Small or medium-scale focused research project (STREP)

ICT Call 8

FP7-ICT-2011-8

Cooperative Self-Organizing System for low Carbon Mobility at low Penetration Rates

COLOMBO: Deliverable 3.2

Results of the Offline Configuration and Tuning of the Emergent Behavior

Document Information	
Title	Deliverable 3.2 – Results of the Offline Configuration and Tuning of the Emergent Behavior
Dissemination Level	PU (Public)
Version	1.2
Date	17.11.2014
Status	Final.

Authors	Jérémie Dubois-Lacoste (ULB), Thomas Stützle (ULB), Michela Milano (UNIBO), Alessio Bonfietti (UNIBO), Riccardo Belletti (UNIBO), Daniel Krajzewicz (DLR)

Contents

1 Introduction

Virtually all computing systems make use of parameters that steer their behavior. This is true for essentially all algorithms that try to solve computationally hard problems arising, for example, in many classes of optimization problems. In recent years, this fact has engendered significant research efforts in the algorithmic community to develop automatic methods that allow to find performance optimizing parameter settings for algorithms [1, 6, 20, 14, 18, 16]. Automatic in this context means that the problem of assigning appropriate values to the parameters of a specific target algorithm is itself again tackled in an algorithmic way on a meta-level. The algorithm parameters are determined usually in an initial training phase that takes place before the algorithm is actually deployed to solve specific application problems.

Automatic algorithm configuration tools will have and in part already have a strong impact on the way algorithms are designed. This is because a *parameter* in automatic algorithm configuration can also choose, for example, among different types of search strategies. Such decisions are associated rather to algorithm design than to simply calibrating an already fully designed or instantiated algorithm. Hence, the advent of automatic algorithm configuration tools has a major implication not only to the practice of fine-tuning algorithms, but it opens a fully new approach to algorithm design, which by some authors recently has been paraphrased as *programming by optimization* [16].

Essentially the same task of algorithm configuration and tuning arises when trying to set the parameters of a traffic light control software. Such software in the simplest case may contain only few parameters, for example, the fixed length of the red and green phases for a specific traffic light when a simple marching policy with static durations is assumed. The number of parameters increases on one side if one considers the control by traffic lights of an increasingly complex crossing: the more lanes and directions need to be managed the more parameters the control has even for the most simple policies. Best values of these parameters obviously depend on traffic flows. The problem of the setting of parameters becomes more relevant if more complex and sophisticated traffic light control algorithms are considered. One such example is the swarm-based policy selection algorithm developed by the University of Bologna within the COLOMBO project, where a policy selection algorithm is put on top of several policies for defining the specific traffic light control policy that is used in dependence of the traffic density. Even for simple crossings, this algorithm has a large number of a few tens parameters that need to be appropriately set to optimize performance. For details of the traffic light control algorithms we refer to deliverables D2.2 and D2.3 [8, 9].

The main goal of work-package **WP3** is to transfer the methodology and the associated benefits of automatic algorithm configuration for the design and development of optimization algorithms [16] to the design and tuning tasks in the COLOMBO project and, in particular, to the design and tuning tasks for novel traffic light control algorithms.

Traffic light control algorithms just as many other algorithms can have different design choices and the control strategies usually depend on the setting of specific, numerical parameters. While for optimization algorithms the solution quality is evaluated by cost or profit measures, in traffic light control the evaluation is based on metrics that rate the quality of the traffic flow, waiting times, the emission caused over a specific time period [7]. It is therefore clear that the two areas are clearly linked and encourage the transfer of techniques that have been proven successful in one area to the other one. WP3 of the COLOMBO project is the first attempt we are aware of that examines the applicability and the potential advantages of

automatic algorithm configuration in the configuration and tuning of traffic light control algorithms. A successful accomplishment of this transfer will result in the following benefits. First, the reduction of time and human intervention necessary for the design and fine-tuning of the algorithms under development. Second, a better adaptation of the traffic light control software to the specifics of the traffic situation at the traffic lights where the control algorithms are deployed. Third, the automatization of the fine-tuning process using simulation-based optimization. At the very least, we believe that the methodology developed here, will provide an alternative tool to traffic engineers to support their tasks.

1.1 Objectives

The task 3.2 of WP3 focuses on the offline configuration and tuning of traffic light control software proposed in the COLOMBO project.[1] The objectives that, in particular, have been followed in this task and deliverable were the following.

- Improvement and publication of an open source version of the tuning tool kit developed in task 3.1 (the prototype has been described in Deliverable D3.1 [10]).
- Analysis of currently available automatic algorithm configuration software to identify the most suitable candidates for the tuning and configuration tasks in the COLOMBO project.
- Analysis and improvement of the configuration software that has been identified as the most suitable candidate.
- Adaptation of the automatic configuration software and the traffic light control software to usage on our computing cluster, execution of automatic configuration on the traffic light control software and analysis of the tuning performance.

Deliverable D3.2 reports on the progress made towards these objectives. In addition to the above mentioned objectives, we started also the development of a new automatic configuration software that should allow to obtain an even better performing automatic tuner than we have currently available. However, this is still ongoing and rather preliminary work and therefore we decided not to report on it in the present deliverable.

1.2 Structure

The deliverable D3.2 is structured as follows. In Section 2, we give relevant background on automatic algorithm configuration and describe in more detail the automatic algorithm configuration software that is most used in the experimental parts of this deliverable. In Section 3,

[1] More specifically, in the description of work of the COLOMBO project, the task 3.2 to which this deliverable relates was formulated as follows. *A traffic light system can be seen as an agent whose individual behaviour is defined by local decision rules and their parameters. This task (carried on by ULB with the help of UNIBO) will focus on the development, implementation, and testing of procedures that allow optimizing the definition of the local decision rules and their parameters in an automatic, offline fashion. The sub-tasks comprise the following: (i) algorithmic methods for the automatic tuning of algorithms will need to be adapted and tested for a distributed agent environment; (ii) new approaches for making the automatic configuration and tuning process more effective will be developed.*

we describe the analysis and the improvement of the software that is related to the automatic algorithm configuration tasks in the COLOMBO project. This includes (i) improvements to the tuning tool kit and its publication under GNU public license (see Section 3.1); (ii) a comparison of available automatic configuration software on standard configuration benchmarks (see Section 3.2); (iii) an analysis of the automatic configuration software, the `irace` package, which we identified for further usage in the COLOMBO project (see Section 3.3); (iv) and the improvement of various aspects of the `irace` package (see Section 3.4). While the developments that are described in Section 3 have used for the evaluation benchmark configuration tasks that arise in the tuning of optimization algorithms, we adapted and applied the `irace` software in a next step to the tuning tasks that arise in the COLOMBO project. In Section 4, we describe the steps taken to do so and the results that we obtained by tuning the traffic light control software SWARM 2 (see Deliverable D2.3 [9]), developed by the University of Bologna. We finish in Section 5 with a summary of the results.

2 Automatic Algorithm Configuration

2.1 Context

Automatic algorithm configuration methods have enhanced the design and development process of high-performing optimization algorithms [4, 14, 16]. Although automatic algorithm configuration methods, which we also call *configurator* or *tuner* in what follows, have been used so far only by a rather small number of researchers for the development of optimization software, the results obtained so far when exploiting this type of techniques are impressive. It includes the design of new state-of-the-art algorithms [13, 26, 28], the improvement of the performance obtained with widely used standard software [17, 24], or the usage of automatic algorithm configuration tools in the design and development process of algorithms [32, 30].

Automatic algorithm configuration can itself be seen as a stochastic optimization problem, where performance optimizing parameter settings need to be identified. The difficulties for this problem arise due to the following issues. First, the performance of the target algorithms under specific parameter settings can only be measured by actually executing them, a process that may take considerable time. Second, the evaluation of the algorithms is usually stochastic: the results of the algorithms depend either on the particular problem instance that is tackled (in traffic situations, say, the specific data defining the situations such as arrival times, density, flow, speed, disturbances etc.) or the algorithm to be configured itself relies on occasional randomized choices during the search process. Third, the algorithms typically have different types of parameters, ranging from categorical ones that typically define alternative algorithm design choices, ordinal ones that are often related to categorical ones but where parameter choices may be ordered by secondary criteria (for example, size, speed etc.), and numerical parameters that may be, for example, integer or real-valued. Often, numerical parameters arise only for specific choices of other, usually categorical parameters.

Automatic algorithm configuration software is usually developed for off-line configuration. Off-line configuration resembles the classical algorithm design and development process, where an algorithm is developed and engineered with a specific application in mind and considering a specific class of target problem instances that can be described, for example, by specific instance sizes, instance characteristics, or available computation times for their solution. The resulting algorithm configuration problem from this situation has been formally defined in Deliverable D3.1 [10], Section 2.1. The overall process followed in off-line configuration can be summarized as given in Figure 1 (see also Deliverable D3.1 [10], page 8). The off-line configuration process also needs the definition of a performance measure. For optimization algorithms, this is often the solution quality reached after a specific computation time, or the time it takes to reach specific bounds on the solution quality. However, also other goals have been used such as the optimization of an algorithm's anytime behavior [29]. In the traffic light control context, the performance measure to be optimized could be any of the measures that are explained, for example, in Deliverable D5.3 [11] of the COLOMBO project. (The notion of instances, the traffic light control algorithm to be configured, and the performance criteria relevant for tuning the traffic light control algorithm are explained in more detail in Section 4.)

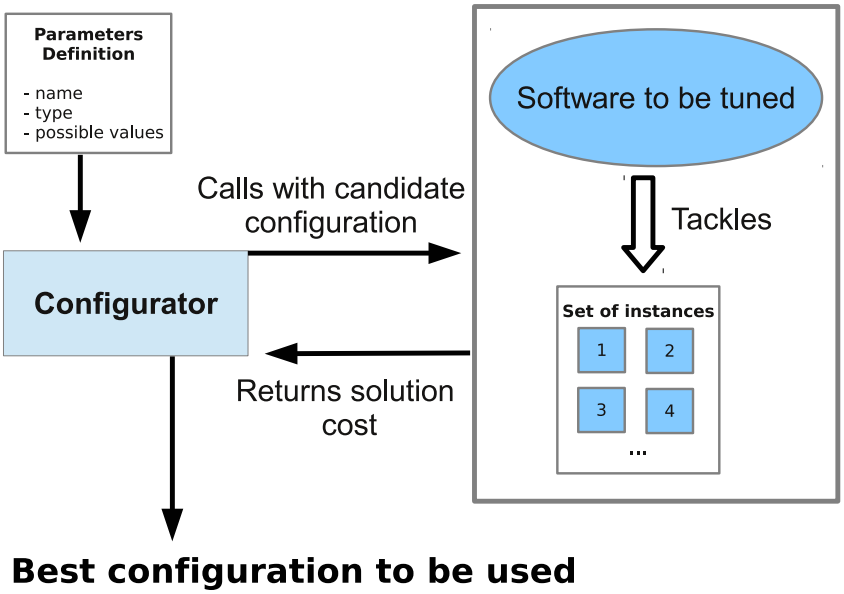


Figure 1: Summary of the relationship between configurator, optimizer, and training instances during the training phase of an automatic algorithm configuration tool.

2.2 The configuration software considered

As we have explained in Deliverable D3.1 [10], the main publically available automatic configuration software that we have considered for the COLOMBO project are the `irace` package [27], ParamILS [21, 20], and the Sequential Model-based Algorithm Configuration (SMAC) method [18]. The three packages have the advantages that (i) they are publically available, (ii) they offer mature software for actual automatic configuration of algorithms, (iii), they are applicable to configuration tasks that are stochastic, require multiple instances, and have all usual different types of parameters (that is categorical, ordinal, numerical but also conditional ones, that is, parameters that only arise if other parameters take specific values). These three configuration software packages have also been interfaced with the Tuning Took Kit (see Deliverable D3.1 [10] and Section 3.1). A comparison of the three software packages on some configuration tasks is given in Section 3.2.

The three software packages rely on rather different paradigms for steering the search. The `irace` software uses iterated racing for identifying high performance configurations. It iteratively repeats the following three steps until the tuning budget is exhausted: (i) sample new configurations according to a particular sampling distribution, (ii) select the best configurations by means of racing, and (iii) update the sampling distribution biasing the sampling towards the best configurations. ParamILS is an iterated local search algorithm that uses a first-improvement local search in the parameter space and occasional perturbations of current configurations in the case the local search is deemed to be stuck in local optima. A potential disadvantage of ParamILS is that it requires a discretization of numerical parameters as it essentially treats all parameters as categorical ones. SMAC is a model-based search algorithm. It uses a surrogate model of algorithm performance on already solved instances. This surrogate model is used to predict the performance of new algorithm configurations and only the

most promising configurations according to this prediction are actually executed on the target algorithm to be configured.

2.3 A detailed description of iterated racing

As we will explain later, the `irace` software was selected to perform the traffic light control software in the context of the COLOMBO project. As specific details of the `irace` software have been analyzed and improved in the context of the COLOMBO project, in what follows we give a more precise description of the details of the `irace` software.

2.3.1 Racing and iterated racing

The racing approach [31, 5] is effective at identifying the best configuration from a given initial set of configurations. It proceeds by step-by-step testing the set of candidate configurations on new example instances of the problem being tackled. At each step, it is checked by statistical testing whether differences exist among the candidate configurations and if it is so, then inferior candidate configurations are eliminated from further testing. The first racing algorithm for automatic algorithm configuration has been F-race [5]. At each step of the race, it applies the non-parametric Friedman test (hence, the `F` in the name of the method) to check whether a difference exists among the current set of configurations. If the null hypothesis of the Friedman test is rejected (that is, there is a difference among the configurations), the ones performing worse than the best configuration are eliminated from the race by applying Friedman post-test [12]. Instead of the Friedman test as "elimination test", also other tests may be performed [4].

The initial set of candidate configurations may be generated by randomly sampling the parameter space. Unfortunately, the number of configurations that are to be sampled in this way would need to be rather large to ensure that at least one very high-performing configuration is sampled with a significant probability and it would result impractical especially with large parameter spaces. Therefore, I/F-Race was proposed [3, 6] extending F-Race. I/F-Race applied in each iteration one F-Race to a set of candidate that depend on the results of the previous races. Candidate configurations are generated by sampling a probability model P_X thatis defined over the parameter space X. Without prior knowledge or specific initial configurations such as default configurations, the sampling in the first iteration is done following a uniform distribution. In the following iterations, categorical parameters are sampled according to a discrete probability function and numerical parameters are sampled according to a normal distribution. The `irace` software that we developed is a generalization of I/F-Race, which remains one of the possible racing methods usable in `irace`.

2.3.2 Iterated racing

In this section, we describe the search process and the sampling model as implemented in the `irace` package.

Algorithm 1 outlines the overall iterated racing algorithm. Iterated racing requires as input a set of instances ($\mathcal{I}$), a parameter space (X), a cost function ($\mathcal{C}$), and a tuning budget (B).

For the setup of the search process, iterated race requires first an estimation of the number of iterations N^{iter} (that is, the number of races) that it will execute. The default setting of N^{iter} depends on the number of parameters $Nparam$ and is given by $N^{iter} = \lfloor 2 + \log_2 N^{\text{param}} \rfloor$. Each

Algorithm 1 Iterated Racing

Require: $\{I_1, I_2, \dots\} \sim \mathcal{I}$,
parameter space: X,
cost measure: $\mathcal{C}\colon \Theta \times \mathcal{I} \to \mathbb{R}$,
tuning budget: B

$\Theta_1 \sim \mathsf{SampleUniform}(X)$
$\Theta^{\text{elite}} := \mathsf{Race}(\Theta_1, B_1)$
$i := 2$
while $B_{\text{used}} \leq B$ **do**
  $\Theta^{\text{new}} \sim \mathsf{Sample}(X, \Theta^{\text{elite}})$
  $\Theta_i := \Theta^{\text{new}} \cup \Theta^{\text{elite}}$
  $\Theta^{\text{elite}} := \mathsf{Race}(\Theta_i, B_i)$
  $i := i + 1$
  if $i > N^{iter}$ **then**
    $N^{iter} := i$
  end if
end while
Output: Θ^{elite}

iteration applies one race using a computation budget $B_i = (B - B_{\text{used}})/(N^{iter} - i + 1)$, where $i = 1, \dots, N^{iter}$. Here, B_{used} is the computation budget that has already been used. The computation budget is typically specified as the maximum number of times candidate configurations are executed on an instances of the problem to be solved or as the maximum total runtime spent by algorithm configurations. Each race starts from a set of candidate configurations Θ_i whose number is computed as $|\Theta_i| = N_i = \lfloor B_i/(\mu + \min(5, i)) \rfloor$. The number of generated candidate configurations decreases with the number of iterations. This results in more evaluations per configuration being performed in later iterations. The parameter μ further gives the user the possibility to influence the ratio between budget and number of configurations, which also depends on the number of iterations i. The intuition of this setting is that based on the fact that configurations that are generated in later iterations will be more similar, and therefore it is also expected that more evaluations will have to be done to identify with reasonable statistical precision the best ones. To avoid having too few configurations in a race, we however do not consider more than five iterations for computing this setting.

