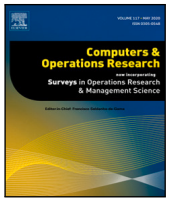
Computers & Operations Research 139 (2022) 105615

Contents lists available at ScienceDirect

Computers and Operations Research

journal homepage: www.elsevier.com/locate/cor



Capping methods for the automatic configuration of optimization algorithms

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

[a] Department of Software Engineering, Santa Catarina State University, Brazil
[b] Institute of Informatics, Federal University of Rio Grande do Sul, Brazil
[c] School of Computer Science, University of Málaga, Spain
[d] Alliance Manchester Business School, University of Manchester, United Kingdom

ARTICLE INFO

Keywords:
Automatic algorithm configuration
Capping methods
Optimization algorithms
Parameter tuning

ABSTRACT

Automatic configuration techniques are widely and successfully used to find good parameter settings for optimization algorithms. Configuration is costly, because it is necessary to evaluate many configurations on different instances. For decision problems, when the objective is to minimize the running time of the algorithm, many configurators implement capping methods to discard poor configurations early. Such methods are not directly applicable to optimization problems, when the objective is to optimize the cost of the best solution found, given a predefined running time limit. We propose new capping methods for the automatic configuration of optimization algorithms. They use the previous executions to determine a performance envelope, which is used to evaluate new executions and cap those that do not satisfy the envelope conditions. We integrate the capping methods into the irace configurator and evaluate them on different optimization scenarios. Our results show that the proposed methods can save from about 5% to 78% of the configuration effort, while finding configurations of the same quality. Based on the computational analysis, we identify two conservative and two aggressive methods, that save an average of about 20% and 45% of the configuration effort, respectively. We also provide evidence that capping can help to better use the available budget in scenarios with a configuration time limit.

1. Introduction

Many algorithms expose parameters that control their internal operation and allow users to adapt their behavior to different scenarios. Their performance depends on the chosen parameter setting (or configuration), and algorithm configuration is the task of finding a good or the optimal setting for a given set of inputs. Algorithms are often configured manually in a trial-and-error process guided by the users' experience. However, a manual search may be biased, can be difficult in irregular parameters landscapes, e.g., when there are interactions between parameters that are hard to foresee, and is time consuming.

The above problems may be overcome by an automated, systematic experimental approach and, hence, substantial effort has been dedicated to the development of methods for automatic algorithm configuration (Birattari, 2003; Hutter et al., 2009, 2011; Ansótegui et al., 2009; López-Ibáñez et al., 2016a), which have been found to be effective in many applications; see Hutter et al. (2010), KhudaBukhsh et al. (2016) and López-Ibáñez and Stützle (2012) for some examples. Although these methods solve a part of the problem, the configuration time remains a bottleneck, since they need to evaluate different parameter values on different (training) instances, and each evaluation often takes a considerable amount of time.

There is a trade-off between the number of evaluations (or instances evaluated) and the quality of the final configurations found, and so reducing the effort will usually lead to worse configurations. A better approach to reduce the configuration time is to evaluate the quality of a configuration during its execution, and immediately stop it when poor performance is expected. This approach has been applied previously to the automatic configuration of decision algorithms (Hutter et al., 2009; Pérez Cáceres et al., 2017a), where the performance of an algorithm is measured by its running time and the goal of configuration is to minimize it. The key idea behind existing methods is to *cap* the execution time, i.e. to determine a bound on the running time based on the best-performing configuration found so far and, if the execution reaches the running time bound, stop it and discard the current configuration.

For optimization problems, we usually execute an algorithm with a predefined stopping criterion, and measure its performance by the cost of the best found solution (assuming, without loss of generality,

* Corresponding author at: Department of Software Engineering, Santa Catarina State University, Brazil.
** Corresponding authors.
E-mail addresses: marcelo.desouza@udesc.br (M. de Souza), marcus.ritt@inf.ufrgs.br (M. Ritt), manuel.lopez-ibanez@uma.es (M. López-Ibáñez).

https://doi.org/10.1016/j.cor.2021.105615
Received 13 May 2020; Received in revised form 2 September 2021; Accepted 23 October 2021
Available online 2 November 2021

minimization of solution cost). Thus, existing capping methods are not suitable for these scenarios. In this paper, we propose capping methods for configuring optimization algorithms. The main idea is to use previously seen executions to determine a performance envelope for the current execution. We then monitor the evolution of the performance of each configuration, and stop it if at some point the conditions defined by the performance envelope are violated. We present and evaluate an implementation of such capping methods in the irace configurator.

The rest of this paper is organized as follows. Section 2 introduces the notation used in this paper, the problem of automatic algorithm configuration, and current available configurators. Section 3 presents and discusses related work. In Section 4 we introduce several new capping methods for optimization problems. Section 5 presents results of extensive computational experiments comparing the different methods, and discusses them. Finally, Section 6 concludes and presents some future research directions.

2. Automatic algorithm configuration

Approaches to automatic algorithm configuration can be categorized into online methods, which try to find the best parameter settings during the execution of the algorithm, and offline methods, which divide the process into training and test phases. In this paper we focus on offline methods. Offline methods search, during a training phase, for the best parameter setting by evaluating the target algorithm with different configurations. When this phase ends, the best found configurations can be used to solve the problem. Usually, offline methods use a set of training instances to configure the algorithm, and then evaluate the resulting configurations on a different set of test instances. Therefore, it is important to select training instances with similar characteristics to the instances that will be solved by the configured algorithm, either in the test phase or in production.

The offline algorithm configuration problem can be formalized as follows. Let $\mathcal{A}$ be a target algorithm with n parameters p_i, $i = 1, \ldots, n$, each one with domain Θ_i. The space of all parameter settings Θ is the subset of $\Theta_1 \times \cdots \times \Theta_n$ of valid parameter combinations. Parameter domains Θ_i may have different types. Categorical parameters can assume a fixed number of values, and are often used to model discrete choices such as algorithmic components. Ordinal parameters have a natural ordering but no definite distance; an example would be the choice of a neighborhood in a local search when ordered by size. Numerical parameters represent real or integer values, e.g. the numerical tolerance in a mathematical solver, or the temperature in Simulated Annealing.

