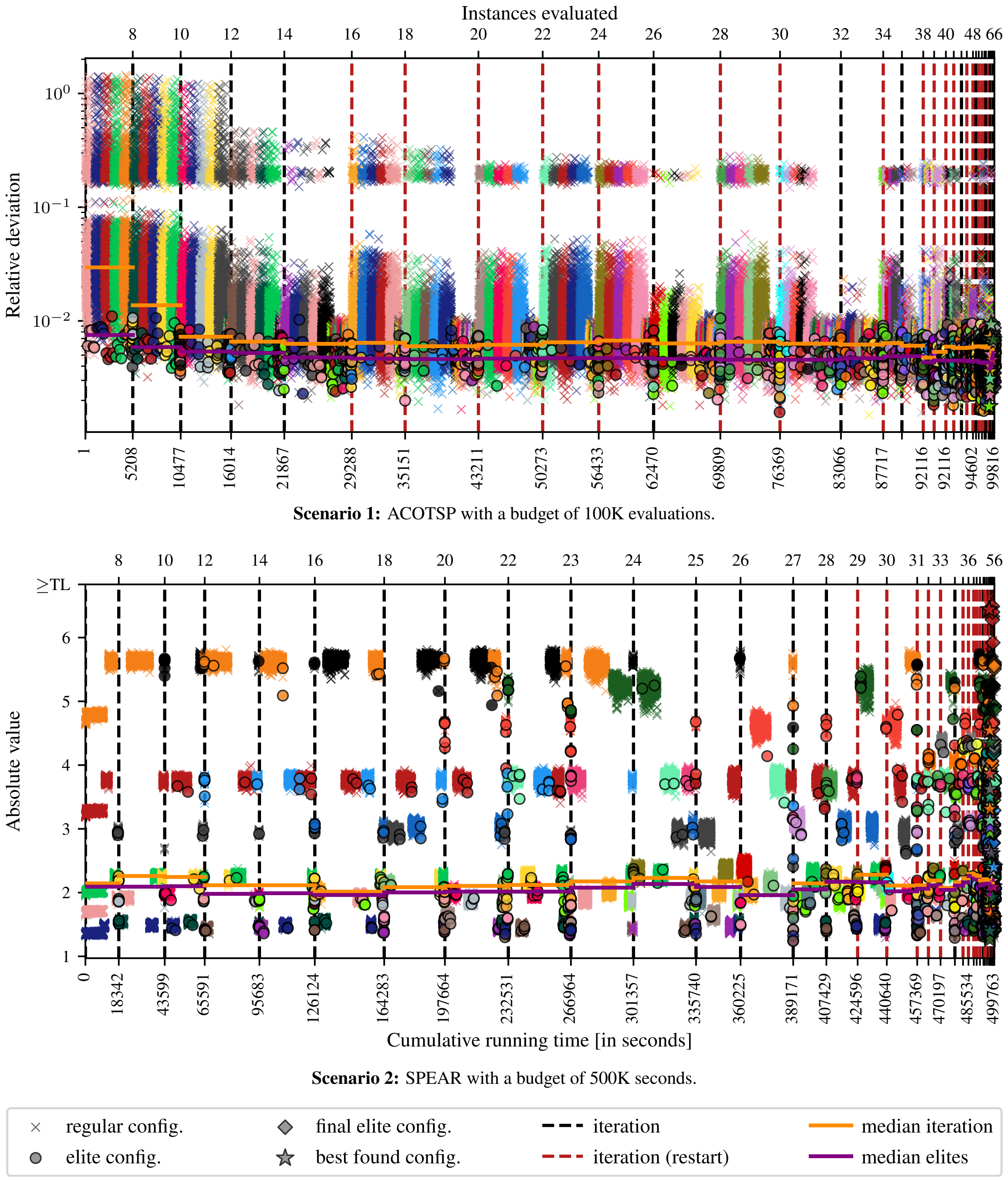