In the first iteration, the candidate configurations are generated by uniformly sampling the parameter space X. In addition, other configurations may be given directly to `irace` such as algorithm default configurations or configurations deemed to be particularly promising by the algorithm designer. Upon the start of a race, each configuration is evaluated by running the algorithm on the first instance and using a cost measure $\mathcal{C}$. Configurations are then evaluated step-by-step on subsequent instances until a number of instances have been seen (T^{first}). Then, a statistical test is performed on the results. If enough statistical evidence is gathered to identify some candidate configurations as performing worse than at least another configuration, the worst configurations are removed from the race, while the others, the *surviving* candidates, are run on the next instance. A statistical test is done every T^{each} instances. As default we have $T^{\text{each}} = 1$, but there may be situations where it is useful to perform each test only after the configurations have been run on a number of instances. The race continues until the computation

budget in the current iteration is not enough to test all remaining candidate configurations on a new instance ($B_i < N^{\text{surv}}$), or when at most $N^{\min}$ configurations remain, $N^{\text{surv}} \leq N^{\min}$. Upon termination of a race, surviving configurations are assigned a rank r_z either according to the sum of ranks in the case of usage of a non-parametrical test or the mean cost, usually when applying a parametric elimination test such as Student's t-test. The $N_i^{\text{elite}} = \min(N^{\text{surv}}, N^{\min})$ configurations with the lowest rank are selected as the set of elite configurations Θ^{elite}.

In the next race, $N_i^{\text{new}} = N_i - N_i^{\text{elite}}$ new candidate configurations are generated. To generate a new configuration, first a parent configuration θ_z is chosen from the elite configurations Θ^{elite} with a probability

$$p_z = \frac{N_i^{\text{elite}} - r_z + 1}{N_i^{\text{elite}} \cdot (N^{\text{elite}} + 1)/2} \ , \tag{1}$$

which is proportional to its rank r_z: higher ranking configurations thus have also a higher probability of being chosen as "parent".

Next, a new value is sampled for each parameter X_d, $d = 1, \ldots, N^{\text{param}}$, based on a distribution that its associated to each parameter of θ_z. The parameters are assigned values in an order that is determined by the dependency graph of conditions: unconditional parameters are sampled first and parameters that are conditional on other parameters are sampled next if the condition is satisfied, and so on. If a conditional parameter was disabled in the parent configuration and it becomes enabled in the new configuration due to the sampling, then this parameter is assigned a value that is chosen randomly according to a uniform distribution, just as in the initialization phase.

If X_d is a numerical parameter defined within the range $[\underline{x}_d, \overline{x}_d]$, a value is sampled using the truncated normal distribution $\mathcal{N}(x_d^z, \sigma_d^i)$.[2] The mean of the distribution x_d^z is the value of parameter d in elite configuration θ_z. The parameter σ_d^i is set to $(\overline{x}_d - \underline{x}_d)/2$ in the first iteration, and then it is lowered iteration by iteration following

$$\sigma_d^i := \sigma_d^{i-1} \cdot \left(\frac{1}{N_i^{\text{new}}} \right)^{1/N^{\text{param}}} \tag{2}$$

This way of reducing σ_d^i iteration by iteration allows to sample values that are increasingly closer to the value in the parent configuration and, thus, to focus the search around the best parameter settings found. Roughly speaking, the multi-dimensional volume of the sampling region is reduced by a constant factor at each iteration, but the reduction factor is higher when sampling a larger number of new candidate configurations (N_i^{new}).

If the numerical parameter is of integer type, we round the sampled value to the nearest integer. Parameters of ordinal type are encoded as integers.

If X_d is a categorical parameter with levels $X_d \in \{x_1, x_2, \ldots, x_{n_d}\}$, a discrete probability distribution $\mathcal{P}^{i,z}(X_d)$ is used for sampling values. In the first iteration ($i = 1$), $\mathcal{P}^{1,z}(X_d)$ is uniformly distributed. In subsequent iterations, the discrete probability distribution is updated before sampling as

$$\mathcal{P}^{i,z}(X_d = x_j) := \mathcal{P}^{i-1,z}(X_d = x_j) \cdot \left(1 - \frac{i-1}{N^{iter}}\right) + \Delta\mathcal{P} \tag{3}$$

[2] We use the `msm` package [22] to sample from a truncated normal distribution.

where

$$\Delta\mathcal{P} = \begin{cases} \dfrac{i-1}{N^{iter}} & \text{if } x_j = x_z \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Finally, the new configurations generated after sampling inherit the probability distributions from their parents, and a new race is launched with the union of the new configurations and the elite configurations.

The `irace` algorithm stops either once the computation budget is exhausted ($B_{\text{used}} > B$) or if the number of candidate configurations that have to be generated at the start of an iteration is smaller or equal to the number of elite candidates ($N_i \leq N_i^{\text{elite}}$). If the iteration counter i reaches the initially estimated number of iterations N^{iter} but enough budget remains to start yet another race, we increase N^{iter} and continue.

3 Automatic Configuration Tools: Comparison, Analysis and Improvements

In this section, we describe the analysis of the automatic algorithm configuration software and the improvements and updates made to the this software within the COLOMBO project. In particular, we have completed the tuning tool kit and made it publically available (see Section 3.1). We have compared the performance of automatic algorithm configuration software, in particular, `irace`, ParamILS, and SMAC on few selected benchmark configuration tasks; this analysis is reported in Section 3.2. As a next step, we have analyzed the impact of specific setting of the parameters of the automatic configuration software on its performance as reported in Section 3.3. Finally, we describe some of the improvements of the `irace` software, which have been done in the COLOMBO project, in Section 3.4.

3.1 Tuning tool kit

In what follows we will first briefly recall the purpose and the usage of the tool kit.

3.1.1 Purpose

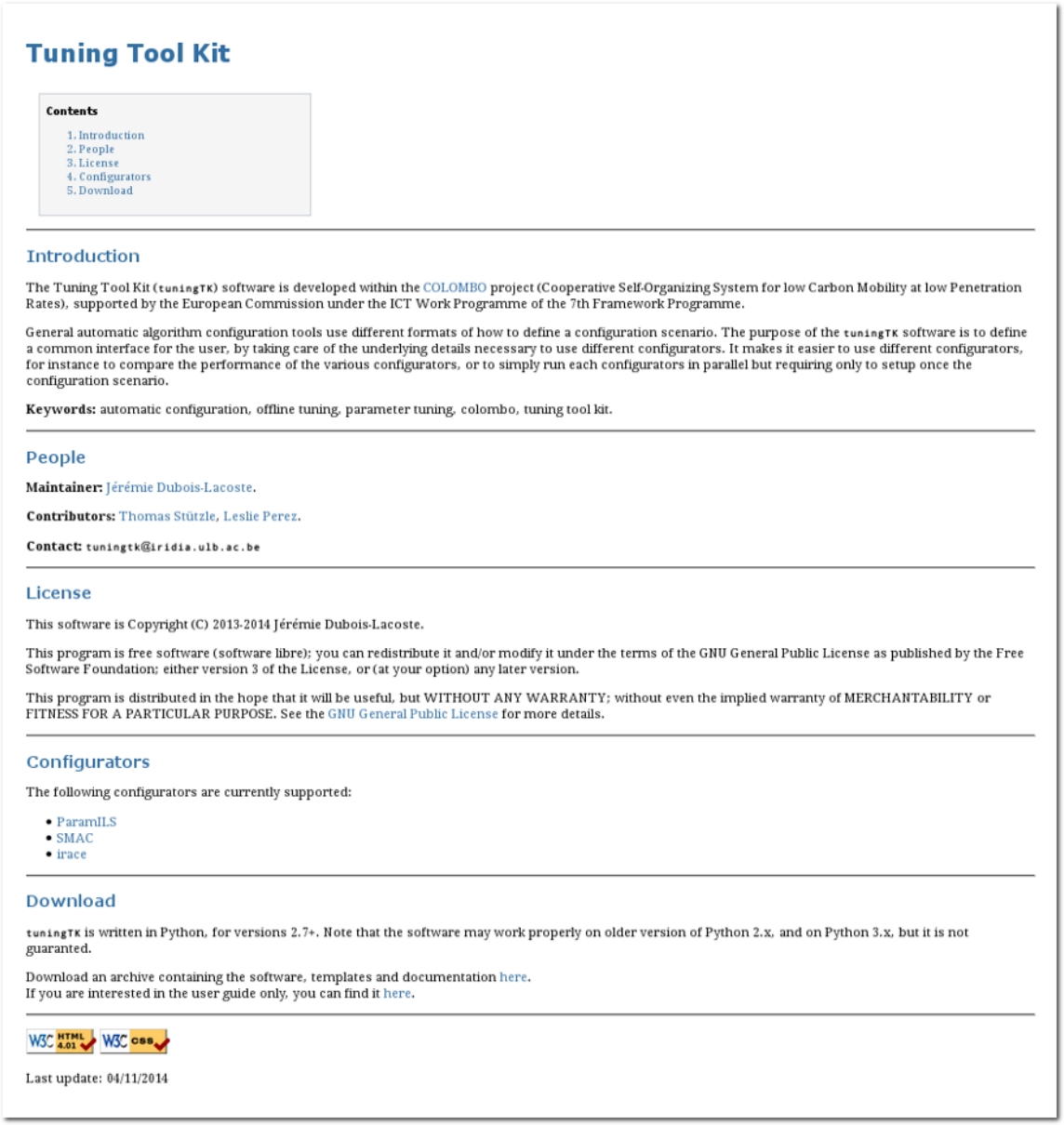
The currently available, general automatic algorithm configuration tools usually use different formats of how to define a configuration scenario, probably because these methods have only recently been devised, and that its usage is still largely limited to research. This results in various ways of how to interact with these tools. This concerns mainly the way how to define the parameters to be configured, but also the general settings of the configurator (such as the experimental budget to be used for the configuration), and the software interface to the algorithm to be configured. As a consequence, a user must spend time learning how to use each configurator, and even then, cannot reuse the same set-up across the different configurators.

The purpose of the tuning tool kit is to define a common interface for the user to define the parameters, and to take care of the underlying details necessary to use different configurators in a transparent way. Thus, it makes it easier to use several configurators, to compare the performance of the various configurators, or to simply run each of the configurators in parallel but requiring only to setup once for all the configuration scenario. This is specially useful because for a specific configuration task it is usually a priori unknown which configurator will reach the best overall results. In addition, it may also ease the comparison of different configurators to better understand their relative advantages and weaknesses on different optimization problems. Due to the novelty of the automatic configuration research field, this task has been rarely done in the literature as of today.

Currently, the tuning tool kit supports the configurators, which have been shortly presented in Section 2.2: `irace`, SMAC and ParamILS. More configurators might be added if they become relevant in the future for the COLOMBO project itself or as a generic tool provided by the project. Each of these configurators must be installed independently on the system. For more details on the inner working of the tuning tool kit, the reader can refer to Deliverable 3.1 [10].

3.1.2 Public Release of tuning tool kit

Some details have been corrected and tested in the prototype and the tuning tool kit has been publically released in May 2014. It is available on a dedicated webpage at `http://iridia.ulb.ac.be/tuningTK`, which is given in Figure 2. The tuning tool kit release has also been announced through dedicated mailing, on the COLOMBO webpages and the COLOMBO LinkedIn group.



Tuning Tool Kit

Contents

1. Introduction
2. People
3. License
4. Configurators
5. Download

Introduction

The Tuning Tool Kit (tuningTK) software is developed within the COLOMBO project (Cooperative Self-Organizing System for low Carbon Mobility at low Penetration Rates), supported by the European Commission under the ICT Work Programme of the 7th Framework Programme.

General automatic algorithm configuration tools use different formats of how to define a configuration scenario. The purpose of the tuningTK software is to define a common interface for the user, by taking care of the underlying details necessary to use different configurators. It makes it easier to use different configurators, for instance to compare the performance of the various configurators, or to simply run each configurators in parallel but requiring only to setup once the configuration scenario.

Keywords: automatic configuration, offline tuning, parameter tuning, colombo, tuning tool kit.

People

Maintainer: Jérémie Dubois-Lacoste.

Contributors: Thomas Stützle, Leslie Perez.

Contact: tuningtk@iridia.ulb.ac.be

License

This software is Copyright (C) 2013-2014 Jérémie Dubois-Lacoste.

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. See the GNU General Public License for more details.

Configurators

The following configurators are currently supported:

- ParamILS
- SMAC
- irace

Download

tuningTK is written in Python, for versions 2.7+. Note that the software may work properly on older version of Python 2.x, and on Python 3.x, but it is not guaranted.

Download an archive containing the software, templates and documentation here.
If you are interested in the user guide only, you can find it here.

Last update: 04/11/2014

Figure 2: Screenshot of the webpage `http://iridia.ulb.ac.be/tuningTK`, where the tuning tool kit is available for download.

3.2 Comparison of automatic configuration software

As a next step, the configurators `irace`, ParamILS, and SMAC were compared using few configuration benchmarks. In this section, we first detail the configuration scenarios that were used for comparing different automatic configuration software and that was also used later to analyse the `irace` performance on the software's own parameters.[3] Each configuration scenario has a target algorithm, a set of training and test instances and an evaluation budget allowed for the configurator. The data of the configuration scenarios is available at the supplementary information page (`http://iridia.ulb.ac.be/supp/IridiaSupp2013-008/`).

3.2.1 Configuration benchmark scenarios

ACOTSP is a software package that provides various ant colony optimization (ACO) algorithms [37] for solving the Traveling Salesman Problem (TSP). The ACOTSP scenario requires the setting of 11 parameters of ACOTSP, three categorical, four integer and four continuous. The training set of instances is composed of ten random Euclidean TSP instances for each of the number of 1000, 1500, 2000, 2500 and 3000 cities; the test set has 50 instances of the same sizes. The goal is to minimize tour length. For all instances the optimal solutions are known and therefore we can give the results as the percentage deviations from these optimal solutions. The maximum execution time of a run of ACOTSP is 20 seconds and the total configuration budget is 5000 runs of the ACOTSP software.

SPEAR is an exact backtrack-style solver for SAT problems [2] available from `http://www.domagoj-babic.com/index.php/ResearchProjects/Spear` The SPEAR scenario requires the setting of 26 categorical parameters of SPEAR. The training and the test set are composed of 302 SAT instances each from the SAT configuration benchmark "Spear-swv". The goal is to minimize mean algorithm runtime. The maximum execution time for each run of SPEAR is set to 300 seconds and the total configuration budget is 10000 runs of the SPEAR solver.

MOACO is a framework of multi-objective ACO algorithms [28]. The MOACO scenario requires the configuration of 16 parameters: 11 categorical, one integer and four real. The training and the test set are composed of 10 instances of 500, 600, 700, 800, 900, 1000 cities each. The goal is to optimize the quality of the Pareto-front approximation as measured by the hypervolume quality measure [38]. The hypervolume is to be maximized, however, for consistency with the other scenarios, we plot the negative normalized hypervolume, which is to be minimized. The maximum execution time of each run of MOACO is set to $4 \cdot (instance\_size/100)^2$. The total configuration budget is 5000 runs.

3.2.2 Experimental Setup

In this and the following sections, each experiment consists of 20 trials for each of the configurators that is tested. This gives as a result 20 configurations for each configuration scenario and

[3] Note that the automatic configuration software itself is actually an optimization algorithm for stochastic, non-convex mixed-variable optimization. As such, also the configuration software itself has parameters that in turn influence the software's behavior. One may wonder whether by this fact one may actually need to tune the configurator's parameters. The answer is, yes, the configurator parameters have an influence on performance and one may recursively argue for even more levels of "meta-meta-. . ."-tuning. However, as one reaches higher meta-levels one is faced with effects of diminishing returns and empirically it has already shown in some preliminary experiments that further performance gains may be minor [20].

for each configurator. For each of these configurators and the 20 so obtained configurations for each configurator we compute the average performance on the test set. Each of the configurators is tested using its default parameter settings. In the case of the `irace` package, we repeat each experiment using either the F-test (and its associated post-hoc tests) or the Student t-test without multiple test correction as the statistical test of eliminating candidates.[4] This is done as in the final testing we compute average performance and there is a subtle difference in using either the Student t-test of the F-test: While the statistic that is used in the Student t-test is based on average performance (and, hence, the same as in the evaluation on the test set), in the F-test the statistic is based on the ranking of the candidates and, thus, absolute differences in the performance of the candidates are not considered. Depending on the variability of the data, there may therefore be differences between using either test. The experiments were executed on a cluster running Cluster Rocks GNU/Linux 6.0. The experiments involving the ACOTSP scenario were executed on an AMD Opteron 6128 with 8 cores of 2 GHz and 16 GB RAM. The ones involving the SPEAR scenarios were executed on an AMD Opteron 6272 with 16 cores of 2.1GHz and 64GB RAM.

ACOTSP scenario

The ACOTSP scenario requires to identify configurations that reach best possible solution quality of an ACO algorithm after 20 seconds of computation time. In Figure 3 we give a box-plot of the results obtained by `irace` (using either F-test or t-test for the elimination test), ParamILS, and SMAC on this scenario. These box-plots clearly indicate that on this scenario the `irace` package obtains the best results, which are also statistically significantly better than those of ParamILS and SMAC. This is true independent of whether the Student t-test or the F-test is used as the elimination test.