Given a set of instances Π and a metric $c(\theta, \pi)$ that measures the performance of $\mathcal{A}$ with parameter setting (configuration) $\theta \in \Theta$ and instance $\pi \in \Pi$ as input, the algorithm configuration problem is to find a configuration $\theta^* \in \Theta$ that optimizes the expected performance of $\mathcal{A}$ over instances Π. If $\mathcal{A}$ is stochastic, then $c(\theta, \pi)$ is a random variable. There are different choices for the performance metric c. For decision problems, usually the running time is used. For optimization problems, it is common to use the cost of the best found solution, after an execution with a given computational effort, such as running time or number of iterations.

There are several tools for the automatic configuration of algorithms. ParamILS (Hutter et al., 2009) is an iterated local search to explore the parameter space. It repeatedly perturbs a certain number of randomly chosen parameters and then applies a local search in a neighborhood that changes one parameter at a time. SMAC (Hutter et al., 2011) builds a random forest model from configuration runs to predict the performance of new configurations on a given instance and select the most promising ones for evaluation. The results are used to fit the random forest model for the next iteration. GGA (Ansótegui et al., 2009) is a gender-based genetic algorithm to evolve configurations. The population is divided into competitive and non-competitive genders, which guide the behavior of the recombination and mutation operations. A more recent version, GGA++ (Ansótegui et al., 2015), adds non-parametric models to predict promising regions of the parameter space.

Algorithm 1: Iterated racing procedure

Input : Training instances Π, parameter space Θ, performance metric $c(\theta, \pi)$, computational budget B.
Output: Set of best configurations Θ^{elite}.

1 $\Theta^{\text{elite}} \leftarrow \emptyset$
2 **repeat**
3 $\quad \Theta' \leftarrow \texttt{sample}(\Theta, \Theta^{\text{elite}})$
4 $\quad \Theta^{\text{elite}} \leftarrow \texttt{race}(\Theta' \cup \Theta^{\text{elite}}, \Pi, c)$
5 **until** *budget B is exhausted*
6 **return** Θ^{elite}

The irace configurator (López-Ibáñez et al., 2016a) is an iterated racing method based on the Friedman-Race (F-Race) proposed by Birattari (2009) and the iterated F-Race (I/F-Race) proposed by Balaprakash et al. (2007). Algorithm 1 shows the main steps of irace. The algorithm maintains an elite set $\Theta^{\text{elite}} \subseteq \Theta$ of the best found configurations, which guides the generation of new configurations by the `sample` procedure (line 3). Then, the new configurations and the elite ones are evaluated by the `race` procedure (line 4) on a subset of the instances Π according to the performance metric $c(\theta, i)$. The result of `race` is a new set of elite configurations. The elite set may be unchanged if no better configuration is found. Sampling and racing alternates until a given computational budget is exhausted. The number of iterations and the available budget per iteration is defined at the beginning of the execution. The budget can be set to either a maximum number of evaluated configurations or, different from I/F-Race, a maximum configuration time.

The `sample` procedure is based on the elite configurations Θ^{elite}. At the beginning, when Θ^{elite} is empty, it samples the parameter space Θ uniformly. In subsequent iterations, the elite configurations are ranked according to the observed quality in previous evaluations, and iteratively one of them is selected to generate a new configuration θ, where elite configurations of a higher rank have a higher probability of being selected. Then, a new value of each parameter in θ is determined based on the value of the selected elite configuration and on a truncated normal distribution for continuous parameters, or a discrete probability distribution for discrete parameters. The parameters of these distributions are adjusted over the iterations, such that with an increasing number of iterations, the generated configurations get more similar to those of their parents.

The `race` procedure starts evaluating each configuration on a subset of the available instances. After executing the configurations on a predefined number of instances, irace uses either the non-parametric Friedman test or the paired t-test to identify worse configurations to be discarded. The racing method in irace extends I/F-Race by reusing the evaluations from previous races within the current race and by preventing elite configurations from being discarded without considering all their evaluations from previous races. The surviving configurations are evaluated on another instance, and a new statistical test is performed. This process is repeated until the budget of this iteration is exhausted, or a minimum number of surviving configurations remains.

There are several studies using irace for configuring algorithms for different optimization problems (e.g Pagnozzi and Stützle, 2019; Blum et al., 2015; Franzin and Stützle, 2019), or to minimize their running time, e.g. Pérez Cáceres et al. (2017c). See López-Ibáñez et al. (2016a) for more applications and further details about irace.

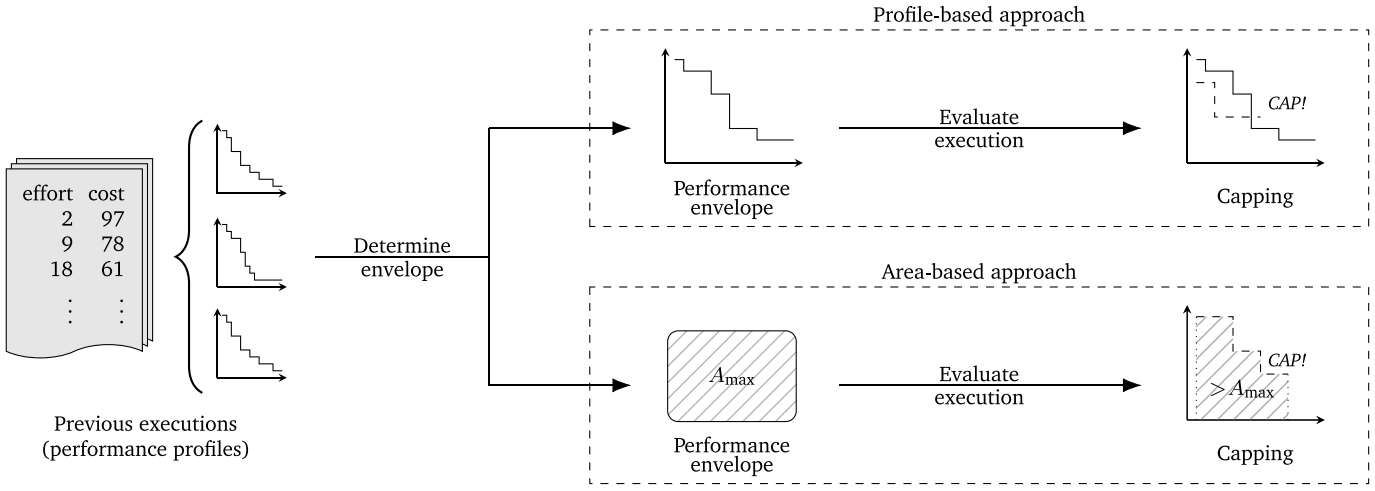


Fig. 1. Overview of the capping methods. Left: for each instance previous performance profiles are kept. Upper right: in the profile-based approach these performance profiles are combined into a performance envelope. If the current performance profile (dashed line) leaves the envelope (solid line) it is capped. Lower right: in the area-based approach profiles are combined into a maximum allowed area $A_{\max}$. If the total area of the current performance profile (dashed line) exceeds $A_{\max}$ it is capped.