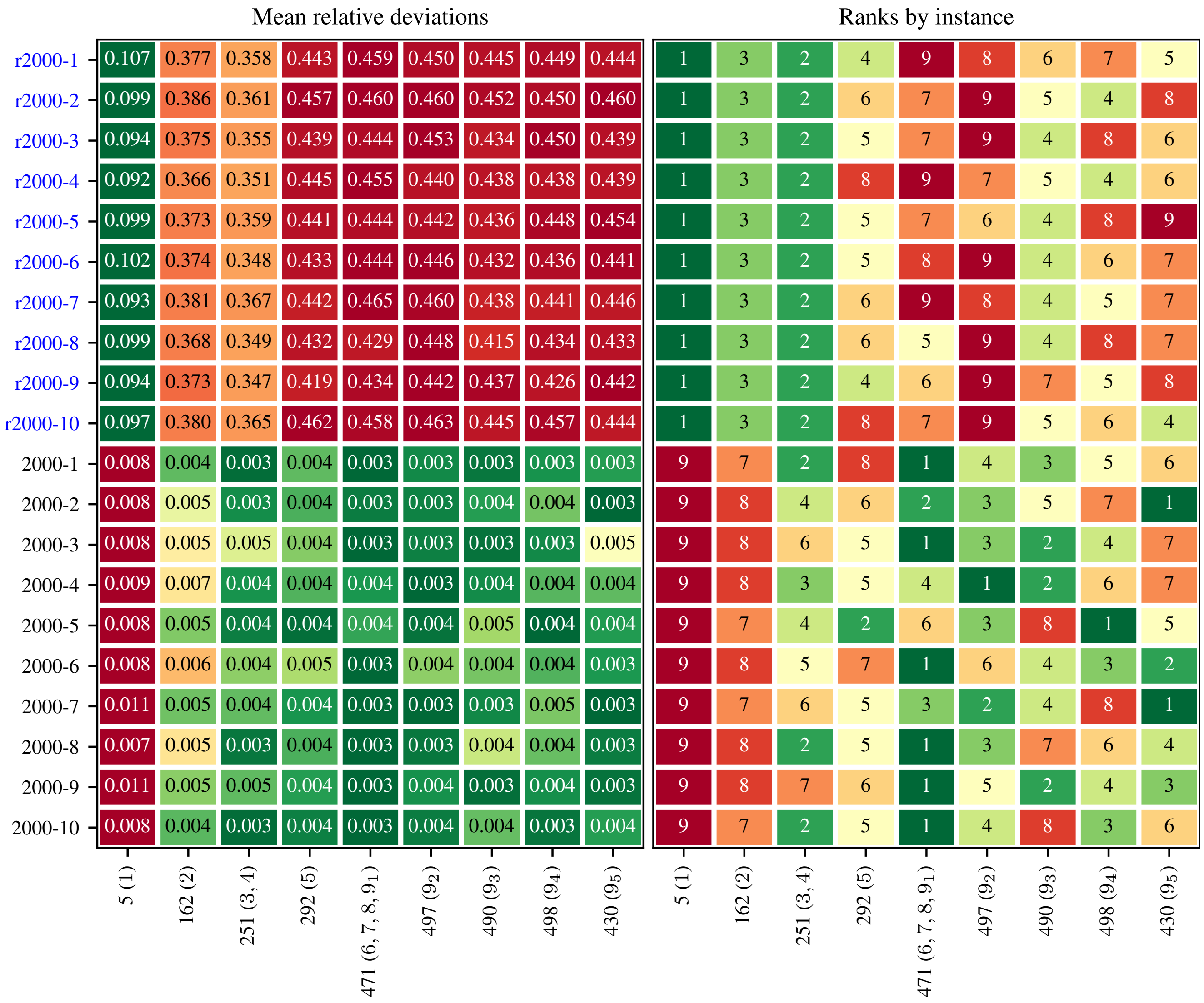