As a further test, we tried to reduce the tuning budget to 1000 evaluations of candidate configurations. The results, shown in Figure 4, indicate that for this smaller tuning budget the three tuners perform similarly. This is mainly due to the worse performance of `irace` for this smaller tuning budget. On the contrary, the results of paramILS and SMAC are not really much worse than the results they obtain for the larger tuning budget. This indicates that the latter two configurators may suffer from some stagnation behavior as by a higher computational budget they are not able to identify significantly better configurations, differently from `irace`, which clearly benefits from the additional budget.

SPEAR scenario

As a next test, we compare `irace` and paramILS on the SPEAR scenario. This scenario is very different from the ACOTSP scenario in the sense that it only contains parameters that are treated as categorical and that this scenario concerns the minimization of algorithm run-time to a decision and not the optimization of solution quality within some specific budget. As paramILS benefits in such a scenario from specific early pruning techniques for configurations that exploit the fact that they concern minimization of computation time we compare `irace`

[4]The use of multiple test corrections in the Student t-test results, while being statistically more correct, in a non-effective elimination of poor candidates as the power of the test is rather small [4]. Thus, the search process is not aggressive enough towards identifying high quality candidates. While not using multiple test corrections results in obtaining a more heuristic search process, it, however, shows to be rather effective.

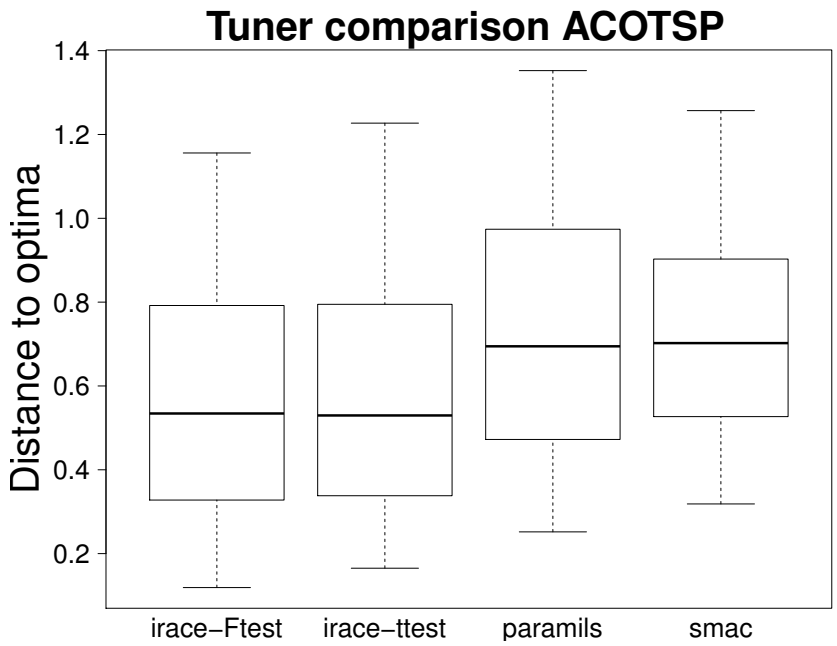


Figure 3: Box plots of the mean performace over the test instances of 20 configurations obtained by `irace` using either the F-test or the Student t-test as elimination test, ParamILS, and SMAC on the ACOTSP configuration scenario using a tuning budget of 5000.

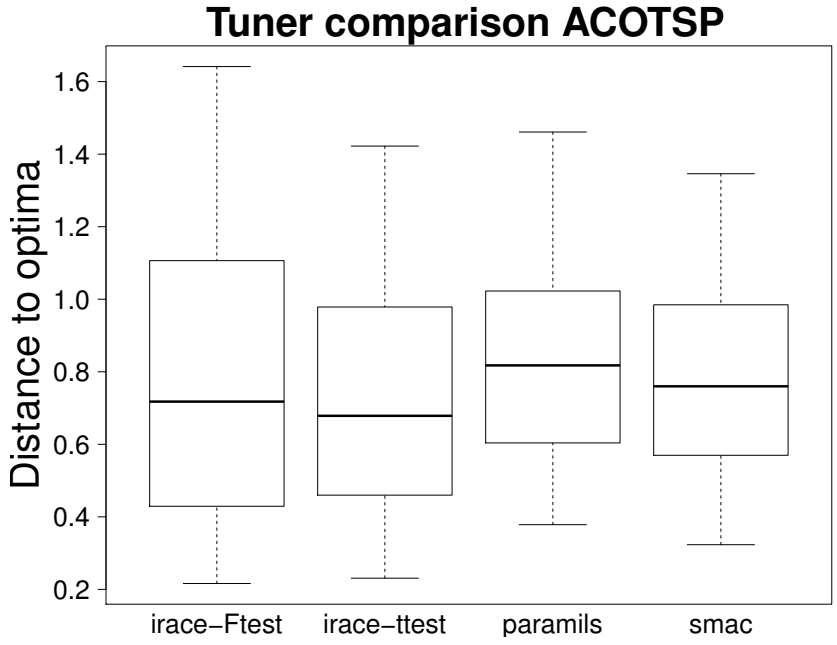


Figure 4: Box plots of the mean performace over the test instances of 20 configurations obtained by `irace` using either the F-test or the Student t-test as elimination test, ParamILS, and SMAC on the ACOTSP configuration scenario using a tuning budget of 1000.

and paramILS based on the same number of candidate configuration evaluations. (In fact, this choice is also justified by the target tuning runs of the COLOMBO project: The early pruning techniques applied in scenarios where computation time is to be minimized are not effective (or applicable at all) in COLOMBO scenarios.) The results given in Figure 5 indicate that on

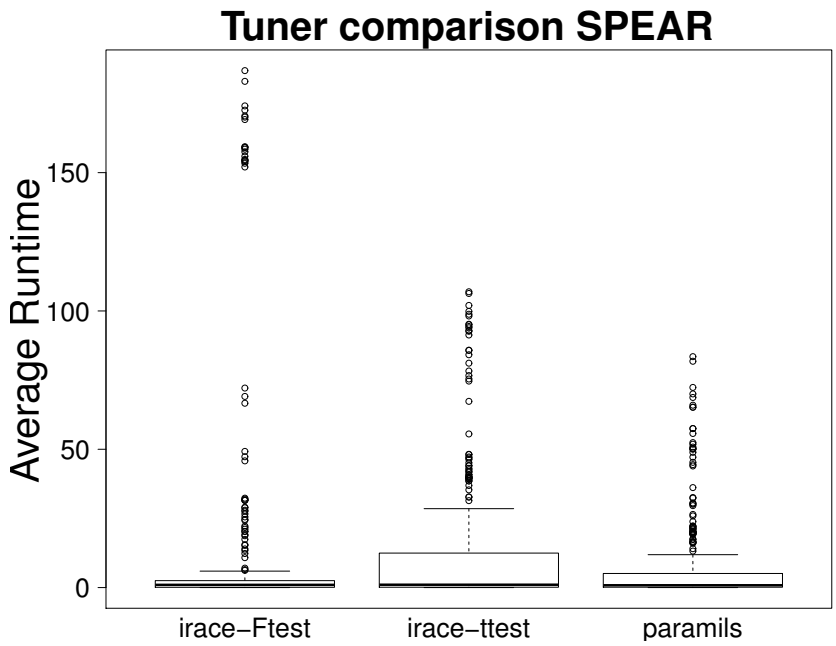


Figure 5: Box plots of the mean performance over the test instances of 20 configurations obtained by `irace` using either the F-test or the Student t-test as elimination test and ParamILS on the SPEAR configuration scenario using a tuning budget of 10000 evaluations.

this scenario paramILS has better slightly performance than `irace`. In fact, the performance differences

MOACO scenario

The MOACO scenario requires the configuration of a multi-objective algorithm. Automatic configuration tools, however, require the usage of a scalar number of identifying the best configuration. To adapt automatic configuration tools for the tuning of multi-objective problems, we have proposed in previous research a methodology that exploits performance indicators for multi-objective optimization [28]. This methodology relies on a on-the-fly normalization of results that are obtained by various configurations on a same instance. While with `irace` this can be easily accomplished, ParamILS and SMAC cannot readily handle such a task and would require major modifications or the tuning scenarios would have to be adapted in unnatural ways by precomputations. Hence, no comparisons were run on this scenario, but this scenario was used for a more in-depth analysis of `irace` reported in the next section.

Recommendation of configurator usage

The comparison of the configurators has indicated that on one side, their performance may differ significantly. For example, while `irace` was clearly best performing on the ACOTSP scenario once given enough budget, on the SPEAR scenario ParamILS obtained better performance than `irace`. A further difference among configurators is whether they can readily be applied to different tasks. Here, `irace` has advantages if, for example, a multi-objective configuration is to be applied, as it can directly be extended to this task. This task is also relevant for the COLOMBO project: If the traffic light control software is to be optimized

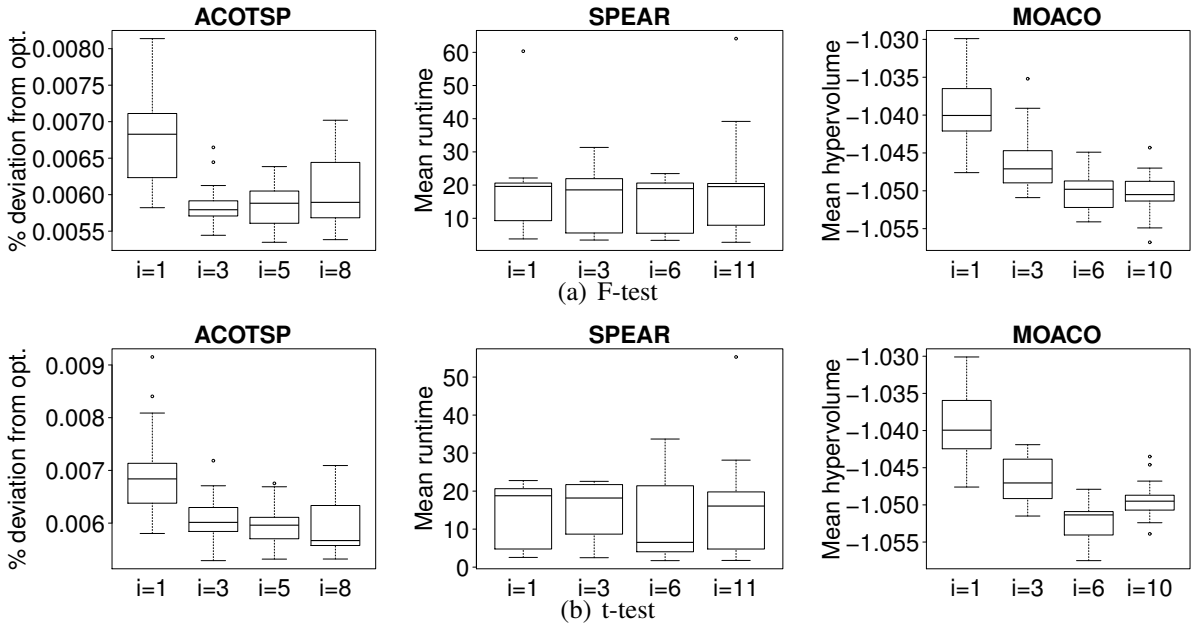


Figure 6: Box plots of the mean performace over the test instances of 20 configurations obtained by `irace` using $N^{iter} \in \{1, 3, \text{default}, \text{large}\}$.

not only considering one single performance measure but considering multiple ones (such as emission minimization in addition to average waiting time minimization). Finally, a relevant feature to consider is also the possible parallelization of automatic configuration software. So far, paramILS and SMAC have mainly used single-threaded computation but parallelization has been explored through the usage of mainly parallel runs [19]. Although a parallel version of SMAC has also been explored, it is not publicly available. Differently, `irace` uses parallelization using protocols such as MPI, which allows to execute a single configuration run using multiple CPU cores (experiments have been done with up to 200 computing cores in parallel so far). This possibility is particularly important if we consider the high computation times that can be expected from the traffic simulation scenarios that will be considered in the COLOMBO project in the final year.

Based on these reasons, we explored and improved further the `irace` software. These developments are described in the following two sections.

3.3 Analysis of `irace`

In this section, we examine the impact of five parameters of `irace` on the performance of the final algorithm configuration found in the configuration process [33]. In addition to the scenarios ACOTSP and SPEAR used in Section 3.2.1, here we consider also the MOACO scenarios, which were executed on an AMD Opteron 6272 with 16 cores of 2.1GHz and 64GB RAM. For the imact different parameter settings have, we also check the statistical signifciance of the differences by the use of the Wilcoxon signed-rank test.

Table 1: Wilcoxon signed-rank test p-values comparing the mean performance over the test instances of configurations obtained by `irace` using $N^{iter} \in \{1, 3, \text{default}, \text{large}\}$.

	default vs. large			default vs. 3			default vs. 1		
	ACOTSP	SPEAR	MOACO	ACOTSP	SPEAR	MOACO	ACOTSP	SPEAR	MOACO
F-test	0.33	0.4304	0.8695	0.7562	0.7562	0.0003948	$1.907e^{-5}$	0.7285	$1.907e^{-6}$
t-test	0.2943	0.498	0.0007076	0.7285	0.4304	$1.907e^{-6}$	0.0002098	0.5459	$1.907e^{-6}$

3.3.1 Number of Iterations.

The number of iterations (N^{iter}) determines the number of iterations over which `irace` is run and it has a significant impact on the search behavior of `irace`. The more iterations are given the fewer configurations are used in each iteration. Additionally, the number of newly sampled configurations is also reduced when increasing N^{iter} as the number of elite configurations remains the same. The main effect is that an increase of N^{iter} intensified the search by splitting the budget in short races. Less iterations, on the other hand, diversify stronger the search. The default number of iterations of `irace` depends on the number of parameters. We increase this value to

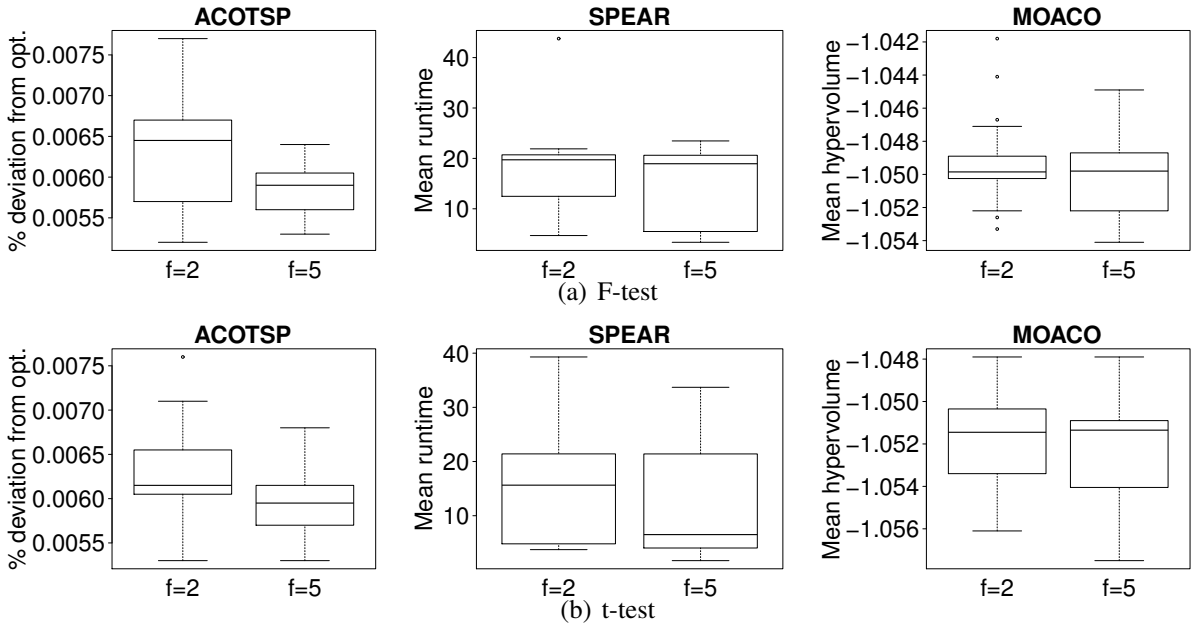
$$N^{iter} = \lfloor 2 + 2 \cdot \log_2(N^{param}) \rfloor \tag{5}$$

and we refer to this setting as "large" in the following. Additionally, we use two constant values for the parameter: $N^{iter} = 3$ and $N^{iter} = 1$. The latter actually corresponds to a single race using configurations sampled uniformly at random [3]. In Figure 6, we give the results of the 20 executions of `irace` on the three configuration scenarios and the results of the Wilcoxon test are shown in Table 1. In the SPEAR scenario, none of the differences is statistically significant. Surprisingly, even a race based on a single random sample of configurations ($N^{iter} = 1$) obtains reasonable performance here. Differently, for the MOACO and ACOTSP scenarios, `irace` with $N^{iter} = 1$ is significantly worse than the other settings, as also observed occasionally in previous research [6]. Other differences in the ACOTSP scenario are, however, not statistically significant. In the MOACO scenario, the default setting performs significantly better than $N^{iter} = 3$, while the large setting performs significantly worse than the default only when using t-test.