3. Related work

Hutter et al. (2009) proposed capping methods for ParamILS to be used for configuration scenarios minimizing running time. The best configuration θ' of the current iteration of ParamILS is used to determine a cut-off running time for subsequent executions. A new configuration θ must be executed (and present better performance) on the same instances that θ' has been executed to replace it and become the new best configuration. If the time used by θ to solve a subset of those instances exceeds the time used by θ' to solve all of them, θ is discarded before the complete evaluation. Hutter et al. (2009) call this method *trajectory-preserving capping*, since it discards only configurations that would be discarded after the complete evaluation. It reduces the configuration time, but does not change the search trajectory. A second approach, called *aggressive capping*, considers not the best found configuration of the current iteration, but the best configuration overall θ^*. In this case, the cut-off time when evaluating a new configuration θ is b times the mean running time of θ^*. Multiplier b defines the aggressiveness of the capping method. Although the aggressive capping method may change the search trajectory, it can also lead to large savings in the configuration time.

Based on the above ideas, Pérez Cáceres et al. (2017a) integrated a capping method for configuring decision algorithms into irace. Consider a new configuration θ that will be executed on instance π_i and was previously evaluated on instances $\pi_1, \pi_2, \ldots, \pi_{i-1}$. The cut-off time t^c is given by $t^c = i \cdot t_i^{\text{elite}} + t^{\min} - (i-1)t_{i-1}^{\theta}$, where t_i^{elite} is the median running time of the elite configurations Θ^{elite} on instances $\pi_1, \pi_2, \ldots, \pi_i$, t_{i-1}^{θ} is the average running time of configuration θ on instances $\pi_1, \pi_2, \ldots, \pi_{i-1}$, and $t^{\min} > 0$ is the minimally measurable running time. The cut-off time can be seen as the maximum time available for θ to improve over the performance of the elite configurations. Pérez Cáceres et al. (2017a) also proposed the following dominance-based elimination criterion. A configuration θ is dominated if $t_i^{\text{elite}} + t^{\min} < t_i^{\theta}$. Whenever all configurations have been evaluated on a new instance π_i, the dominated configurations are eliminated.

The capping methods proposed for ParamILS (Hutter et al., 2009) and irace (Pérez Cáceres et al., 2017a) are designed for decision problems, when the goal is to minimize the running time of the target algorithm. Those capping methods cannot handle optimization scenarios, where the solution cost may be improved over time and, thus, there is information about the progress (or lack thereof) of the algorithm. In an optimization context, Karapetyan et al. (2018) propose an approach to approximate the 1% best configurations of optimization algorithms based on short runs. They uniformly sample and evaluate 1% of the configuration space, determining a performance envelope, which is the hull defined by the worst solution cost obtained in those executions at each point in time. Then, previously untested configurations are executed. If, at some point in time, the cost of the best found solution is more than 20% worse than the one defined by the performance envelope, the execution is stopped and the configuration is discarded. If the configuration survives, it replaces the worst performing element of the 1% pool (according to the final solution cost). Among the capping methods proposed here, we include this definition of performance envelope as a particular case. Moreover, our capping methods are designed to work within the existing algorithm configuration techniques, e.g. the racing mechanism of irace, instead of relying on uniform sampling.

Capping methods for optimization scenarios analyze the performance of the configurations during their execution, thus, they require that the configurations produce feasible solutions prior to completion and continuously produce improved solutions until completion. Capping methods are not useful for configuring algorithms that do not satisfy these requirements, such as those that only return a single solution, that need a long exploratory phase or are based on random restarts. The ability of an optimization algorithm to produce as good solution cost as possible at any point during its execution is called its anytime behavior (Zilberstein, 1996; López-Ibáñez and Stützle, 2014). Fortunately, many practical optimization algorithms are anytime algorithms. Some techniques used to find configurations that present good anytime behavior are similar to those implemented by capping methods. For example, Branke and Elomari (2011) use a higher-level genetic algorithm to search for parameter settings of a lower-level genetic algorithm aiming to optimize its anytime behavior. To evaluate the population of configurations, they analyze their performance profiles. Configurations that present the best solution cost for some time point are ranked first. These configurations are removed from the population and the process is repeated, but now the best remaining configurations are ranked second, and so on. The ranks are used to guide the selection of configurations. Although their proposal is not a capping method, one of our methods similarly builds an envelope by considering the best solution cost found at any running time (effort) point.

López-Ibáñez and Stützle (2014) also explore the automatic configuration of algorithms to improve their anytime behavior. They model this configuration task as a bi-objective optimization problem, considering the performance profiles as a set of nondominated bi-objective points. They use irace to find configurations that maximize the hypervolume of such nondominated sets. The results show that the proposed

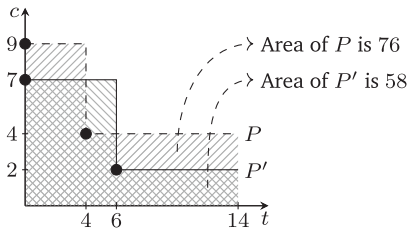


Fig. 2. Example of profile- and area-based capping behaviors.

techniques are effective in improving the anytime behavior of algorithms for two different scenarios. We propose here an area-based measure to determine the performance envelope (Section 4.2) that is equivalent to the hypervolume metric used in López-Ibáñez and Stützle (2014).