Figure 2: Example of configuring ACOTSP and SPEAR with large budgets.

tions on all Euclidean instances (used as training set) and all random distance instances (not used as training set, thus we call it test set). Random distance and Euclidean instances define structurally different TSP instances and thus, ACOTSP configurations that exhibit high performance in one instance class are not expected to maintain such high performance in the other class. The testing results are shown in Figure 3. Since we evaluate the resulting configurations on both train-



Mean relative deviations

	5 (1)	162 (2)	251 (3, 4)	292 (5)	471 (6, 7, 8, 9_1)	497 (9_2)	490 (9_3)	498 (9_4)	430 (9_5)
r2000-1	0.107	0.377	0.358	0.443	0.459	0.450	0.445	0.449	0.444
r2000-2	0.099	0.386	0.361	0.457	0.460	0.460	0.452	0.450	0.460
r2000-3	0.094	0.375	0.355	0.439	0.444	0.453	0.434	0.450	0.439
r2000-4	0.092	0.366	0.351	0.445	0.455	0.440	0.438	0.438	0.439
r2000-5	0.099	0.373	0.359	0.441	0.444	0.442	0.436	0.448	0.454
r2000-6	0.102	0.374	0.348	0.433	0.444	0.446	0.432	0.436	0.441
r2000-7	0.093	0.381	0.367	0.442	0.465	0.460	0.438	0.441	0.446
r2000-8	0.099	0.368	0.349	0.432	0.429	0.448	0.415	0.434	0.433
r2000-9	0.094	0.373	0.347	0.419	0.434	0.442	0.437	0.426	0.442
r2000-10	0.097	0.380	0.365	0.462	0.458	0.463	0.445	0.457	0.444
2000-1	0.008	0.004	0.003	0.004	0.003	0.003	0.003	0.003	0.003
2000-2	0.008	0.005	0.003	0.004	0.003	0.003	0.004	0.004	0.003
2000-3	0.008	0.005	0.005	0.004	0.003	0.003	0.003	0.003	0.005
2000-4	0.009	0.007	0.004	0.004	0.004	0.003	0.004	0.004	0.004
2000-5	0.008	0.005	0.004	0.004	0.004	0.004	0.005	0.004	0.004
2000-6	0.008	0.006	0.004	0.005	0.003	0.004	0.004	0.004	0.003
2000-7	0.011	0.005	0.004	0.004	0.003	0.003	0.003	0.005	0.003
2000-8	0.007	0.005	0.003	0.004	0.003	0.003	0.004	0.004	0.003
2000-9	0.011	0.005	0.005	0.004	0.003	0.004	0.003	0.004	0.003
2000-10	0.008	0.004	0.003	0.004	0.003	0.004	0.004	0.003	0.004

Ranks by instance

	5 (1)	162 (2)	251 (3, 4)	292 (5)	471 (6, 7, 8, 9_1)	497 (9_2)	490 (9_3)	498 (9_4)	430 (9_5)
r2000-1	1	3	2	4	9	8	6	7	5
r2000-2	1	3	2	6	7	9	5	4	8
r2000-3	1	3	2	5	7	9	4	8	6
r2000-4	1	3	2	8	9	7	5	4	6
r2000-5	1	3	2	5	7	6	4	8	9
r2000-6	1	3	2	5	8	9	4	6	7
r2000-7	1	3	2	6	9	8	4	5	7
r2000-8	1	3	2	6	5	9	4	8	7
r2000-9	1	3	2	4	6	9	7	5	8
r2000-10	1	3	2	8	7	9	5	6	4
2000-1	9	7	2	8	1	4	3	5	6
2000-2	9	8	4	6	2	3	5	7	1
2000-3	9	8	6	5	1	3	2	4	7
2000-4	9	8	3	5	4	1	2	6	7
2000-5	9	7	4	2	6	3	8	1	5
2000-6	9	8	5	7	1	6	4	3	2
2000-7	9	7	6	5	3	2	4	8	1
2000-8	9	8	2	5	1	3	7	6	4
2000-9	9	8	7	6	1	5	2	4	3
2000-10	9	7	2	5	1	4	8	3	6

Figure 3: Test results after configuring ACOTSP with unrepresentative training instances. Instances with random distances (starting with 'r' and in blue) were used only for test, while Euclidean instances (in black) were used for both training and test.

ing and test instances, we have useful information about how they perform on both instance sets. The mean deviations give an overview of the results, allowing us to observe the evolution in the quality of configurations found during the configuration process. We can also observe how those configurations compare with each other by analyzing the obtained ranks.