Overall, the default setting of N^{iter} appears to be reasonably robust. Nonetheless, the number of iterations has an impact on the quality of the final configurations and the adaptation of the number of iterations to the configuration scenario may be useful to improve `irace` performance.

3.3.2 First elimination test.

The elimination of candidates during the race allows `irace` to focus the search around the best configurations. The number of instances that are evaluated before the first elimination test is done is determined by the parameter T^{first}. Here, we analyze the sensibility of `irace` to this parameter. We tested the default setting of ($T^{first} = 5$) and a smaller value of $T^{first} = 2$. The latter settings allows `irace` to more aggressively eliminate configurations. The saved budget may then be used later to sample more configurations, but on the downside good configurations may erroneously be lost more easily. The experimental results are shown in Figure 7. While in the ACOTSP scenario a value of $T^{first} = 2$ seems to worsen performance, in the SPEAR and



(a) F-test

(b) t-test

Figure 7: Box plots of the mean performace over the test instances of 20 configurations obtained by `irace` using $T^{first} \in \{2, 5\}$.

Table 2: Wilcoxon signed-rank test p-values comparing the mean performance over the test instances of configurations obtained by `irace` using $T^{first} = 2$ vs. $T^{first} = 5$.

	ACOTSP	SPEAR	MOACO
F-test	0.01362	0.1231	0.1231
t-test	0.03623	0.5958	0.6477

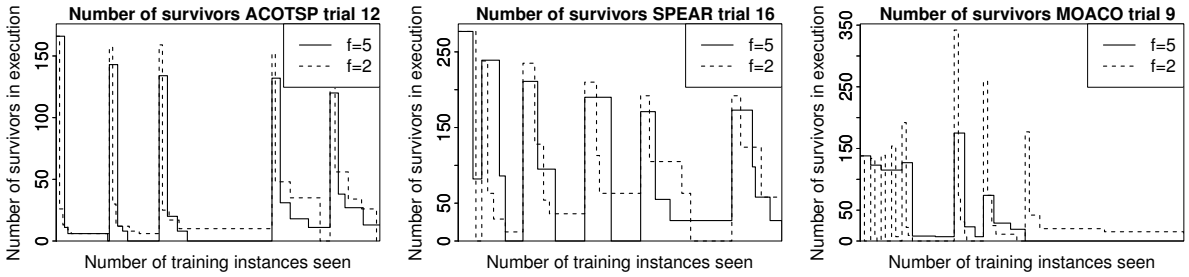


Figure 8: Number of surviving candidates in `irace` using $T^{first} \in \{2, 5\}$ and F-test.

MOACO scenarios no clear differences are detectable. The Wilcoxon paired test in Table 2 supports this analysis.

Our hypothesis was that with a setting of $T^{first} = 2$, poor candidates are eliminated earlier and in later iterations more candidates may be sampled. In order to corroborate this hypothesis, we plot the development of the number of surviving configurations during the search process of `irace` (Fig. 8). The plots show one run of `irace` that is representative for the general behavior.

3.3.3 Maximum number of elite configurations.

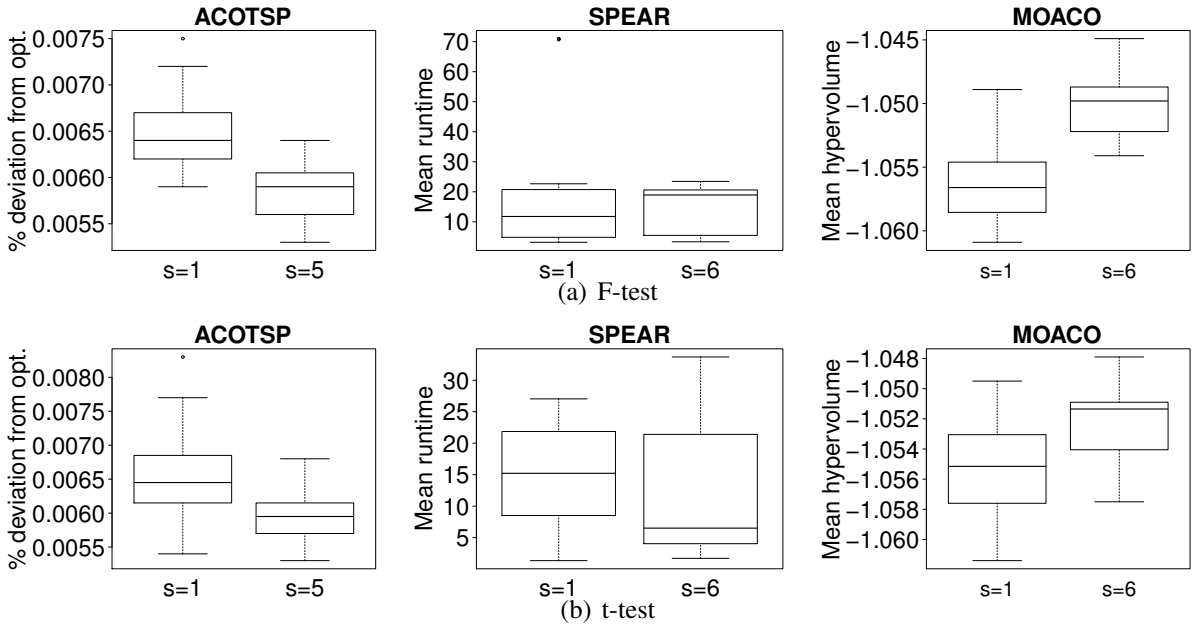
The maximum number of elite configurations (N^{max}) influences the exploration / exploitation trade-off in the search process. If we set $N^{max} = 1$, `irace` generates new configurations only around the best configuration that has been identified so far, while larger values of N^{max} induce a more diversified search. Here, we examine differences that result by setting $N^{max} = 1$ and compare it to the default setting. The experimental results are shown in Figure 9 and the Wilcoxon test p-values in Table 3. While the usage of a single elite configuration worsens significantly the results in the ACOTSP scenario, in the MOACO and SPEAR scenarios no significant performance differences could be observed. Intensifying the search by strongly a strong reduction of the number of elite candidates apparently does not help to improve `irace` performance in any of the configuration scenarios, thus, indicating that the default setting is reasonably adequate.

3.3.4 Statistical test.

The main difference between the F-test (plus post-test) and the Student t-test is that the latter uses directly the quality values returned by the target algorithm, while the former transforms the values into ranks. Hence, the F-test can detect minimal but consistent differences between the performance of the configurations but it is insensitive to large sporadic differences, while the Student t-test is influenced by such outliers. We give the results of the comparison of the two statistical tests in Figure 10 and give in Table 4 the Wilcoxon test p-values for a comparison of the two choices. The upper row of plots shows the average performance of the candidates on the test set and the lower row of plots compares the average performance of the candidates per instance separated for different instances classes or sizes. The results of the Wilcoxon test indicate significant differences only for the MOACO case, where the usage of the t-test leads to better performance. It is interesting, however, to analyze in more detail the SPEAR configuration scenario. While no significant difference with respect to to the average performance (mean runtime) was observed, the configurations obtained by using the F-test in `irace` leads to shorter runtimes on more instances than the configurations obtained by using Student's t-test; however, configurations obtained using Student's t-test performs much better than the configurations obtained by the F-test on the subset of the *hsat* instances. Actually, the F-test configurations give in a statistically significantly larger percentage of shorter runtimes than t-test configurations. This is consistent with the fact that the F-test prefers a lower mean ranking as it is obtained by a better performance on a majority of instances while the t-test improves the mean performance and tends to reduce worst case performance, which in the SPEAR configuration scenario are noticeable by very high runtimes. In this sense, these results confirm earlier observations for different configurators [36, 20].

3.3.5 Statistical test confidence level.

The default confidence level of the `irace` elimination test is 0.95. Larger values mean that the test becomes more strict, so that it takes more evaluations to eliminate configurations; lower values allow eliminating configurations faster, save budget, but incur the danger of removing good configurations due to few unlucky runs. Here, we examine the impact of this parameter on the configuration process by running experiments with confidence levels $\in \{0.75, 0.95, 0.99\}$. We summarize the results in Figure 11 and Table 5. For the ACOTSP scenario, a confidence



(a) F-test

(b) t-test

Figure 9: Box plots of the mean performace over the test instances of 20 configurations obtained by `irace` using $N^{max} \in \{1, \text{default}\}$.

Table 3: Wilcoxon signed-rank test p-values comparing configurations obtained by `irace` using the default setting of N^{max} vs. $N^{max} = 1$, over the test set.

	ACOTSP	SPEAR	MOACO
F-test	$3.624e^{-5}$	0.7285	0.4304
t-test	0.0005856	0.5958	0.4304

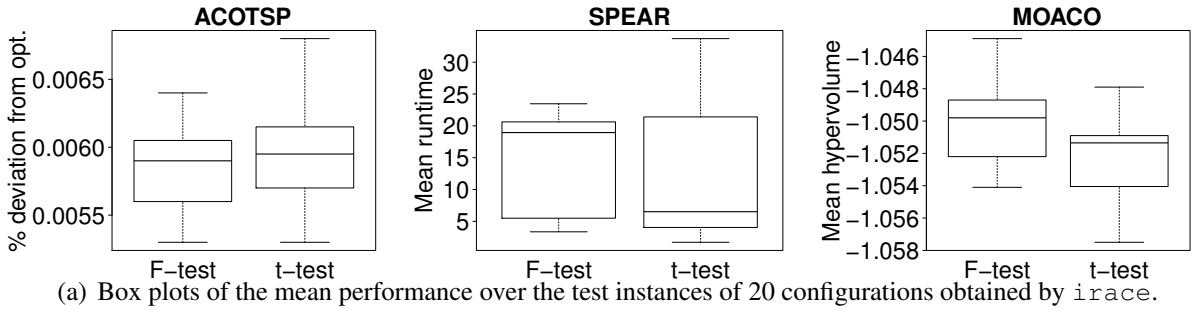
Table 4: Wilcoxon signed-rank test p-values comparing configurations obtained by `irace` using F-test vs. t-test.

ACOTSP	SPEAR	MOACO
0.2943	0.5958	0.03277

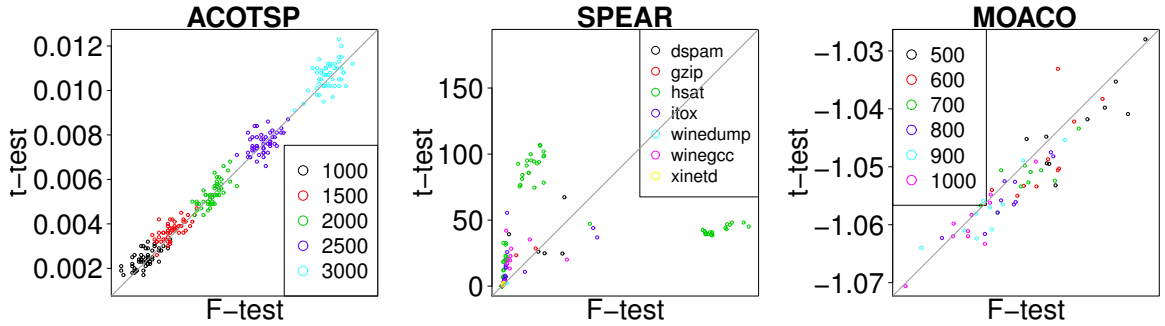
level of 0.99 is clearly worse than the default one. Even if on the MOACO configuration scenario the 0.99 confidence level is significantly better than default, the absolute difference is small and we would still recommend using the default 0.95 level. Differently, a smaller confidence level such as 0.75 may be a reasonable option. In fact, in two cases this setting is statistically better than the default setting while in one it is worse.

However, the results also indicate that the behavior of `irace` is affected differently by the confidence level used depending on the statistical test used (see, e.g. MOACO configuration scenario). This is different from the other experiments, where the impact of `irace` parameter settings was similar for both elimination tests.

Overall, the analysis provided above indicates that the default settings of `irace` appear to be reasonably robust and if nothing is known about the particular configuration scenario, they are a good first choice.

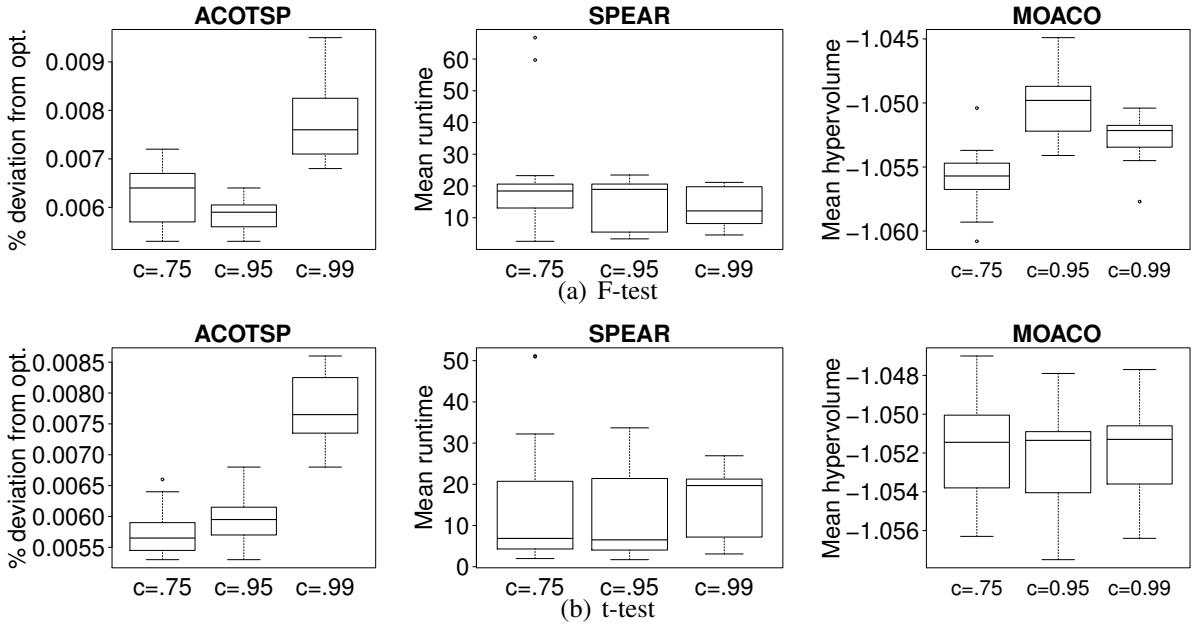


(a) Box plots of the mean performance over the test instances of 20 configurations obtained by `irace`.



(b) Scatter plots of the mean performace per instance over the 20 configurations obtained by `irace`.

Figure 10: Comparison of the mean performance over the test instances of 20 configurations obtained by `irace` using F-test and t-test. The bottom line of plots indicates the differences between the two possible elimination tests in dependence of specific subsets of instances in the test set.



(a) F-test

(b) t-test

Figure 11: Box plots of the mean performace over the test instances of 20 configurations obtained by `irace` using confidence level in $\{0.75, 0.95, 0.99\}$.

3.4 Improvements of `irace`

Following the analysis of `irace`, we have included a number of improvements in the `irace` software itself, which are summarized in the following. Many of these improvements had as

Table 5: Wilcoxon signed-rank test p-values comparing configurations obtained by `irace` using confidence level in $\{0.75, 0.95, 0.99\}$.

	0.75 vs. 0.95			0.99 vs. 0.95		
	ACOTSP	SPEAR	MOACO	ACOTSP	SPEAR	MOACO
F-test	0.01531	0.4091	$1.907e^{-6}$	$1.907e^{-6}$	0.7841	0.002325
t-test	0.02148	0.9563	0.1429	$1.907e^{-6}$	0.3683	0.2455

target to make the usage of the software more reliable for executing time-consuming configuration tasks as well as trying to improve further the performance of `irace`.

3.4.1 Similarity candidate check

In `irace` some steps such as the partial restart rely on checks of the similiarity of candidates. In the original implementation this was done using pairwise comparisons on the full list of parameters before taking a decision on the similarity. In the new version, the similarity test has been simplified by comparing parameters in dependence of their type through special functions and using an early termination of the similarity check that stops the comparison as soon as a threshold is passed that indicates that the similarity check would be evaluated to false. Empirical checks of the speed-up where done using example data to compare 1000 configurations of 30 parameters. The computation times for the similarity check where reduced from 456 seconds to 27 seconds on our hardware, resulting in a significant speed-up of the computations incurred by `irace`. This is particularly relevant for situations where the number of candidate configurations is large, a situation that arises if the configuration budget is large in the range of a few ten thousands of evaluations.

3.4.2 Restore functionality

In the new version of the `irace` package a restore functionality was introduced. This functionality allows to re-start a tuning run from the current iteration in case the execution of `irace` fails due to some external event. The extension of `irace` with the restore functionality was deemed to be necessary as `irace` execution could fail due to possible execution failures of the algorithms to be tuned or due to any problem that may happen on a cluster node. Such failures may also happen as parameter combinations are tested during the tuning that were not considered during the software development. This functionality proved to be relevant for longer tuning runs such as those that were considered in the tuning of the traffic lights.