4. Capping for optimization scenarios

The capping methods proposed in this paper use the performance profiles of previously seen configurations to determine a minimum performance bound for new executions, and then stop poor performers early. The performance profile of a run is given by the solution cost as a function of the effort $P : \mathbb{R}_+ \to \mathbb{R}$, where $P(t) = c$ denotes the cost c of the best found solution after spending computational effort t. Any effort measure, e.g. the running time, number of iterations, or number of objective function evaluations, can be used. Assuming a minimization problem and an anytime algorithm that iteratively evaluates solutions and remembers the best solution found, a profile will be a monotonically decreasing step function. Thus, in practice, we only need to collect the points (t, c) at efforts t where c decreases. Fig. 1 presents an overview of the capping method, which is applied before executing each configuration. The first step of this process is to obtain all performance profiles of the previous executions on the current instance. They are used to determine a performance envelope, which represents the minimum performance required for this instance. If at some point the observed performance profile is worse than the envelope, the execution is stopped. Unlike capping methods for decision problems, our methods do not cap after a fixed running time. Instead, they monitor the progress of the execution and terminate it when appropriate.

We propose two different types of envelopes. The *profile-based envelope* is represented by a performance profile, which determines the maximum allowed solution cost throughout the used effort. When using this method to evaluate an execution, as soon as its performance profile exceeds the envelope, i.e., the solution cost of the execution is worse than the one defined by the envelope for some value of effort, the execution is capped (see *Profile-based approach* in Fig. 1). This approach not only defines the limits in terms of the expected solution cost, but also the acceptable behavior of the performance profile. For example, it is not allowed to be worse than the envelope in the beginning of the execution, e.g., by trying to diversify the search to find better solutions at a later time in the execution. Hence, this approach implies that the execution must show a clear good anytime behavior. In other cases, the final solution cost is the only performance criterion that matters, and allowing a worse performance in the beginning is not a problem, as long as a good solution is found in the end. Stützle et al. (2012), for example, studied parameter adaptation techniques for ant colony optimization algorithms. Their results on the traveling salesperson problem show that for some parameter settings, worse performance in the beginning of the execution leads to better final solutions.

In the *area-based envelope*, instead of dealing directly with the performance profiles, we consider the area defined by them. In this case, the area under the performance profiles of previous executions is used

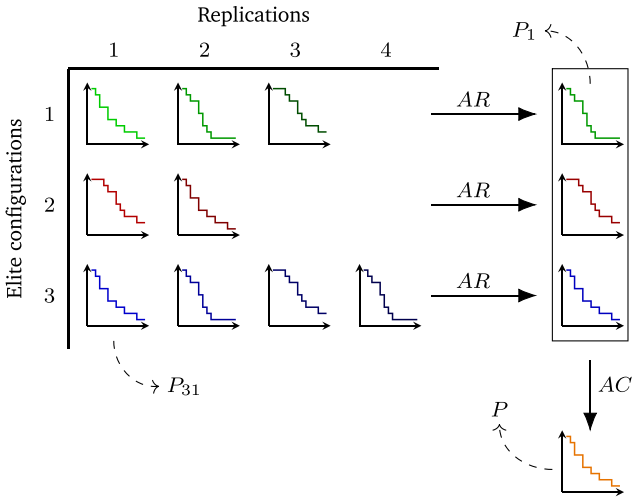


Fig. 3. Aggregation scheme for elitist capping methods.

to determine a maximum area for the current execution. This maximum area value is the envelope. The current execution will be capped if the area of its performance profile exceeds the envelope, i.e. the maximum area available (see *Area-based approach* in Fig. 1). As long as the maximum area is not exceeded, the performance profile can present different behaviors. For example, a configuration can produce worse solutions in the beginning of its execution in comparison to previously seen configurations, but produce better solutions towards the end of the execution, maintaining the total area within the envelope. Fig. 2 illustrates this situation for the performance profile P' of a running execution. With a profile-based envelope P, execution P' would be capped at effort $t = 4$, since the solution cost of P' at that point exceeds the corresponding solution cost of envelope P. However, when defining the envelope as the area under P at $t = 14$, then P' would not be capped, since its area is always smaller than the envelope.

It is important to consider that an unsatisfactory performance of a capped configuration on the current instance does not imply a similar performance on other instances. Therefore, instead of directly discarding the configuration, we just stop the execution and return to irace the cost of the best solution found. In other words, the configuration is penalized by having a reduced execution effort, but it can compensate this by presenting a better performance on other instances. The decision of discarding configurations is delegated to irace.

4.1. Profile-based envelope generation

The profile-based methods define the envelope as a performance profile. We propose two different strategies to compute the envelope: *elitist* and *adaptive*. For both approaches, we first select a subset of the non-capped previous executions on the current instance. Then, we aggregate their performance profiles into a performance envelope. The elitist envelope generation is based on the executions of configurations from an elite set, while the adaptive strategy considers all previous executions and discards some of them according to an aggressiveness value. Another difference between elitist and adaptive strategies is the behavior when evaluating elite configurations. Elitist strategies use those configurations to determine the performance envelope, and never cap executions of elite configurations. Adaptive methods, on the other hand, do not differentiate configurations when determining the envelope and when evaluating them. Therefore, adaptive strategies can cap both elite and non-elite configurations.

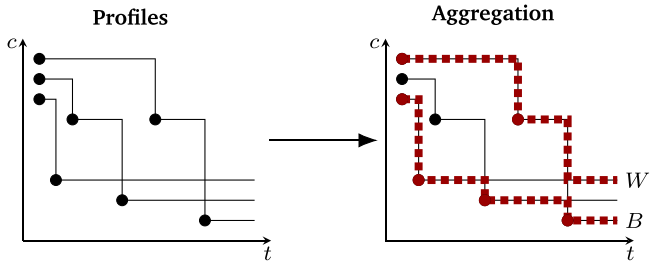


Fig. 4. Best (B) and worst (W) aggregation methods for profile-based envelopes.