When the training instances are not representative, the found configurations may specialize on the known training instances and present poor performance on unseen test instances. Such an overtuning can be observed in Figure 3. As the configuration progresses, the performance of the configurations is becoming better on the training instances. On the other hand, the performance on the test instances degrades over the iterations. Since we initialized irace with configurations known to perform well on the test instances, the best configuration in the first iteration still performs well on the test set, but the performance quickly degrades on subsequent iterations. To solve this problem, we need to include some random distance instances in the training set, and make sure that the relative frequency of each type of instance seen during training matches their relative frequency in the test set, or the frequency expected in unseen instances when the algorithm is deployed in production.

Reproducibility. All materials necessary for reproducing the experiments are available from De Souza et al. [48]. Experiments were run on a GNU/Linux platform running on an 8-core AMD FX-8150 CPU 3.6 GHz and 32 GB memory. We used acviz 1.1, irace 3.1, ACOTSP 1.03, and SPEAR 1.2.1. The acviz program was written in Python 3 and requires R (≥ 3.4) and the following libraries: numpy (≥ 1.18), pandas ($\geq 1.0.3$), matplotlib (≥ 3.1), and rpy2 (≥ 3.2). Results of ACOTSP, SPEAR, and as a consequence of irace, are sensitive to CPU speed.

5. Concluding remarks

We described in this paper a graphical tool to support the automatic configuration of algorithms with irace. We presented a visualization scheme for the configuration process, which provides useful information to help the design of configuration scenarios. We also presented a second visualization to analyze the performance of the resulting configurations on test instances. We discussed some examples, showing how these visualizations can help to identify common problems when configuring algorithms. Both plots are implemented in the acviz program. Additional features to control the visualization elements and export the results are also provided. The source code of acviz, instructions of use, and further application examples are available at `https://github.com/souzamarcelo/acviz`.

Acknowledgments. This research has been supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. M. de Souza acknowledges the support of the Santa Catarina State University. M. Ritt acknowledges the support of CNPq (grant 437859/2018-5) and Google Research Latin America (grant 25111). L. Pérez Cáceres acknowledges the support of the Fondecyt Iniciación project 11190154. M. López-Ibáñez is a "Beatriz Galindo" Senior Distinguished Researcher (BEAGAL 18/00053) funded by the Spanish Ministry of Science and Innovation (MICINN).

References

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

[2] F. Hutter, H. H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: C. A. Coello Coello (Ed.), Learning and Intelligent Optimization, 5th International Conference, LION 5, volume 6683 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2011, pp. 507–523.

[3] C. Ansótegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, in: I. P. Gent (Ed.), Principles and Practice of Constraint Programming, CP 2009, volume 5732 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2009, pp. 142–157.

[4] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, M. Birattari, The irace package: Iterated racing for automatic algorithm configuration, Operations Research Perspectives 3 (2016) 43–58.

[5] H. H. Hoos, Automated algorithm configuration and parameter tuning, in: Y. Hamadi, E. Monfroy, F. Saubion (Eds.), Autonomous Search, Springer-Verlag, Berlin, Germany, 2012, pp. 37–71.

[6] A. Franzin, T. Stützle, Revisiting simulated annealing: A component-based analysis, Computers & Operations Research 104 (2019) 191–206.

[7] C. Blum, B. Calvo, M. J. Blesa, FrogCOL and FrogMIS: New decentralized algorithms for finding large independent sets in graphs, Swarm Intelligence 9 (2015) 205–227.

[8] M. Mühlenthaler, Fairness in Academic Course Timetabling, Springer-Verlag, 2015.

[9] A. Yarimcam, S. Asta, E. Özcan, A. J. Parkes, Heuristic generation via parameter tuning for online bin packing, in: P. Angelov, et al. (Eds.), Evolving and Autonomous Learning Systems (EALS), 2014 IEEE Symposium on, IEEE, 2014, pp. 102–108.

[10] L. Pérez Cáceres, M. López-Ibáñez, H. H. Hoos, T. Stützle, An experimental study of adaptive capping in irace, in: R. Battiti, D. E. Kvasov, Y. D. Sergeyev (Eds.), Learning and Intelligent Optimization, 11th International Conference, LION 11, volume 10556 of *Lecture Notes in Computer Science*, Springer-Verlag, Cham, Switzerland, 2017, pp. 235–250.

[11] N. Dang Thi Thanh, L. Pérez Cáceres, P. De Causmaecker, T. Stützle, Configuring irace using surrogate configuration benchmarks, in: P. A. N. Bosman (Ed.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, ACM Press, New York, NY, 2017, pp. 243–250.