3.4.3 Testing functionality

We have increased the usability and degree of automatization of the `irace` package by including a specific testing functionality. It is designed to directly extract the best configuration of the tuning process and execute this configuration on the set of test instances that are used to evaluate the performance of the tuned configurations. This allows to end a configuration run directly with an independent evaluation of the winners and streamlines the usage of the `irace` package.

3.4.4 Modified sampling models for categorical parameters

We have considered also different ways of how to generate the parameter settings for categorical parameters, that is, we have considered different sampling models for generating the values of categorical parameters. For this, we have adopted sampling models that have proven to improve the state-of-the-art algorithms in mixed-variable optimization [25, 23]. We adapted these sampling distributions for usage in the `irace` package and tested it using the benchmark tuning scenarios (experimental tests are not reported here explicitly). To our surprise, we did not achieve improved performance over the sampling model that was already used in the `irace` package, which is also the reason why the modified models are not included in the currently distributed version 1.05 of `irace`.

3.4.5 Elite `irace`

Over the recent months and weeks, we have been working on a rather strong modification of the `irace` search process. In particular, we have considered the development of an elitist version of `irace`. To explain the motivation for doing so, we have to recall that in `irace` in each iteration all the candidates including the elite ones from previous iterations are evaluated on new instances that are, depending on the number of instances available for tuning, often different from the ones used in the previous iterations. Due to this element of stochasticity in the evaluation, it may happen during the run of `irace` that very good and, in particular, the potentially final best configurations may be lost. This issue is exacerbated by the fact that the winning configuration in `irace`'s final iteration may be determined by evaluating it on less instances than some of the instances in previous iterations of an `irace` run. One example for the loss of high performance candidates is indicated in Figure 12, which gives the performance over the test instances of the iteration-best configuration in a tuning run on the ACOTSP configuration scenario. Interestingly, the configuration identified as `767` performs clearly better than the finally selected configuration with identifier `899`.

To avoid such situations, we decided to change the search behavior of `irace` by (i) guaranteeing that en elite configuration is never eliminated based on less instances than it has seen in a previous iteration of `irace`, (ii) re-using some of the computational results that have been obtained in previous iterations of `irace`. The first goal is implemented by forbidding to eliminate a configuration early before it has seen all the instances on which it has been evaluated previously. The second goal is implemented by re-using the evaluations of a configuration it has done in previous iterations. To do so, each instance is now seen as a pair of instance and seed for the algorithm's (the algorithm to be tuned) random number generator. If an elite configuration is to be evaluated again on an instance it has seen in a previous iteration of `irace`, the result it obtained is read from memory instead of re-executing the algorithm. As a side effect, this way of defining an instance as a pair of instance and random number seed also allows to introduce the common variance reduction technique of common random number.

In more detail, elitist `irace` works as the original `irace` with the following modifications. First, a random sample of k instances is evaluated on all initial configurations, which from iteration two on includes the elitist configurations and newly sampled ones. Next, all i_l instances that have been seen in the previous iteration $i-1, i \leq 2$ are evaluated in a random order. Non-elite configurations may be eliminated, while elite configurations may not in this process. If after the total number of $k+i_l$ instances more configurations survive than pre-scribed by `irace` settings, additional instances are evaluated. The number of additional instances for

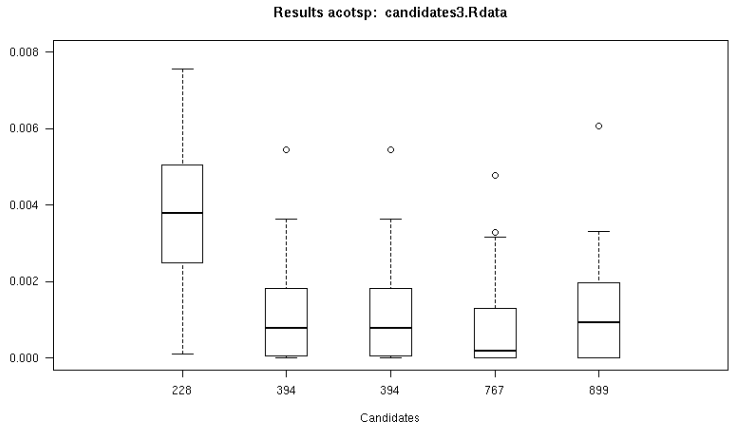
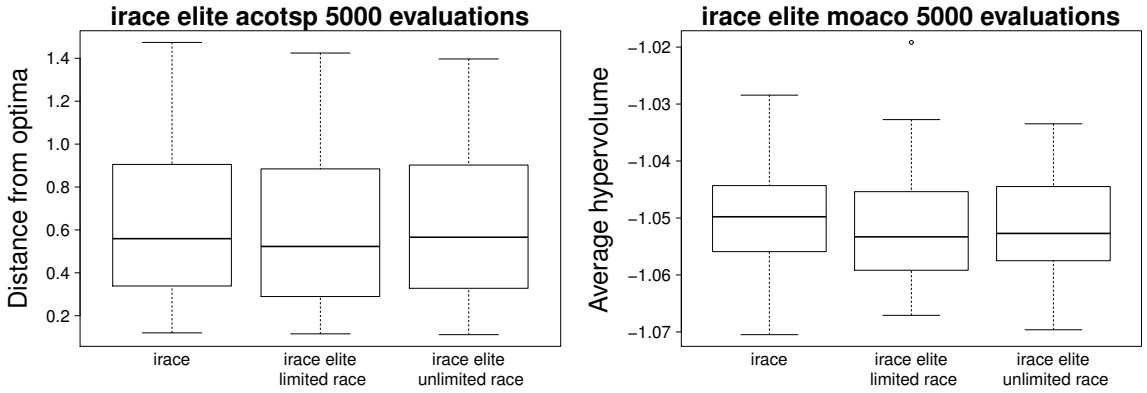


Figure 12: Box plots of the performance (given on the y-axis as the relative deviation from optimal solutions) over the test instances of the iteration-best configuration in a tuning run on the ACOTSP configuration scenario. Note that configuration with identifier `767` performs better than the finally returned one, which is `899`, which corresponds to a loss of the best configuration.



(a) Box plots of the mean performance over the test instances of 20 configurations obtained by `irace`.

Figure 13: Comparison of the mean performance over the test instances of 5 configurations obtained by `irace` and the limited and unlimited versions of elite `irace`.

evaluation may be limited or unlimited, leading to the two variants "limited elitist `irace`" and "unlimited elitist `irace`".

Some initial tests we did with the elitist `irace` appear to be positive. In Figure 13 we give two example results on the ACOTSP and MOACO configuration benchmarks for five executions of the `irace` variants. In both examples, the elite `irace` versions appear to improve slightly over the standard `irace` version. These are positive results as actually we expect that the elite `irace` will help especially in cases of configuration problems where the number of parameters is larger, as the changes that we introduced should generally lead to a more intensifying search behavior, which is particularly important for large dimensional problems. The elite version of `irace` will be included in the next major update of the publicly available software, which is expected by December 2014.

4 Tuning Traffic Light Control Algorithms: Preliminary results

In this section, we report on the results that were obtained for the tuning of the traffic light control algorithms that have been developed by the University of Bologna within the COLOMBO project. In the following we first explain how the tuning has been setup and applied to tune traffic light control software. Next, we report on the experimental results that have been obtained during our experiments. Note that some specific results on the traffic light control algorithms, an evaluation of their performance with respect to different penetration rates, and a presentation of the tuned parameter settings are described in detail in deliverable D2.3 [9]. Here, we focus on an analysis of the scenario as it is possible from the data obtained during the tuning and some results with different configuration budgets.

4.1 Experimental setup

4.1.1 Algorithm and scenario

In the COLOMBO project, a novel traffic light control algorithm is developed that is based on swarm intelligence principles. The traffic light control algorithm does not rely on specific infrastructure such as control loops etc. but instead can sense a percentage of equipped cars to infer information that is used to make control decisions. In particular, the algorithm works on a meta-level and allows to switch between different underlying policies such as Phase, Platoon, Marching, and Congestion. It uses pheromones which abstract from the particular traffic situation to switch between policies. A preliminary version of the algorithm is described in Deliverable D2.2 [8] and the most recent version is described and evaluated in Deliverable D2.3 [9]; we refer to these deliverables for a description of the algorithm.

For the configuration of the algorithm it is important to highlight the fact that it requires the setting of a large number of parameters in the range of a few tens (typical values are around 40), depending on the specific version that is used. These parameters are mostly numerical parameters that are either real or integer valued, but there are also few categorical parameters that allow to switch on or off specific algorithm components.

As a test of the automatic configuration process, we consider the simulation of a modified version of the first example of the German guideline for traffic light systems (RiLSA for short for "Richtlinie für Lichtsignalanlagen"). The layout of this scenario is shown in Fig. 14.

For the purpose of limiting the downstream flow, the scenario was modified by adding traffic lights for the north-south and east-west directions. More details on this scenario are given in Section 4 of Deliverable D2.3 [9].

4.1.2 Instance generation and performance measure

A main step for allowing the parameter tuning of the swarm algorithm is the definition of *instances* of the problem to be tackled. The problem to be tackled is the control of the traffic light at a specific intersection such that some given performance measure is optimized. The performance measure to be minimized in our example is the average waiting time. Analogous to classical optimziation problems, we then can define an instance of this problem to be a specific sequence of cars entering the "system" that want to pass through the traffic light. This

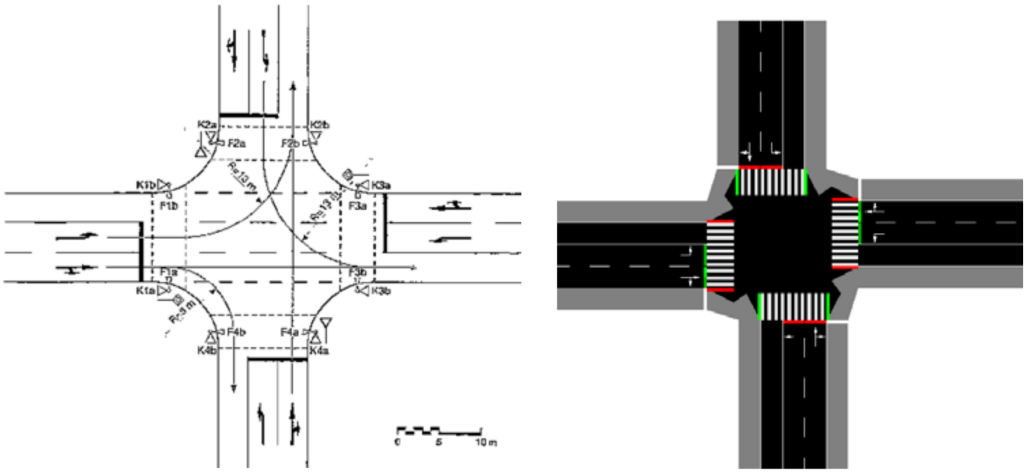


Figure 14: RiLSA scenario used for the automatic configuration process.

sequence is a realization of a random process that is defined by a specific entry flow of vehicles, a distribtion of the entry times, specific penetration rates[5] and any other details such as vehicle maximum speed etc. that are necessary to have data for all required data for the simulation. Note that once all these details are defined, the simulation done by SUMO and the traffic light control is deterministic. What differs among the instances is the final result of the performance measure as it depends on the particular instance.

Overall, the goal of the tuning then becomes to find a parameter configuration such that the performance measure is optimized. While for simple traffic light control algorithms, optimal parameter settings such as green light length may be obtained analytically, this is not anymore possible for the traffic light control algorithms considered here and, hence, the need to estimate performance through the actual execution of simulations.

To run the tuning, we have generated 1100 possible instances for each possible value of penetration rates in $\{100, 50, 25, 10, 5, 2.5, 1\}\%$ and each of the three possibilities (sensed car-time seconds, estimated car-time seconds, and queue length in Section 3 of Deliverable D2.3 [9]) considered to obtain indicator data that allow to switch between different chains in the traffic light control. 1000 of these instances were considered for the tuning while 100 were used as an independent test set to evaluate the performance of the traffic light control algorithm. For details on the traffic generator and the definition of the traffic flow patterns used in the generation, we refer the reader to Section 4.2.1 “Traffic Generator” of Deliverable D2.3 [9].

Each of the instances simulates a real time of 70 minutes, that is, 4200 seconds. Vehicles that enter the system in the first five minutes are discarded — these first five minutes do not represent a realistic situation as initially the net is empty. The performance measure used to evaluate a specific parameter setting is the average ”waiting steps” of vehicles (excluding the vehicles that entered the network in the first five minutes): SUMO simulations are divided into discrete time steps (1s), and a waiting step occurs when a vehicle waits because of a signal or because of other vehicles. If for any reason vehicles are blocked in the system, we apply a penalization that assigns to each vehicle not leaving the system 4200 waiting steps. This corresponds to a vehicle that waits during the whole simulation and if we have a complete

[5]The penetration rates defines how many vehicles are equipped so that they can be detected by the traffic light control.

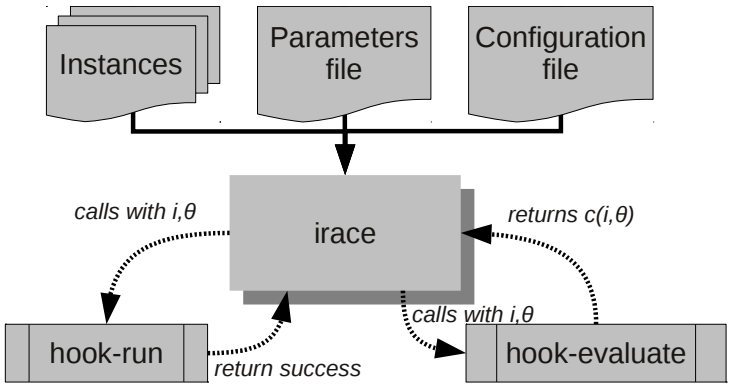


Figure 15: Scheme of `irace` flow of information.

grid-lock situation, that is, no vehicle at all leaves the net the obtained performance measure is 4200, which therefore also corresponds to an upper bound of the performance measure.

4.1.3 Specific setup for `irace` and evaluation of results

To set up the automatic configuration process, a number of procedures and data files need to be defined. The different components that are needed to execute `irace` are summarized in Figure 15.

Instances. A first component is a list of instances that are used for the tuning. In practice, these correspond to the instance data that we have generated for the tuning process as described above. The instances can then be stored in some directory and essentially a link to this directory needs to be given to `irace`. The instances are then fed into `irace` in a random order as to avoid any biases that may have been incurred by the specific way how instances are generated.

Configuration file. A second component is the configuration file, which corresponds to a specific setting of the parameters that are used by the `irace` package to direct its search process. Here, we use the default settings of `irace`. Hence, the configuration file mainly needs to specify the location of the training data (link to the directory) and the tuning budget (that is, the number of times the simulation is executed at most), which in our case here we set to $20,000$ evaluations, given that the number of parameters is relatively large (46).

Parameter file. A third component is the *parameter file*. The parameter file describes the parameters to be tuned by `irace`. A parameter is defined by (i) its name, (ii) its command line flag, (iii) its type, which can be real (r), integer (i), categorical (c) or ordinal (o), and (iv) the range of its possible values, which is given either as an interval (mostly used for numerical parameters, that is, real or integer ones) or as an explicit enumeration (usually used for ordinal and categorical parameters). Additionally, parameters can be dependent on constraints to be activated or conditional, that is, the activation of a parameter depends on specific values other parameters take. (Conditions are indicated after the symbol "|" in the parameter file. An example of a parameter file that we used for the tuning is given in the appendix, in Section 2, page 45.

Hook-run. The role of the *hook-run* is typically to interact with the algorithm to be tuned by calling it appropriately, and to pass along a value returned by the algorithm, corresponding to

the performance measure to be optimized, to `irace`. However, in the setup of `irace` for the tuning in COLOMBO, the hook-run needed to be more elaborated as it needs to return a single numerical value by parsing the files produced by SUMO that describe the outcome of the evaluation. The hook-run file (given in the appendix in Listing 1), was responsible of the following steps when evaluating a candidate configuration:

1. Translate the command line arguments (received from `irace`) to an XML file format that can be used by SUMO.

2. Call SUMO to perform the simulation

3. Parse the SUMO output file with the per-vehicle statistic, in order to compute a single value reflecting the quality of the candidate configuration used to manage the traffic light control in the simulation. This is done by only counting vehicles whose departures are more than five minutes after the beginning of the simulation; these vehicles we call "valid" vehicles. The performance measure to be computed is then the average number of waiting steps of valid vehicles. In the case of a vehicle that does not leave the net, its number of waiting steps is set to 4200. Since vehicles that do not leave the net are not reported in the output files by sumo, these vehicles are identified by searching for vehicles that are present in the flow file (ie, the instance), but not in the output file of SUMO. If no vehicle leaves the net at all, the value of 4200 is returned as the result of the whole evaluation.

4. Return the value computed after analyzing the flow and output files back to `irace`.

Hook-evaluate. In our case, hook-evaluate is not necessary.