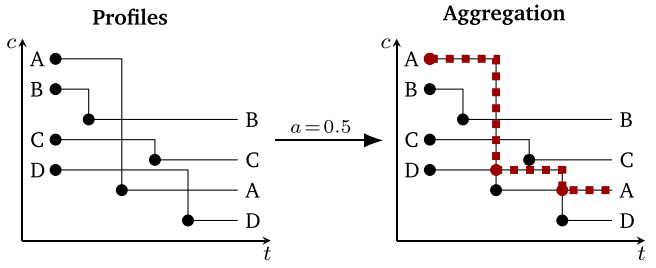


Fig. 5. Adaptive aggregation for profile-based envelope.

4.1.1. Elitist profile-based envelope generation

The elitist strategy uses only the best performing (elite) configurations to define the envelope. In irace, the elite configurations are those that survived the previous race. We divide the aggregation of the previous executions for a fixed instance in two steps, as illustrated in Fig. 3. Let us consider a set of n configurations $\theta_1, \ldots, \theta_n$, where each configuration θ_i has been evaluated m_i times on the same instance. Let $\mathbb{P}$ be the space of all possible performance profiles and $P_{ij} \in \mathbb{P}$ be the performance profile of the jth replication of θ_i on that instance. We define the aggregated performance profile for configuration θ_i as $P_i = AR(P_{i1}, P_{i2}, \ldots, P_{im_i})$, where $AR\colon 2^{\mathbb{P}} \to \mathbb{P}$ is an function that aggregates the performance profiles of multiple runs of a configuration. The profile-based envelope is defined as $P = AC(P_1, P_2, \ldots, P_n)$, where $AC\colon 2^{\mathbb{P}} \to \mathbb{P}$ is a function that aggregates the performance profiles of multiple configurations.

Two possible approaches for defining AR and AC are the *worst* aggregation function W and the *best* aggregation function B. Function $W\colon 2^{\mathbb{P}} \to \mathbb{P}$ generates a performance profile that selects, for each effort t, the pointwise maximum solution cost among all input performance profiles. Given a set of performance profiles $P_1, P_2, \ldots, P_k$, the performance profile generated by W is given by

$$W(P_1, \ldots, P_k)(t) = \max\{P_1(t), \ldots, P_k(t)\}. \tag{1}$$

Analogously, the best aggregation function $B\colon 2^{\mathbb{P}} \to \mathbb{P}$ generates a performance profile that selects, for each value of effort t, the pointwise minimum solution cost among all performance profiles, then

$$B(P_1, \ldots, P_k)(t) = \min\{P_1(t), \ldots, P_k(t)\}. \tag{2}$$

Aggregation methods W and B are illustrated in Fig. 4. There are three different performance profiles to be aggregated (left side). The aggregated performance profiles given by W and B functions are shown on the right side of the figure. W and B form the hull of the input performance profiles. Function W produces the pessimistic performance (least aggressive profile), while function B produces the optimistic performance (most aggressive profile).

Aggregation functions AR and AC may be different or the same. For example, if $AR = W$ and $AC = B$, then the envelope will be

$$\begin{aligned} P(t) &= AC(AR(P_{11}, \ldots, P_{1m_1}), \ldots, AR(P_{n1}, \ldots, P_{nm_n}))(t) \\ &= B(W(P_{11}, \ldots, P_{1m_1}), \ldots, W(P_{n1}, \ldots, P_{nm_n}))(t) \\ &= \min\{\max\{P_{11}(t), \ldots, P_{1m}(t)\}, \ldots, \max\{P_{n1}(t), \ldots, P_{nm}(t)\}\} \end{aligned} \tag{3}$$

We also propose a model-based aggregation function as follows. Given the performance profile P of a single run and a maximum cut-off effort $t_{\max}$ (maximum budget for any run) such that $P(t) \geq P(t_{\max})$, $\forall t \geq 0$, let $T(P, c)$ be the smallest effort to reach target c defined as

$$T(P, c) = \begin{cases} \min\{t \geq 0 \mid P(t) \leq c\}, & \text{if } P(t_{\max}) \leq c, \\ \alpha t_{\max}, & \text{otherwise,} \end{cases} \tag{4}$$

where $\alpha \geq 1$ is a penalty factor applied when the performance profile does not reach the target, similar to the PARX penalization approach (Pérez Cáceres et al., 2017a). We now assume that the value of $T(P, c)$ for any randomly selected run P of an algorithm is a random variable $\mathcal{T}(c)$ that follows an exponential distribution, which is often a reasonable assumption for optimization algorithms (Hoos and Stützle, 2005). Its empirical cumulative distribution function is given by $F(t; \lambda) = \Pr(\mathcal{T}(c) \leq t) = 1 - e^{-\lambda t}$, where λ is the parameter of the distribution. Given the performance profiles $P_1, \ldots, P_k$ of k runs of the algorithm, we estimate the mean effort to reach target c as $\bar{T}(c) = \sum_i^k T(P_i, c)/k$. Then, the maximum likelihood estimator for parameter λ is $\hat{\lambda}(c) = 1/\bar{T}(c)$ and we can determine the effort $T_p(c)$ required for a fraction $p \in [0, 1]$ of the executions not reaching target value c by setting $F(T_p(c); \lambda) = 1 - p$, which holds for $T_p(c) = -\ln(p) \cdot \bar{T}(c)$. Since $T_p(c)$ is monotone, it has an inverse $T_p^{-1}(t)$ that gives the expected c reached after effort t by at most a fraction $1 - p$ of executions. We can now define a model-based aggregation as

$$M(P_1, \ldots, P_k; p)(t) = T_p^{-1}(t). \tag{5}$$

If there is only one performance profile to be aggregated i.e. $k = 1$, methods W and B produce the same performance profile, since they simply select cost values for each value of effort from the available profile. In contrast, method M computes the aggregated performance profile based on the exponential model, whose parameter can be estimated even with a single sample. For example, given a single performance profile P to be aggregated with $p = 0.1$ and $T(P, c) = 20$ for a particular target cost c, the mean $\bar{T}(c) = 20$ is multiplied by $-\ln(0.1) \approx 2.3$, leading to an effort equal to 43 for the aggregated performance profile. The effort values required for the fraction p not reaching all target costs are computed, producing an aggregated performance profile different from the one used as input.