[12] L. Pérez Cáceres, F. Pagnozzi, A. Franzin, T. Stützle, Automatic configuration of GCC using irace, in: E. Lutton, P. Legrand, P. Parrend, N. Monmarché, M. Schoenauer (Eds.), EA 2017: Artificial Evolution, volume 10764 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2017, pp. 202–216.

[13] L. Pérez Cáceres, T. Stützle, Exploring variable neighborhood search for automatic algorithm configuration, Electronic Notes in Discrete Mathematics 58 (2017) 167–174.

[14] P. Miranda, R. M. Silva, R. B. Prudêncio, Fine-tuning of support vector machine parameters using racing algorithms, in: European Symposium on Artificial Neural Networks, ESSAN, pp. 325–330.

[15] M. Lang, H. Kotthaus, P. Marwedel, C. Weihs, J. Rahnenführer, B. Bischl, Automatic model selection for high-dimensional survival analysis, Journal of Statistical Computation and Simulation 85 (2014) 62–76.

[16] B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, Z. M. Jones, mlr: Machine learning in R, Journal of Machine Learning Research 17 (2016) 1–5.

[17] M. López-Ibáñez, T. Stützle, Automatically improving the anytime behaviour of optimisation algorithms, European Journal of Operational Research 235 (2014) 569–582.

[18] T. Stützle, M. López-Ibáñez, Automated design of metaheuristic algorithms, in: M. Gendreau, J.-Y. Potvin (Eds.), Handbook of Metaheuristics, volume 272 of *International Series in Operations Research & Management Science*, Springer-Verlag, 2019, pp. 541–579.

[19] F. Pagnozzi, T. Stützle, Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems, European Journal of Operational Research 276 (2019) 409–421.

[20] A. Brum, M. Ritt, Automatic algorithm configuration for the permutation flow shop scheduling problem minimizing total completion time, in: A. Liefooghe, M. López-Ibáñez (Eds.), Proceedings of EvoCOP 2018 – 18th European Conference on Evolutionary Computation in Combinatorial Optimization, volume 10782 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2018, pp. 85–100.

[21] A. Brum, M. Ritt, Automatic design of heuristics for minimizing the makespan in permutation flow shops, in: Proceedings of the 2018 Congress on Evolutionary Computation (CEC 2018), IEEE Press, Piscataway, NJ, 2018, pp. 1–8.

[22] M.-E. Marmion, F. Mascia, M. López-Ibáñez, T. Stützle, Automatic design of hybrid stochastic local search algorithms, in: M. J. Blesa, C. Blum, P. Festa, A. Roli, M. Sampels (Eds.), Hybrid Metaheuristics, volume 7919 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2013, pp. 144–158.

[23] M. De Souza, M. Ritt, Automatic grammar-based design of heuristic algorithms for unconstrained binary quadratic programming, in: A. Liefooghe, M. López-Ibáñez (Eds.), Proceedings of EvoCOP 2018 – 18th European Conference on Evolutionary Computation in Combinatorial Optimization, volume 10782 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2018, pp. 67–84.

[24] M. De Souza, M. Ritt, An automatically designed recombination heuristic for the test-assignment problem, in: Proceedings of the 2018 Congress on Evolutionary Computation (CEC 2018), IEEE Press, Piscataway, NJ, 2018, pp. 1–8.

[25] F. Mascia, M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools, Computers & Operations Research 51 (2014) 190–199.

[26] D. G. Ramos, M. Birattari, Automatic design of collective behaviors

for robots that can display and perceive colors, Applied Sciences 10 (2020) 4654.

[27] B. Fisset, C. Dhaenens, L. Jourdan, MO-Mine$_{\text{clust}}$: A framework for multi-objective clustering, in: C. Dhaenens, L. Jourdan, M.-E. Marmion (Eds.), Learning and Intelligent Optimization, 9th International Conference, LION 9, volume 8994 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2015, pp. 293–305.

[28] G. Francesca, M. Brambilla, A. Brutschy, V. Trianni, M. Birattari, AutoMoDe: A novel approach to the automatic design of control software for robot swarms, Swarm Intelligence 8 (2014) 89–112.