4.1.4 Tuning setup

For the tuning experiments, we repeated one tuning for each specific setting of the available penetration rates and each of the three modes (i) sensed cars-times-seconds (CTS, referred to as mode 0), (ii) estimated cars-times-seconds (eCTS, referred to as mode 1), and iii) queue length (referred to as mode 2) of the swarm algorithm. For a precise definition of these modes we refer to Section 3 of Deliverable D2.3 [9]. We apply one tuning to each scenario that is defined by each possible combination of values in $100\%, 50\%, 25\%, 10\%, 5\%, 2.5\%, 1\%$ and the mode in $0, 1, 2$, resulting in a total of 21 independent tunings.

The experiments were run on computing nodes at IRIDIA's computing cluster, at the ULB partner lab. More precisely, each experiment was run on a single core of AMD Opteron 6272 CPUs, running at 2.1 Ghz with a 16 MB cache under Cluster Rocks Linux version 6/CentOS 6.3, 64bits. The cluster supports the MPI parallel environment, which was used to speed up the tuning by evaluating several candidates at the same time. If a tuning were to be executed on a single core, the overall tuning time would be about three CPU days. Thanks to the parallelization, between 10 and 50 parallel executions were used in practice and the tuning can be done in few hours on our cluster.

SUMO revision 17198 from the COLOMBO branch was used for the simulation, and compiled from the sources with gcc 4.4.6. The version of the SWARM algorithm that was used for the tuning is version 2 from Deliverable D2.3 [9], which uses three pheromone levels and one Gaussian for modeling; we will refer to it as SWARM2 in what follows. The python scripts

used to parse the results of the simulation were run with python 2.7.2. irace's version 1.04 was used with default parameter settings except that the tuning budget was set to 20000 simulations.

4.2 Results of irace applied to Traffic Light Systems

In what follows, we apply the irace configuration tool to the Swarm-based traffic light control algorithm (called SWARM 2 in what follows). We analyze both the final quality obtained by doing so, and the inner results that can be observed while performing the tuning, in order to better understand the properties of the search space at hand.

4.2.1 Cumulative distribution of solution quality

First, we analyze the distribution of the solution quality obtained by random configurations. To do so we use the data generated by irace when evaluating 475 random candidates on five different instances. These candidate configurations are actually generated in the first iteration of the irace method: the default behavior of irace is to generat the initial set of candidates randomly as no default configuration was available for SWARM.

We plot in Fig. 16 the cumulative empirical probability distribution of the performance measure obtained for each of these five instances, and the average performance across these five instances. These so called solution cost distributions indicate here the empirical frequency of configurations that obtain an average performance than some specific value given on the x-axis. Due to the occurrence of few outliers, we have displayed the solution quality on the x-axis in logarithmic scale to be able to better differentiate the results among the configurations. Somehow surprisingly, the values of the performance measure of a large fraction of the configurations are very similar, suggesting that even random configurations may be surprisingly good. This can be seen by the fact that the empiricial cimulative distribitions are rather steep and that only very few outliers with poor performance (for example, values larger than 100 are available).

4.2.2 Number of surviving candidate configurations

irace, and racing procedures in general, are based on the idea of discarding candidates as soon as there is sufficient statistical evidence that they are of lower quality than others.

To visualize the ability of irace to discard candidates when tuning the SWARM algorithm, we plot in Fig. 17 the number of configurations that remain alive across the different instances that are seen for each race of a full irace execution (for a target penetration rate of 50%). When the statistical test is applied (after 5 instances), a large number of candidates are discarded (roughly 90%), but very little in subsequent statistical tests until the end of the race - less than what is typically observed when tuning optimization algorithms. This suggests that the search space (as seen from the irace perspective, where solutions are in fact candidate configurations) is relatively flat, that is, that there exists a large number of algorithm configurations that are of similar quality.

One can observe that this is slightly mitigated during the two last races, where irace is able to discard a significant number of candidates several times after the first test is performed. This difference shows that there is a stronger statistical difference between candidates that are

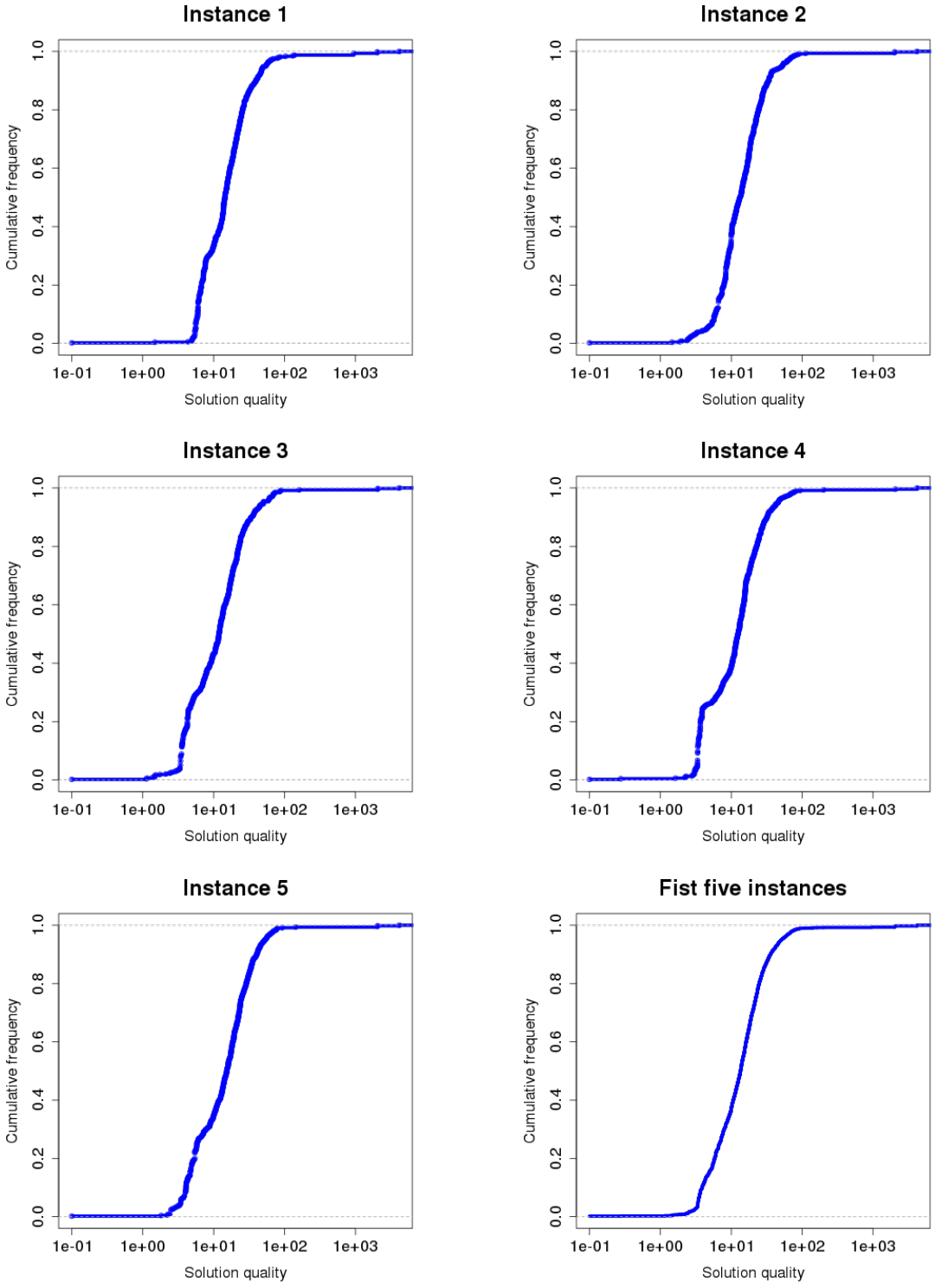


Figure 16: Cumulative distribution of solution quality (measured as the average waiting steps) for each of the first five instances, and average across the five instances, for 475 random candidate configurations).

generated in the small area (when `irace` converges and generate new candidates close to previous successful ones), compared to candidates that are sampled randomly across the entire search space (when `irace` starts and samples candidates without a priori knowledge where good configurations may be).

There may be two explanations of this fact. One is that the SWARM algorithm is rather robust with respect to specific settings of its parameters. Another is that the particular traffic scenario chosen is a relatively simple one that does not provide a significant enough challange to the SWARM algorithm. In fact, both explanations would be consistent with knowledge that we obtained on tuning tasks that we performed earlier in out research for optimization algorithms.

4.2.3 Automatic Configuration vs. Random Settings

Given the previous results, we performed experiments to explicitly compare the quality obtained on the test set after running a full tuning, w.r.t. the quality obtained by random configurations. Figure 18 shows, for two different penetration rates (100% and 1%) and the three different modes, a comparison of the best configuration obtained by the tuning process (with a budget of $20,000$ and under the respective conditions of penetration rate and mode), and two configurations obtained uniformly at random. We used the `irace` sampling mechanism in order to obtain random configurations that satisfy the constraints for each parameter (see parameter file in Sec. 2, Appendix).

The difference is relatively small, which is consistent with the results of the previous experiments, where we have shown the distribution of the solution quality obtained by random configuration of SWARM. It may suggest that the scenario at hand is overly simplified, and may not be sufficient to finely discriminate the best configurations w.r.t. "good enough" random ones.

4.2.4 Comparison of different tuning budgets

In these experiments we compare two different tuning budgets, to assess whether a the large tuning budget of 20000 simulation runs that we used is actually beneficial, or whether a smaller tuning budget could be sufficient. We re-ran again 21 independent tunings for each possible setup, but this time with a smaller budget of $5,000$ evaluations. Figure 19 presents the comparison of the two best configurations obtained for each setup, one obtained with a budget of $20,000$ evaluations (left) and the other with $5,000$ (right).

The differences are generally very small, and sometimes the smaller budget might even obtain slightly better results due to stochasticity. Interestingly, in the experimental results presented here we found that some SWARM configurations gave poor performance on very few test instances. This is visible in Figure 19 in the plots for penetration rates PR 25%, mode 1 and 2 and penetration rate 10%, mode 1. These particular cases should be further analyzed as they may indicate situations where the SWARM algorithm needs adaptation; however, it may be that with further improved variants of the SWARM algorithm, as described in Deliverable D2.3 [9], these situations may be re-solved.

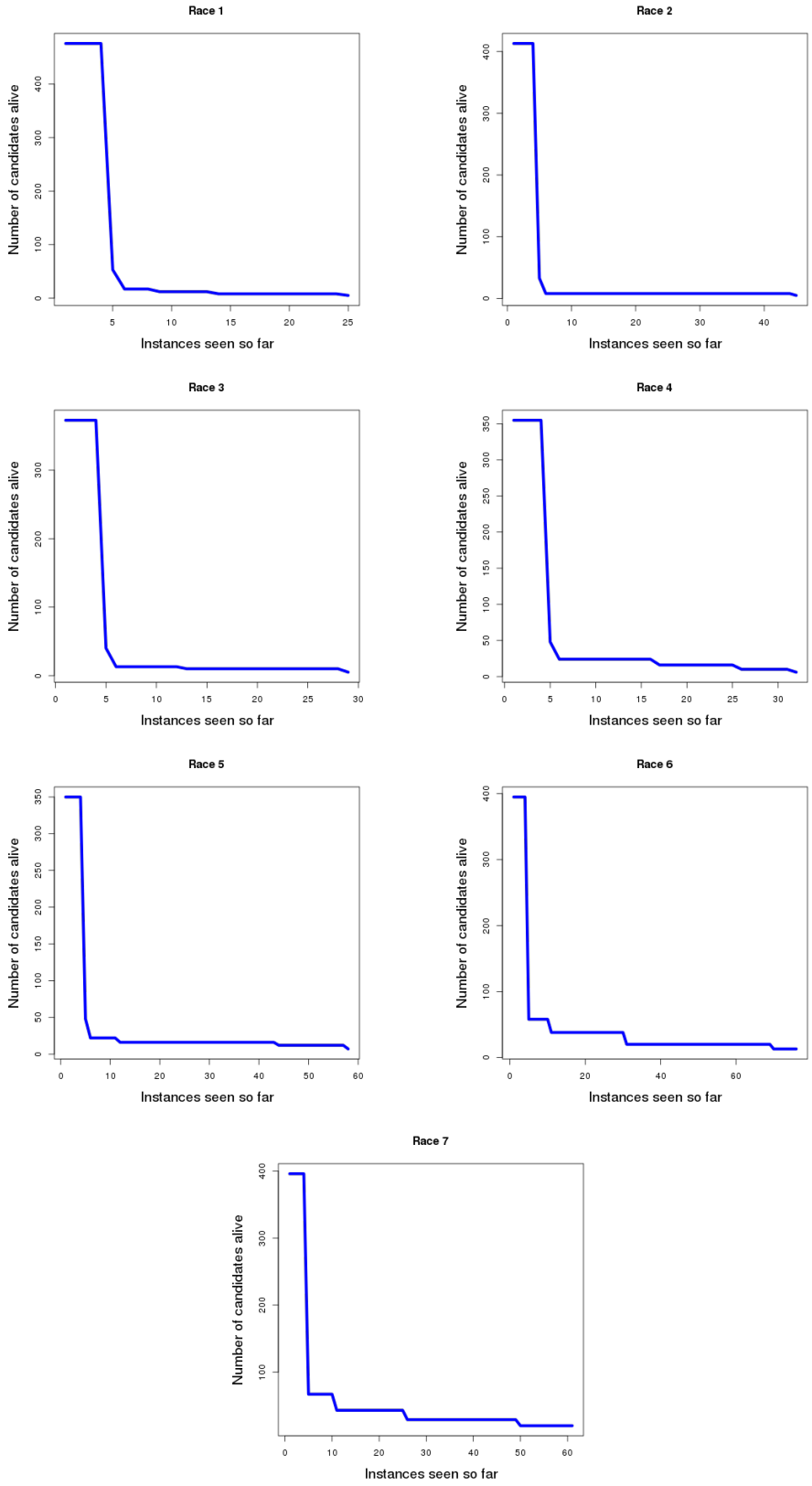


Figure 17: Development of the number of alive candidates during each race of a tuning (with PR=50% and mode=1).

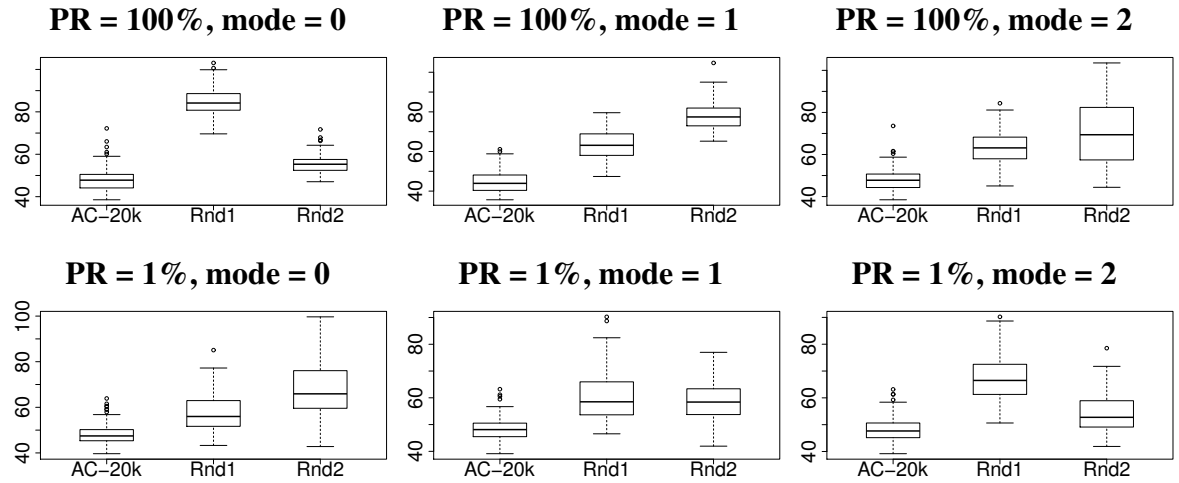


Figure 18: Boxplots comparing the results obtained by configurations returned by `irace` (identified AC-20k) with a budget of 20,000 simulation runs to two randomly sampled configurations. The comparison is done on 100 test instances for each penetration rate and mode combination. The performance measure given on the x-axis is the average waiting time.

4.2.5 Influence of the penetration rates

We analyze the influence of the penetration rate on the results. More precisely, we want to understand if a configuration obtained for a given penetration rate is strongly "specialized", that is, it performs much better for its target penetration rate for which it was tuned than for others. We present in Fig. 20 the results of this comparison, for one mode (mode = 0). Since the parameters are slightly different in the case of a 100% penetration rate, it is not included in the comparison. Therefore, one can see the performance obtained by all configurations, for each penetration rate in $\{50\%, 25\%, 10\%, 5\%, 2.5\%, 1\%\}$. For instance the configuration named "AC-20k-1%" is the configuration obtained by a tuning with a budget of $20,000$ and specifically targeting 1% penetration rates situations. The results indicate that the specialization is relatively low, in the sense that the different configurations obtain similar results across the different penetration rates, whether they were obtained for these penetration rates or not. The strongest difference that can be observed is that the configuration obtained for a penetration rate of 50% performs significantly worse than others on simulations with a penetration rate of 1% and 10%. These results seem to indicate that a configuration obtained for a low penetration rate is better able to scale to higher penetration rates, than the opposite. On the other side, these results also indicate that it is important to tune parameter settings of SWARM taking into account the penetration rate as for small penetration rate the parameters obtained for large ones should not be used.