4.1.2. Adaptive profile-based envelope generation

The adaptive strategy determines the envelope based on all previous executions on the current instance, not only on the executions of elite configurations. These performance profiles are ordered by the cost of the best solution found, and the envelope is determined in such a way that only the best ones would not be capped. The number of such non-capped performance profiles is determined by an aggressiveness parameter $a \in [0, 1]$. Given k performance profiles sorted by the cost of their best solution found, we determine the most aggressive envelope that would cap a fraction a of them. That is, the adaptive profile-based envelope generation is equivalent to the aggregation function

$$D(P_1, \ldots, P_k)(t) = W(P_1, \ldots, P_{\lceil (1-a)k \rceil})(t), \tag{6}$$

given that $P_1, P_2, \ldots, P_k$ are ordered by the cost of their best solution found. Fig. 5 shows an example with 4 performance profiles P_A, P_B, P_C, P_D. When sorting them, we get the ordered list $L = (P_D, P_A, P_C, P_B)$. Given $a = 0.5$, we select only the 2 best performance profiles and compute the most aggressive envelope that would not cap them by applying W function (Eq. (1)). The resulting envelope is shown on the right side of Fig. 5.

Given a value of a for the current iteration, we create an envelope before each execution, and use it to evaluate (and possibly cap) that execution. However, we cannot ensure that this value of a will produce

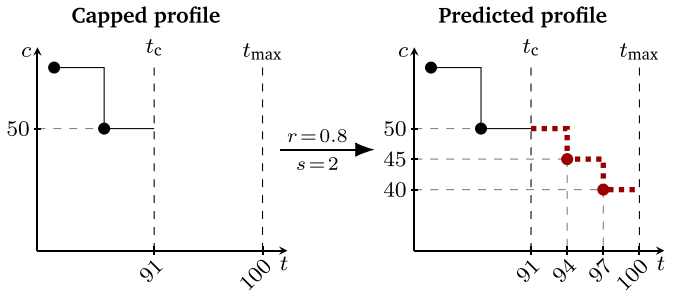


Fig. 6. Example of the performance profile prediction for adaptive profile-based envelope generation.

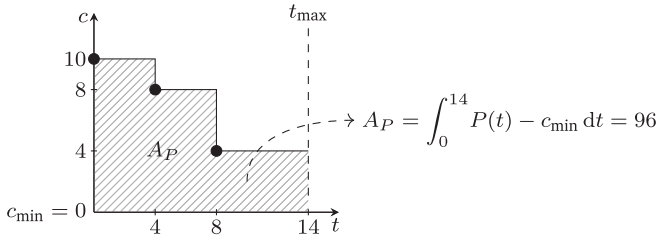


Fig. 7. Example of area calculation.

envelopes that cap exactly a fraction a of those executions. Therefore, once the current iteration is finished, we adapt the value of a based on the number of executions capped, and then use the updated value to compute the envelopes in the next iteration, in an attempt to reach a user-defined aggressiveness goal a_g.

Let a_c be the fraction of executions capped in the previous iteration of irace. Since the goal was to cap a fraction a_g of those executions, we increase a to the lowest value that would have capped a_g, if $a_c < a_g - \varepsilon$; decrease a to the highest value that would have capped a_g, if $a_c > a_g + \varepsilon$; or maintain a, otherwise, where $\varepsilon \in [0, 1]$ is a user-defined parameter that specifies the tolerance of the observed aggressiveness deviation from the aggressiveness goal a_g. For example, given $a_g = 0.3$ and $\varepsilon = 0.05$, if in the previous iteration we capped 18 out of 100 executions, then the adaptation procedure would increase a, since $a_c = 18/100 < 0.3 - 0.05$, such that Eq. (6) would cap exactly $0.3 \cdot 100 = 30$ executions.

When increasing a, we can easily determine which value of a would cap the desired number of executions from the previous iteration by adding performance profiles one at a time to those used by Eq. (6). The case of decreasing a is more complicated since we want to find the value a that would not cap the desired number of executions. Because the executions were capped, we do not know the performance profile until the cut-off effort and, therefore, we cannot compute exactly the value of a that would not ever cap them. Instead, we predict the unknown part of the performance profile and combine it with the known part when calculating the value of a that would not cap it.

To predict how a capped performance profile P_c would behave from the effort at which it was capped t_c until the cut-off effort t_{max}, we consider all previous non-capped performance profiles on the same instance, and compute from them a simple extrapolation. First, we estimate the number of improvements s of the objective function from t_c to t_{max} as the median number of improvements in the same range over all non-capped profiles. Next, we estimate the final solution cost $\hat{P}_c(t_{max}) = r \cdot P_c(t_c)$, where r is the median of the improvement ratios $P(t_{max})/P(t_c)$ between the solution cost at t_c and the final cost at t_{max} over all non-capped performance profiles. Finally, we estimate the solution cost at each effort $t_i = t_c + i(t_{max} - t_c)/(s + 1)$, with $i = 1, \ldots, s$, as

$$\hat{P}_c(t_i) = P_c(t_c) + i(\hat{P}_c(t_{max}) - P_c(t_c))/s. \tag{7}$$

Fig. 6 presents an example of the performance profile prediction. On the left side we can see the performance profile P_c capped at $t_c = 91$ with cost $P_c(t_c) = 50$. Let us assume that $r = 0.8$ and $s = 2$, thus the predicted final cost at $t_{max} = 100$ is $\hat{P}_c(t_{max}) = 50 \cdot 0.8 = 40$. Then, we build $s = 2$ intermediary points until t_{max}. The first will be at $t_1 = t_c + 1 \cdot (t_{max} - t_c)/(s + 1) = 91 + 9/3 = 94$ with cost $\hat{P}_c(t_1) = P_c(t_c) + 1 \cdot (\hat{P}_c(t_{max}) - P_c(t_c))/s = 50 + 1 \cdot (-10/2) = 45$. The second point will be at $t_2 = t_c + 2 \cdot (t_{max} - t_c)/(s + 1) = 91 + 2 \cdot (9/3) = 97$ with cost $\hat{P}_c(t_2) = P_c(t_c) + 2 \cdot (\hat{P}_c(t_{max}) - P_c(t_c))/s = 50 + 2 \cdot (-10/2) = 40$. The predicted performance profile, with the calculated points, can be seen on the right side of Fig. 6.