[29] F. Campelo, L. S. Batista, C. Aranha, The MOEADr package: A component-based framework for multiobjective evolutionary algorithms based on decomposition, Journal of Statistical Software 92 (2020).

[30] L. C. T. Bezerra, M. López-Ibáñez, T. Stützle, Automatically designing state-of-the-art multi- and many-objective evolutionary algorithms, Evolutionary Computation 28 (2020) 195–226.

[31] L. C. T. Bezerra, M. López-Ibáñez, T. Stützle, Automatic component-wise design of multi-objective evolutionary algorithms, IEEE Transactions on Evolutionary Computation 20 (2016) 403–417.

[32] L. C. T. Bezerra, M. López-Ibáñez, T. Stützle, Automatic design of evolutionary algorithms for multi-objective combinatorial optimization, in: T. Bartz-Beielstein, J. Branke, B. Filipič, J. Smith (Eds.), PPSN 2014, volume 8672 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2014, pp. 508–517.

[33] M. López-Ibáñez, T. Stützle, The automatic design of multi-objective ant colony optimization algorithms, IEEE Transactions on Evolutionary Computation 16 (2012) 861–875.

[34] L. C. T. Bezerra, M. López-Ibáñez, T. Stützle, Automatic generation of multi-objective ACO algorithms for the biobjective knapsack, in: M. Dorigo, et al. (Eds.), Swarm Intelligence, 8th International Conference, ANTS 2012, volume 7461 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2012, pp. 37–48.

[35] J. Dubois-Lacoste, M. López-Ibáñez, T. Stützle, Automatic configuration of state-of-the-art multi-objective optimizers using the TP+PLS framework, in: N. Krasnogor, P. L. Lanzi (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2011, ACM Press, New York, NY, 2011, pp. 2019–2026.

[36] J. Ferrer, M. López-Ibáñez, E. Alba, Reliable simulation-optimization of traffic lights in a real-world city, Applied Soft Computing 78 (2019) 697–711.

[37] C. Fawcett, H. H. Hoos, Analysing differences between algorithm configurations through ablation, Journal of Heuristics 22 (2016) 431–458.

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

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

[40] M. Birattari, Z. Yuan, P. Balaprakash, T. Stützle, F-race and iterated F-race: An overview, in: T. Bartz-Beielstein, M. Chiarandini, L. Paquete, M. Preuss (Eds.), Experimental Methods for the Analysis of Optimization Algorithms, Springer-Verlag, Berlin, Germany, 2010, pp. 311–336.

[41] F. Hutter, M. López-Ibáñez, C. Fawcett, M. T. Lindauer, H. H. Hoos, K. Leyton-Brown, T. Stützle, AClib: A benchmark library for algorithm configuration, in: P. M. Pardalos, M. G. C. Resende, C. Vogiatzis, J. L. Walteros (Eds.), Learning and Intelligent Optimization, 8th International Conference, LION 8, volume 8426 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2014, pp. 36–40.

[42] T. Stützle, ACOTSP: A software package of various ant colony optimization algorithms applied to the symmetric traveling salesman problem, 2002.

[43] M. Dorigo, T. Stützle, Ant Colony Optimization, MIT Press, Cambridge, MA, 2004.

[44] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, M. Birattari, The irace package: Iterated racing for automatic algorithm configuration (supplementary material), `http://iridia.ulb.ac.be/supp/IridiaSupp2016-003`, 2016.

[45] D. S. Johnson, L. A. McGeoch, C. Rego, F. Glover, 8th DIMACS implementation challenge: The traveling salesman problem, `http://dimacs.rutgers.edu/archive/Challenges/TSP`, 2001.

[46] D. Babić, F. Hutter, Spear theorem prover, in: SAT'08: Proceedings of the SAT 2008 Race.

[47] I. P. Gent, H. H. Hoos, P. Prosser, T. Walsh, Morphing: Combining structure and randomness, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence, pp. 654–660.

[48] M. De Souza, M. Ritt, M. López-Ibáñez, L. Pérez Cáceres, ACVIZ: Algorithm configuration visualizations for irace: Supplementary material, `https://doi.org/10.5281/zenodo.4714582`, 2020.