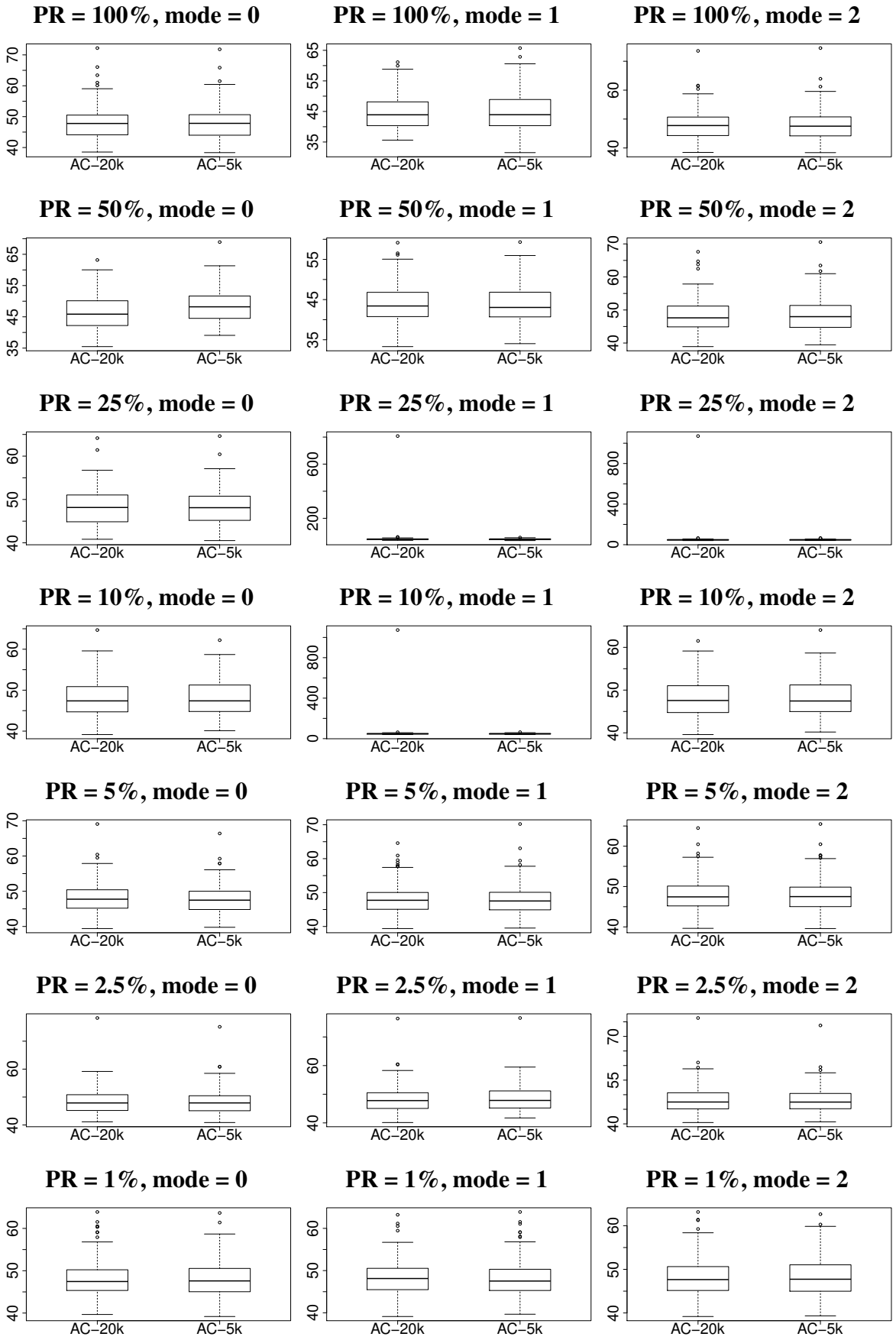


Figure 19: Boxplots comparing the performance of two automatically obtained configurations on 100 test instances. The configuration AC-20k has been obtained with a budget of 20,000 and the configuration AC-5k with a budget of 5,000 simulation runs. The performance measure given on the x-axis is the average waiting time.

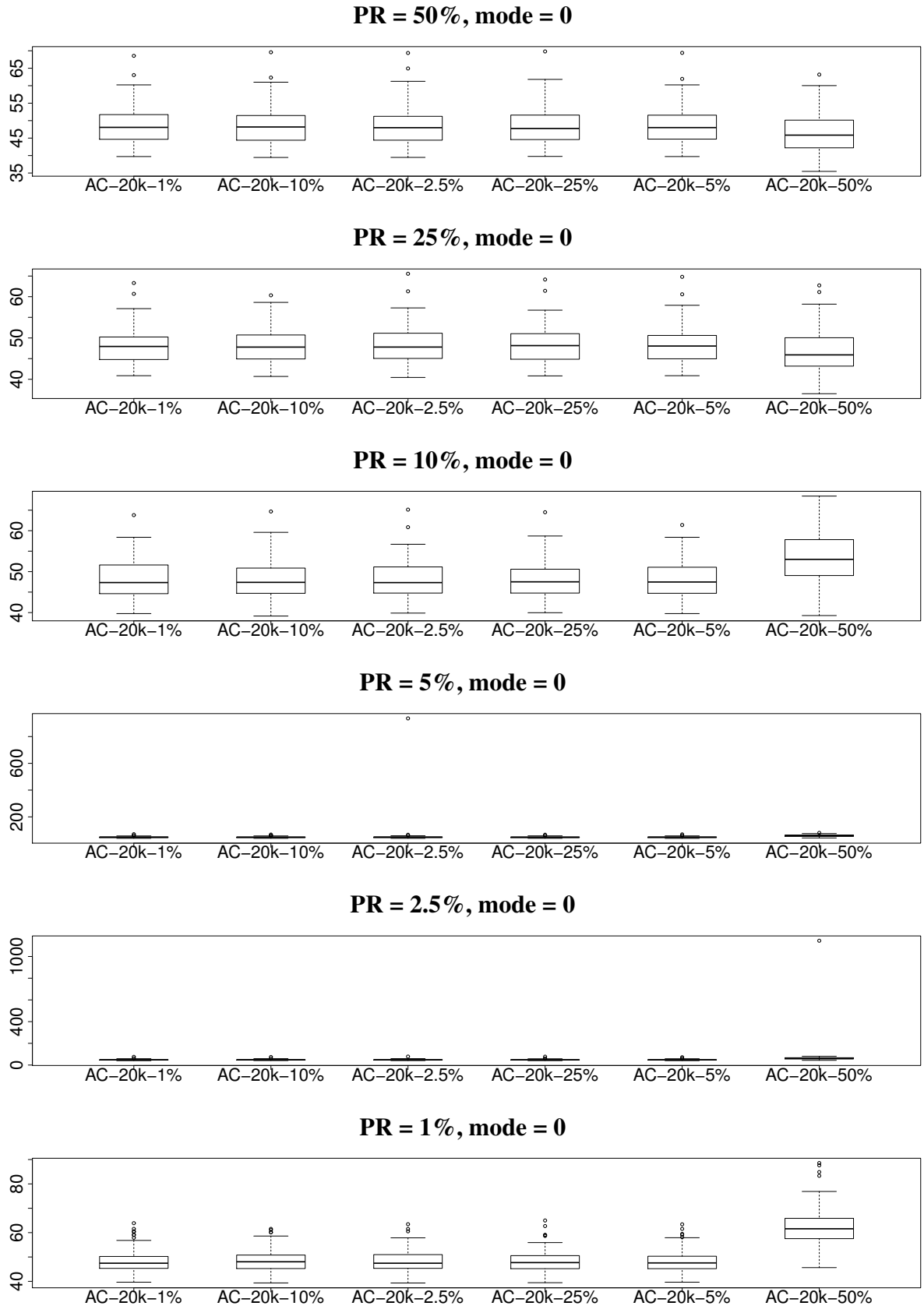


Figure 20: Results of configurations obtained for different target PR, across all different PR (with the exception of 100%). The performance measure given on the x-axis is the average waiting time.

5 Summary

This document outlined the progress that has been made on automatic algorithm configuration methods and software, shows the applicability of the automatic configuration software towards tuning novel traffic light control software and analyzes the results.

Contributions related to automatic algorithm configuration are the following. We have finalized the tuning tool kit and made it publicly available using a dedicated website for it. We have then used this tool kit to compare the performance of high-performing automatic algorithm configuration software on few benchmark configuration tasks that are known from the optimization community. In particular, we compared the three tuners `irace`, paramILS, and SMAC. For the purposes of the COLOMBO project, we have identified the `irace` package, which was developed by Université libre de Bruxelles, as a suitable software for the kind of tuning tasks that arise in the project. Following this comparison, we have started an in-depth analysis of the impact specific parameter settings have of the `irace` package itself on its performance. This analysis was also published at an international conference [33]. The analysis has provided also further ideas of how to improve the software's performance and also practical aspects of its usability. These improvements have been included already or are still to be included (e.g. the elite `irace`) in the publicly available version of `irace`. A further publication on the improved version of `irace` is currently in the final writing stages.

We have then explored the usability and the performance of the `irace` software for the tuning of the traffic light control software developed by the BOLOGNA partner of COLOMBO. In fact, the use of the automatic configuration was identified as crucial as the control software has a large number of parameters and reaching a best possible setting of these parameters manually was deemed to be infeasible. Computational results for the control of a single intersection using different penetration rates of equipped vehicles has shown that the automatic configuration tools could obtain consistently very high performing parameter configurations. While initial experiments were performed with a rather large tuning budget of 20000 simulation runs, qualitatively the same results could be obtained even with a much smaller budget of only 5000 simulation runs. This shows that even despite the large number of parameters, good parameter configurations may be obtained quickly.

Future work has to test the automatic configuration of the traffic lights for more complex scenarios than the ones that we have considered so far including such that have several connected crossings that are to be controlled. We believe that only then the full potential of the approach will become evident.

6 Appendix

6.1 Hook-run

Listing 1: Hook-run file used (logging instructions have been removed for clarity)

```
#!/bin/bash
##############################################################################
# This hook is to tune the policies used by SUMO software.
#
# PARAMETERS:
# $1 is the instance name
# $2 is the candidate number
# The rest ($* after 'shift 2') are parameters to make the .add file
#
# DEPENDS ON:
# ../scripts/MakeAdditional.rilsa.py
# ../nets/rilsa1.net.xml
# ../scripts/singleEvaluator.py (version returning 2 values, avg & nb vehicles)
#
# RETURN VALUE:
# Mean waitSteps value for vehicles departing after 10m
##############################################################################

# Avoid xerces error "Could not load a transcoding service"
export LC_CTYPE="en_US.UTF-8"
# To switch python version, put link in ~/bin
export PYTHON="/lustre/home/jdubois/bin/python-for-colombo"

# The instance name and the candidate id are the first parameters
INSTANCE=$1
CANDIDATE=$2

TRAFFIC_FLOW=$INSTANCE

# Path to the SUMO software and other scripts:
SUMO=~/bin/sumo
MAKEADD=../scripts/MakeAdditional.rilsa.py

# This is the definition of the net, times for traffic lights are in add_xml
NETFILE=../nets/rilsa1.net.xml

# ADDFILE is file with parameters written in xml sumo format
ADD_FILE_CREATED=./$CANDIDATE.add.xml
# Result from sumo:
TRIPINFO=./$CANDIDATE.xml
# Avoid evaluator crash with error "couldn't reach EOF", while reading file
echo -e "\c" > $TRIPINFO

# How to return numerical result from outputs of sumo:
EVALUATOR=../scripts/singleEvaluator.py

# All other parameters are the candidate parameters to be passed to *_make_additional
shift 2 || exit 1

# Setting default parameter value, so if a parameter is not enabled by irace, it still exists
# in bash and therefore will be there to create add file
# Note: we could only set here the optional parameter (see parameters.txt)
threshold=1
max_cong_d=1
decay_constant=1
thrspeed=1
change_plan_p=1
gamma_sp=1
beta_sp=1
gamma_no=1
beta_no=1
```

```
theta_max=1
theta_min=1
theta_init=1
learning_cox=1
forgetting_cox=1
phase_stim_cox=1
phase_stim_offset_in=1
phase_stim_offset_out=1
phase_stim_divisor_in=1
phase_stim_divisor_out=1
phase_stim_cox_exp_in=1
phase_stim_cox_exp_out=1
phase_stim_offset_dispersion_in=1
phase_stim_divisor_dispersion_in=1
phase_stim_cox_exp_dispersion_in=1
platoon_stim_cox=1
platoon_stim_offset_in=1
platoon_stim_offset_out=1
platoon_stim_divisor_in=1
platoon_stim_divisor_out=1
platoon_stim_cox_exp_in=1
platoon_stim_cox_exp_out=1
platoon_stim_offset_dispersion_in=1
platoon_stim_divisor_dispersion_in=1
platoon_stim_cox_exp_dispersion_in=1
marching_stim_cox=1
marching_stim_offset_in=1
marching_stim_offset_out=1
marching_stim_divisor_in=1
marching_stim_divisor_out=1
marching_stim_cox_exp_in=1
marching_stim_cox_exp_out=1
marching_stim_offset_dispersion_in=1
marching_stim_divisor_dispersion_in=1
marching_stim_cox_exp_dispersion_in=1
congestion_stim_cox=1
congestion_stim_divisor_out=1

# Get parameter values from the sequence --flag1 <value1> --flag2 <value2>
while [ "$#" -gt 0 ]
do
    flag=$1
    value=$2
    # Let's remove scientific notation, ie 3e-4 -> 0.00030
    echo $value | fgrep 'e' &> /dev/null
    if [ "$?" -eq 0 ]
    then
        # Scientific notation found
        plainNotation=$(echo $2 | sed 's/+//' | sed 's/e/\*10\^/')
        value=$(echo "scale=5; $plainNotation" | bc)
    fi

    case "$flag" in
        --threshold) threshold=$value
            ;;
        --max_cong_d) max_cong_d=$value
            ;;
        --decay_constant) decay_constant=$value
            ;;
        --thrspeed) thrspeed=$value
            ;;
        --change_plan_p) change_plan_p=$value
            ;;
        --gamma_sp) gamma_sp=$value
            ;;
        --beta_sp) beta_sp=$value
            ;;
        --gamma_no) gamma_no=$value
            ;;
        --beta_no) beta_no=$value
```

```
    ;;
--theta_max) theta_max=$value
    ;;
--theta_min) theta_min=$value
    ;;
--theta_init) theta_init=$value
    ;;
--learning_cox) learning_cox=$value
    ;;
--forgetting_cox) forgetting_cox=$value
    ;;
--phase_stim_cox) phase_stim_cox=$value
    ;;
--phase_stim_offset_in) phase_stim_offset_in=$value
    ;;
--phase_stim_offset_out) phase_stim_offset_out=$value
    ;;
--phase_stim_divisor_in) phase_stim_divisor_in=$value
    ;;
--phase_stim_divisor_out) phase_stim_divisor_out=$value
    ;;
--phase_stim_cox_exp_in) phase_stim_cox_exp_in=$value
    ;;
--phase_stim_cox_exp_out) phase_stim_cox_exp_out=$value
    ;;
--phase_stim_offset_dispersion_in) phase_stim_offset_dispersion_in=$value
    ;;
--phase_stim_divisor_dispersion_in) phase_stim_divisor_dispersion_in=$value
    ;;
--phase_stim_cox_exp_dispersion_in) phase_stim_cox_exp_dispersion_in=$value
    ;;
--platoon_stim_cox) platoon_stim_cox=$value
    ;;
--platoon_stim_offset_in) platoon_stim_offset_in=$value
    ;;
--platoon_stim_offset_out) platoon_stim_offset_out=$value
    ;;
--platoon_stim_divisor_in) platoon_stim_divisor_in=$value
    ;;
--platoon_stim_divisor_out) platoon_stim_divisor_out=$value
    ;;
--platoon_stim_cox_exp_in) platoon_stim_cox_exp_in=$value
    ;;
--platoon_stim_cox_exp_out) platoon_stim_cox_exp_out=$value
    ;;
--platoon_stim_offset_dispersion_in) platoon_stim_offset_dispersion_in=$value
    ;;
--platoon_stim_divisor_dispersion_in) platoon_stim_divisor_dispersion_in=$value
    ;;
--platoon_stim_cox_exp_dispersion_in) platoon_stim_cox_exp_dispersion_in=$value
    ;;
--marching_stim_cox) marching_stim_cox=$value
    ;;
--marching_stim_offset_in) marching_stim_offset_in=$value
    ;;
--marching_stim_offset_out) marching_stim_offset_out=$value
    ;;
--marching_stim_divisor_in) marching_stim_divisor_in=$value
    ;;
--marching_stim_divisor_out) marching_stim_divisor_out=$value
    ;;
--marching_stim_cox_exp_in) marching_stim_cox_exp_in=$value
    ;;
--marching_stim_cox_exp_out) marching_stim_cox_exp_out=$value
    ;;
--marching_stim_offset_dispersion_in) marching_stim_offset_dispersion_in=$value
    ;;
--marching_stim_divisor_dispersion_in) marching_stim_divisor_dispersion_in=$value
    ;;
--marching_stim_cox_exp_dispersion_in) marching_stim_cox_exp_dispersion_in=$value
```

```
            ;;
        --congestion_stim_cox) congestion_stim_cox=$value
            ;;
        --congestion_stim_divisor_out) congestion_stim_divisor_out=$value
            ;;
        *) echo "Unkown parameter '$flag' detected."
            ;;
    esac

    shift 2 || exit 1
done

# Build the whole command line from these arguments (order matters!)
CAND_PARAMS="${threshold} ${max_cong_d} ${decay_constant} ${thrspeed} ${change_plan_p} ${gamma_sp} \
  ${beta_sp} ${gamma_no} ${beta_no} ${theta_max} ${theta_min} ${theta_init} ${learning_cox} \
  ${forgetting_cox} ${phase_stim_cox} ${phase_stim_offset_in} ${phase_stim_offset_out} \
  ${phase_stim_divisor_in} ${phase_stim_divisor_out} ${phase_stim_cox_exp_in} \
  ${phase_stim_cox_exp_out} ${phase_stim_offset_dispersion_in} ${phase_stim_divisor_dispersion_in} \
  ${phase_stim_cox_exp_dispersion_in} ${platoon_stim_cox} ${platoon_stim_offset_in} \
  ${platoon_stim_offset_out} ${platoon_stim_divisor_in} ${platoon_stim_divisor_out} \
  ${platoon_stim_cox_exp_in} ${platoon_stim_cox_exp_out} ${platoon_stim_offset_dispersion_in} \
  ${platoon_stim_divisor_dispersion_in} ${platoon_stim_cox_exp_dispersion_in} ${marching_stim_cox} \
  ${marching_stim_offset_in} ${marching_stim_offset_out} ${marching_stim_divisor_in} \
  ${marching_stim_divisor_out} ${marching_stim_cox_exp_in} ${marching_stim_cox_exp_out} \
  ${marching_stim_offset_dispersion_in} ${marching_stim_divisor_dispersion_in} \
  ${marching_stim_cox_exp_dispersion_in} ${congestion_stim_cox} ${congestion_stim_divisor_out}"

# Where do we write?
STDOUT="c${CANDIDATE}.stdout"
STDERR="c${CANDIDATE}.stderr"
# Create those files (or append empty line)
echo -e "\c" > $STDOUT
echo -e "\c" > $STDERR

echo "Starting to evaluate candidate $CANDIDATE: ${CAND_PARAMS}" >> $STDOUT

# Translate all parameters currently as long command line into sumo XML format
echo "making add file from template..." >> $STDOUT
$PYTHON $MAKEADD ${CAND_PARAMS} $CANDIDATE ../templates/rilsa.swarm.template.add.xml \
   1>> $STDOUT 2>> $STDERR

# Actually call sumo
$SUMO --net-file $NETFILE --route-files ../rilsa_traffic_flows_tuning/${TRAFFIC_FLOW} \
    --additional-files ${ADD_FILE_CREATED},../adds/vtypes.add.xml --tripinfo-output $TRIPINFO \
    --time-to-teleport=-1 -X never -e 4200 1>> $STDOUT 2>> $STDERR

# /tmp used, but we don't mind what's written there
export MPLCONFIGDIR=/tmp

# Evaluate the tripinfo result file generated by sumo with per-vehicle statistics
# Result returned is of this form: "<average_time_per_valid_vehicle> <number_of_valid_vehicles>"
evalResult=$($PYTHON $EVALUATOR $TRIPINFO 2>> $STDERR)
result=$(echo $evalResult | cut -d ' ' -f1)
nbVehicles=$(echo $evalResult | cut -d ' ' -f2)

# If result = 0, not a single car succeeds, it's a giant grid-locks
# let's penalize it to the maximum.
if [ $(echo "$result > 0" | bc) -eq 0 ]
then
    penalty=4200 # Arbitrarily high (here number of steps)
    result=$penalty
    echo "found waiting time of 0, applied penalty!" >> $STDOUT
    echo $result
    exit 0
fi

# if we have some vehicles that did not leave the net,
# we penalize them with 4200. So to the average already computed,
# we add 4200 * (number of vehicles that did not leave the net)
# The next line counts the number of vehicle in the traffic flow file
```

```
# that depart after or equal to 600s (10m). This corresponds to the
# calculation done in scripts/singleEvaluator.py:
nb_vehicles_with_depart_higher_than_10m=\
    $(fgrep "depart" ../rilsa_traffic_flows_tuning/${TRAFFIC_FLOW} | \
    sed 's/^.*depart="\([0-9.]*\).*$/\1/' | sort -n | \
    xargs -I {} echo "{} >= 600" | bc | fgrep "1" | wc -l)
nb_vehicles_in_rou_file=$(fgrep "depart" ../rilsa_traffic_flows_tuning/${TRAFFIC_FLOW} | \
    wc -l)
nb_vehicles_that_left_the_net=$(fgrep "tripinfo id=" $TRIPINFO |  wc -l)
nb_vehicles_that_did_not_left_net=$(echo "${nb_vehicles_in_rou_file} - ${nb_vehicles_that_left_the_net}" | \
    bc)
sum_of_delay_time_from_singleEvaluator=$(echo "scale=4; $result * $nbVehicles" | bc)
newAverage=$(echo "scale=4; (${sum_of_delay_time_from_singleEvaluator} + \
    (4200 * ${nb_vehicles_that_did_not_left_net})) \ (${nb_vehicles_with_depart_higher_than_10m} \
    + ${nb_vehicles_that_did_not_left_net})" | bc)
result=$newAverage

echo "Result computed: $result" >> $STDOUT

# Done with our duty. Clean files created and exit with 0 (no error).
# Delete standard output and err files
rm -f "${STDOUT}" "${STDERR}"
# Delete XML files generated for this candidate
rm -f ${ADD_FILE_CREATED} $TRIPINFO
# Return result
echo $result
}
```