4.2. Area-based envelope generation

The area-based envelope is defined as the maximum area available for the execution. The area of a performance profile P is

$$A_P = \int_{t_s}^{t_f} P(t) - c_{min}\,dt, \tag{8}$$

where t_s and t_f are the start and final effort values, respectively, and c_{min} is a baseline cost. The start effort t_s should be the same for calculating the area of different performance profiles, to ensure a fair comparison between them. In our implementation, the target algorithms report the corresponding cost as soon as the first solution is obtained. Then, when evaluating k performance profiles, we define $t_s = \max_{i \in \{1,\ldots,k\}} t_s^i$, where t_s^i is the starting effort value of performance profile i. The final effort t_f is the cut-off effort (t_{max}) for finished executions or the current effort of the execution in progress. Fig. 7 illustrates a performance profile and the respective value of area.

The area depends on the baseline cost c_{min}. If c_{min} is too low, we obtain a larger area, which makes capping less aggressive; if it is too high, we may have negative areas, and possibly a too aggressive capping. Ideally, we would like to set c_{min} to the optimal solution cost or a good lower bound. These are often unknown, thus we maintain for every instance the best found solution cost and set c_{min} to it.

Finally, we first compute the area of previous performance profiles on the current instance, and then aggregate the areas into the envelope, which is represented here by the area budget A_{max}. The current execution P will be capped as soon as $A_P > A_{max}$. Similar to the profile-based methods, in the area-based envelope generation we also use elitist and adaptive aggregation strategies.

4.2.1. Elitist area-based envelope generation

Given all performance profiles of the elite configurations on the current instance, the area-based elitist strategy calculates their area and uses functions AR and AC to aggregate replications and configurations, respectively. We use worst and best approaches for both AR and AC aggregation steps, replacing the pointwise behavior by the selection of the largest area

$$W(P_1, \ldots, P_k) = \max\{A_{P_1}, \ldots, A_{P_k}\}, \tag{9}$$

or the smallest area

$$B(P_1, \ldots, P_k) = \min\{A_{P_1}, \ldots, A_{P_k}\}. \tag{10}$$

4.2.2. Adaptive area-based envelope generation

The area-based adaptive envelope generation is similar to the profile-based approach. We compute the area of all previous performance profiles of the current instance, and select the area which would cap a part of them, according to the aggressiveness parameter $a \in [0, 1]$. In this case, the performance profiles are sorted by their area values. Given a list of performance profiles $P_1, P_2, \ldots, P_k$ ordered such that $A_{P_i} \leq A_{P_{i+1}}$, the adaptive area-based envelope generation is given by

$$D(P_1, \ldots, P_k) = A_{P_{\lfloor (1-a)k \rfloor}}. \tag{11}$$

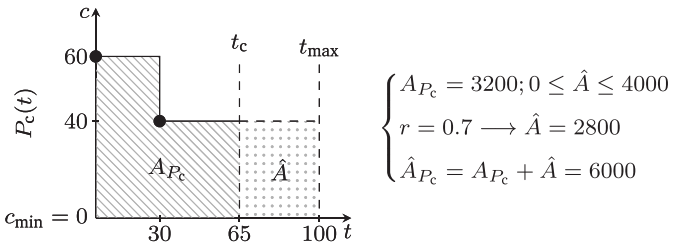


Fig. 8. Example of area prediction.

The aggressiveness is adjusted at the beginning of each iteration. It increases or decreases a according to the amount of capping reached in the last iteration, the aggressiveness goal parameter $a_g \in [0, 1]$, and the tolerance ε.

As done in the profile-based approach, when the amount of capped executions in the previous iteration is out of the range $[a_g - \varepsilon, a_g + \varepsilon]$, we calculate the value of a that would cap exactly the number of executions needed to achieve the aggressiveness goal a_g. The only difference with respect to the profile-based approach is in the estimation required when decreasing a. Here we need to estimate the area of the capped performance profile P_c by predicting its behavior after the capping point $(t_c, P_c(t_c))$. In the best case, the best possible solution would have been found just at t_c, such that $P_c(t_c) = c_{\min}$, and thus the remaining area would be zero. In the worst case, no solution better than $P_c(t_c)$ would have been found until $t_{\max}$, thus the upper bound of the unknown area would be $(P_c(t_c) - c_{\min}) \cdot (t_{\max} - t_c)$. Let $\hat{A}_{P_c} = A_{P_c} + \hat{A}$ be the (estimated) total area used by P_c if it had not been capped, where A_{P_c} is the known area until t_c and $\hat{A} \in [0, (P_c(t_c) - c_{\min}) \cdot (t_{\max} - t_c)]$ is the unknown area between t_c and $t_{\max}$, which must be estimated. Fig. 8 shows an example where the execution was capped at effort $t_c = 65$ with $P(t_c) = 40$. Then, the real area of this execution until t_c is $A_{P_c} = 3200$ and the remaining area is $0 \leq \hat{A} \leq 4000$.

To estimate the value of $\hat{A}$, we use the performance profiles of non-capped previous executions on the current instance. For each performance profile P, we compute the real area A' from t_c to $t_{\max}$ and the upper bound $A'' = (P_c(t_c) - c_{\min}) \cdot (t_{\max} - t_c)$, and then calculate the ratio A''/A'. We use the median ratio r to estimate the remaining area of the capped performance profile P_c as $\hat{A} = r \cdot (P_c(t_c) - c_{\min}) \cdot (t_{\max} - t_c)$ and its predicted area $\hat{A}_{P_c} = A_{P_c} + \hat{A}$. In the example of Fig. 8, by using $r = 0.7$, the remaining area is estimated as $\hat{A} = 2800$, giving a predicted total area of $\hat{A}_{P_c} = 6000$.

Table 1 summarizes the components of all the methods described in this section. A complete capping method consists of an envelope type (P or A), an envelope generation strategy (E or D), the corresponding aggregation functions (W, B or M, when applied), and parameter values (p and a_g, when applied).