6.2 Parameter file

Listing 2:

```
threshold                   "--threshold "              i       (10, 3000)
max_cong_d                  "--max_cong_d "             i       (30, 120)
decay_constant              "--decay_constant "         r       (-0.001, -0.00001)
thrspeed                    "--thrspeed "               r       (1, 8)
change_plan_p               "--change_plan_p "          r       (0.01000, 0.99000)
gamma_sp                    "--gamma_sp "               r       (0.01000, 1.00000)
beta_sp                     "--beta_sp "                r       (0.01000, 0.99999)
gamma_no                    "--gamma_no "               r       (0.01000, 1.00000)
beta_no                     "--beta_no "                r       (0.01000, 0.99999)
theta_max                   "--theta_max "              r       (0.60000, 1.00000)
theta_min                   "--theta_min "              r       (0.03000, 0.50000)
theta_init                  "--theta_init "             r       (0.50000, 0.60000)
learning_cox                "--learning_cox "           r       (0.00010, 0.01000)
forgetting_cox              "--forgetting_cox "         r       (0.00010, 0.10000)
phase_stim_cox              "--phase_stim_cox "         r       (0.02000, 1.00000)
platoon_stim_cox            "--platoon_stim_cox "               r       (0.02000, 1.00000)
marching_stim_cox           "--marching_stim_cox  "             r       (0.02000, 1.00000)
phase_stim_cox_exp_in       "--phase_stim_cox_exp_in "          c       (0, 1)
phase_stim_divisor_in       "--phase_stim_divisor_in "          r       (1, 10.00000)       | phase_stim_cox_exp_in == "1"
phase_stim_offset_in        "--phase_stim_offset_in "           r       (0.00000, 10.00000) | phase_stim_cox_exp_in == "1"
phase_stim_cox_exp_out      "--phase_stim_cox_exp_out "         c       (0, 1)
phase_stim_divisor_out      "--phase_stim_divisor_out "         r       (1, 10.00000) | phase_stim_cox_exp_out == "1"
phase_stim_offset_out       "--phase_stim_offset_out "          r       (0.00000, 10.00000) | phase_stim_cox_exp_out == "1"
phase_stim_cox_exp_dispersion_in        "--phase_stim_cox_exp_dispersion_in "           c       (0, 1)
phase_stim_divisor_dispersion_in "--phase_stim_divisor_dispersion_in " r (1, 10.00000) | phase_stim_cox_exp_dispersion_in == "1"
phase_stim_offset_dispersion_in "--phase_stim_offset_dispersion_in " r (0.00000, 10.00000) | phase_stim_cox_exp_dispersion_in == "1"
platoon_stim_cox_exp_in "--platoon_stim_cox_exp_in "    c (0, 1)
platoon_stim_divisor_in "--platoon_stim_divisor_in "    r (1, 10.00000)   | platoon_stim_cox_exp_in == "1"
platoon_stim_offset_in  "--platoon_stim_offset_in "     r (0.00000, 10.00000)     | platoon_stim_cox_exp_in == "1"
platoon_stim_cox_exp_out "--platoon_stim_cox_exp_out "  c (0, 1)
platoon_stim_divisor_out "--platoon_stim_divisor_out "  r (1, 10.00000)        | platoon_stim_cox_exp_out == "1"
platoon_stim_offset_out "--platoon_stim_offset_out "    r (0.00000, 10.00000)  | platoon_stim_cox_exp_out == "1"
platoon_stim_cox_exp_dispersion_in "--platoon_stim_cox_exp_dispersion_in " c (0, 1)
platoon_stim_divisor_dispersion_in "--platoon_stim_divisor_dispersion_in " r (1, 10.00000) | platoon_stim_cox_exp_dispersion_in == "1"
platoon_stim_offset_dispersion_in "--platoon_stim_offset_dispersion_in " r (0.00000, 10.00000) | platoon_stim_cox_exp_dispersion_in == "1"
marching_stim_cox_exp_in "--marching_stim_cox_exp_in " c (0, 1)
marching_stim_divisor_in "--marching_stim_divisor_in " r (1, 10.00000)        | marching_stim_cox_exp_in == "1"
marching_stim_offset_in "--marching_stim_offset_in "   r (0.00000, 10.00000)  | marching_stim_cox_exp_in == "1"
marching_stim_cox_exp_out "--marching_stim_cox_exp_out " c (0, 1)
marching_stim_divisor_out "--marching_stim_divisor_out " r (1, 10.00000)        | marching_stim_cox_exp_out == "1"
marching_stim_offset_out "--marching_stim_offset_out " r (0.00000, 10.00000)  | marching_stim_cox_exp_out == "1"
marching_stim_cox_exp_dispersion_in "--marching_stim_cox_exp_dispersion_in " c  (0, 1)
marching_stim_divisor_dispersion_in "--marching_stim_divisor_dispersion_in " r  (1, 10.00000) | marching_stim_cox_exp_dispersion_in == "1"
marching_stim_offset_dispersion_in "--marching_stim_offset_dispersion_in " r (0.00000, 10.00000) | marching_stim_cox_exp_dispersion_in == "1"
congestion_stim_cox "--congestion_stim_cox " r  (0.02000, 1.00000)
congestion_stim_divisor_out "--congestion_stim_divisor_out " r  (1, 10.00000)
```

References

[1] B. Adenso-Díaz and M. Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99–114, 2006.

[2] Domagoj Babić and Frank Hutter. Spear theorem prover. In *SAT'08: Proceedings of the SAT 2008 Race*, 2008.

[3] Prasanna Balaprakash, Mauro Birattari, and Thomas Stützle. Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In Thomas Bartz-Beielstein, María J. Blesa, Christian Blum, Boris Naujoks, Andrea Roli, Günther Rudolph, and M. Sampels, editors, *Hybrid Metaheuristics*, volume 4771 of *Lecture Notes in Computer Science*, pages 108–122. Springer, Heidelberg, Germany, 2007.

[4] Mauro Birattari. *Tuning Metaheuristics: A Machine Learning Perspective*, volume 197 of *Studies in Computational Intelligence*. Springer, Berlin/Heidelberg, Germany, 2009.

[5] Mauro Birattari, Thomas Stützle, Luís Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[6] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-race and iterated F-race: An overview. In Thomas Bartz-Beielstein, Marco Chiarandini, Luís Paquete, and Mike Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer, Berlin, Germany, 2010.

[7] R.J. Blokpoel, D. Krajzewicz, and R. Nippold. Unambiguous metrics for evaluation of traffic networks. In *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*, pages 1277–1282, Piscataway, NJ, 2010. IEEE Press.

[8] [COLOMBO D2.2, 2014]. COLOMBO project consortium: Policy Definition and dynamic Policy Selection Algorithms. 2014.

[9] [COLOMBO D2.3, 2014]. COLOMBO project consortium: Performance of the Traffic Light Control System for different Penetration Rates. 2014.

[10] [COLOMBO D3.1, 2013]. COLOMBO project consortium: Tuning Tool Kit. 2013.

[11] [COLOMBO D5.3, 2014]. COLOMBO project consortium: Traffic Light Algorithm Evaluation System. 2014.

[12] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, New York, NY, third edition, 1999.

[13] Jérémie Dubois-Lacoste, Manuel López-Ibáñez, and Thomas Stützle. Automatic configuration of state-of-the-art multi-objective optimizers using the TP+PLS framework. In Natalio Krasnogor and Pier Luca Lanzi, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2011*, pages 2019–2026. ACM Press, New York, NY, 2011.

[14] Agoston E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[15] Youssef Hamadi and Marc Schoenauer, editors. *6th International Conference, LION 6, Paris, France, January 16-20, 2012. Selected Papers*, volume 7219 of *Lecture Notes in Computer Science*. Springer, Heidelberg, Germany, 2012.

[16] Holger H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, February 2012.

[17] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Automated configuration of mixed integer programming solvers. In A. Lodi, M. Milano, and P. Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010*, volume 6140 of *Lecture Notes in Computer Science*, pages 186–202. Springer, Heidelberg, Germany, 2010.

[18] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization, 5th International Conference, LION 5*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, Heidelberg, Germany, 2011.

[19] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Parallel algorithm configuration. In Hamadi and Schoenauer [15], pages 55–70.

[20] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009.

[21] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proc. of the Twenty-Second Conference on Artifical Intelligence (AAAI '07)*, pages 1152–1157, 2007.

[22] Christopher H. Jackson. Multi-state models for panel data: The msm package for R. *Journal of Statistical Software*, 38(8):1–29, 2011.

[23] Tianjun Liao. *Population-based Heuristic Algorithms for Continuous and Mixed Discrete-Continuous Optimization Problem*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, 2013.

[24] Tianjun Liao, Marco A. Montes de Oca, and Thomas Stützle. Computational results for an automatically tuned CMA-ES with increasing population size on the CEC'05 benchmark set. *Soft Computing*, 17(6):1031–1046, 2013.

[25] Tianjun Liao, K. Socha, Marco A. Montes de Oca, Thomas Stützle, and Marco Dorigo. Ant colony optimization for mixed-variable optimization problems. *IEEE Transactions on Evolutionary Computation*, 18(4):503–518, 2014.

[26] Tianjun Liao and Thomas Stützle. Benchmark results for a simple hybrid algorithm on the CEC 2013 benchmark set for real-parameter optimization. In *Proceedings of the 2013 Congress on Evolutionary Computation (CEC 2013)*, pages 1938–1944. IEEE Press, Piscataway, NJ, 2013.

[27] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[28] Manuel López-Ibáñez and Thomas Stützle. The automatic design of multi-objective ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation*, 16(6):861–875, 2012.

[29] Manuel López-Ibáñez and Thomas Stützle. Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research*, 235(3):569–582, 2014.

[30] Marie-Eléonore Marmion, Franco Mascia, Manuel López-Ibáñez, and Thomas Stützle. Automatic design of hybrid stochastic local search algorithms. In María J. Blesa, Christian Blum, Paola Festa, Andrea Roli, and M. Sampels, editors, *Hybrid Metaheuristics*, volume 7919 of *Lecture Notes in Computer Science*, pages 144–158. Springer, Heidelberg, Germany, 2013.

[31] O. Maron and A. W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Research*, 11(1–5):193–225, 1997.

[32] Marco A. Montes de Oca, Dogan Aydin, and Thomas Stützle. An incremental particle swarm for large-scale continuous optimization problems: An example of tuning-in-the-loop (re)design of optimization algorithms. *Soft Computing*, 15(11):2233–2255, 2011.

[33] Leslie Pérez Cáceres, Manuel López-Ibáñez, and Thomas Stützle. An analysis of parameters of irace. In *Proceedings of EvoCOP 2014 – 14th European Conference on Evolutionary Computation in Combinatorial Optimization*, Lecture Notes in Computer Science, pages 37–48. Springer, Heidelberg, Germany, 2014.

[34] Marius Schneider and Holger H. Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In Hamadi and Schoenauer [15], pages 190–204.

[35] Sydney Siegel and N. John Castellan, Jr. *Non Parametric Statistics for the Behavioral Sciences*. McGraw Hill, New York, NY, 2 edition, 1988.

[36] S. K. Smit and Agoston E. Eiben. Beating the 'world champion' evolutionary algorithm via REVAC tuning. In H. Ishibuchi et al., editors, *Proceedings of the 2010 Congress on Evolutionary Computation (CEC 2010)*, pages 1–8. IEEE Press, Piscataway, NJ, 2010.

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

[38] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M. Fonseca, and Viviane Grunert da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.