The capping methods proposed here are mostly independent of the configurator, and can be applied whenever the following requirements are met. First, the target algorithm must periodically report the progress of the objective function. It must also report the effort if it is different from wall-clock time, e.g., the number of evaluations. Second, the elitist capping methods require that elite configurations are identified. Third, the adaptive capping methods require to indicate when the aggressiveness parameter needs to be updated. The latter two requirements can be satisfied by the configurator but may also be implemented in additional external components.

When integrated with irace, the capping methods identify the elites from the data already reported by irace, update the aggressiveness at the end of each race (Algorithm 1), and do not apply capping in the first race. This integration does not require any changes in irace except using the capping methods as a wrapper around the target algorithm.

Table 1
Summary of capping methods proposed in this article.

Envelope type	Strategy	Parameters
Profile (P)	Elitist (E)	AR = $\{W, B, M\}$, AC = $\{W, B\}$, $p \in [0, 1]$, $\alpha \in [1, \infty)$.
	Adaptive (D)	$a_g \in [0, 1]$, $\varepsilon \in [0, 1]$.
Area (A)	Elitist (E)	AR = $\{W, B\}$, AC = $\{W, B\}$.
	Adaptive (D)	$a_g \in [0, 1]$, $\varepsilon \in [0, 1]$.

5. Computational experiments

In this section, we detail the configuration scenarios, including the target algorithms and the benchmark instances (Section 5.1). Then, our first experiment evaluates how good the capping methods are in saving effort during the tuning process, as well as in finding good configurations (Section 5.2). We identified two conservative, robust methods and analyzed their behavior in detail (Section 5.3). Finally, we assess the contributions of the capping methods when using the total execution time as budget for irace (Section 5.4).

5.1. Experimental setup

We evaluated the proposed capping methods on different configuration scenarios together with irace version 3.1 with its default parameter values. The capping methods have been implemented as a Python 3 script (tested with Python 3.6.8) that can be used together with irace and does not require any changes to irace. The source code and documentation are available at https://github.com/souzamarcelo/capopt (De Souza et al., 2020). For the aggregation method using the exponential model (Eqs. (4) and (5)), we used the penalty constant $\alpha = 10$. For the adaptive methods, we used an aggressiveness goal $a_g = 0.5$ and a tolerance of the deviation from that goal $\varepsilon = 0.05$. When experiments are replicated with different random seeds, the same initial seed is used for all executions of irace in the same replication.

We selected six configuration scenarios from the literature. They consist in heuristic algorithms to solve the traveling salesperson problem, graph coloring, bin packing, binary quadratic programming, and an exact solver for mixed integer programming (MIP) applied to the combinatorial auction winner determination problem. The instances used for the training and test are always different. The input parameters are numeric (integer or real) and categorical. All heuristic algorithms were implemented in C or C++ and compiled with the GNU C/C++ compiler version 7.0.4 with maximum optimization. Table 2 summarizes the characteristics of each scenario: the budget used in the experiments, whether the algorithm is deterministic or stochastic, the unit and limit of the effort measure of the algorithm, the number of integer, categorical, real, and conditional parameters of the algorithm, and the number of training and test instances. We give further details on each scenario below. All files to reproduce the experiments can be found in the supplementary page (De Souza et al., 2021a), including the source code of the capping methods and the target algorithms, the training and test instances with the corresponding best known solution costs, parameter description, and irace input settings for all scenarios.

ACOTSP This solver implements several ant colony optimization (ACO) algorithms applied to the symmetric traveling salesperson problem (TSP). All algorithms are described in Dorigo and Stützle (2004), and the source code can be obtained in Stützle (2002) (we used ACOTSP version 1.03). ACOTSP is part of the AClib benchmark library for algorithm configuration (Hutter et al., 2014) and is widely used as a testbed for studying automatic algorithm configuration (see López-Ibáñez and Stützle, 2014; Pérez Cáceres et al., 2014, 2015, and López-Ibáñez et al., 2018 for some examples).

This scenario has 11 parameters, 5 of them being conditional. We used 60 s of wall clock time as termination criterion of ACOTSP. We defined 2000 executions as budget for irace and used the instances

Table 2
Summary statistics of the configuration scenarios. Column *Det.* indicates whether the target algorithm is deterministic or stochastic.

Scenario	Budget	Det.	Effort		Parameters				Instances	
			Type	Limit	Int	Cat	Real	Cond	Train	Test
ACOTSP	2000	no	time	60 s	4	3	4	5	50	200
HEACOL	2000	no	checks	10^9	4	2	1	0	27	79
TSBPP	500	yes	iterations	5000	3	2	1	0	20	500
HHBQP	2000	no	time	20/30 s	10	3	1	7	9	10
LKH	2000	no	time	10 s	12	9	0	0	50	250
SCIP	2000	yes	time	30 s	0	207	0	0	50	50

provided in López-Ibáñez et al. (2016b), which consist of Euclidean TSP instances of size 2000. A total of 400 instances are available, 200 for training and 200 for test. We selected 50 out of the 200 training instances at random for the configuration step, and selected all 200 test instances for the evaluation step.

HEACOL This scenario concerns the configuration of a hybrid evolutionary algorithm (HEA) for the graph coloring problem (COL). This algorithm was proposed by Galinier and Hao (1999) and detailed in Lewis (2016a). It combines a population of solutions, which are evolved by a problem-specific recombination operator, with a local search procedure. The source code of HEACOL is provided by Lewis (2016b).

The HEACOL scenario has 7 unconditional parameters. The termination criterion is a maximum number of constraint checks. A constraint check is counted whenever the algorithm requests some information about the instance, e.g., whether two vertices are neighbors. We used a termination condition of 10^9 constraint checks and a budget of 2000 executions. For the training, we generated 27 instances, which consist of randomly generated graphs with all combinations of sizes $n \in \{250, 500, 1000\}$ and densities $d \in \{0.1, 0.5, 0.9\}$, where each of pair of vertices are made adjacent with probability d. For the test, we used the 79 well known graph instances available in Trick (2018), with sizes ranging from 11 to 1000 vertices.

TSBPP This scenario concerns a tabu search (TS) algorithm for the two- and three-dimensional bin packing problems (BPP) (Delorme et al., 2016) proposed by Lodi et al. (1999). The source code is described in Lodi et al. (2004a) and is available in Lodi et al. (2004b).

This scenario has 6 unconditional parameters. The termination criterion is the number of iterations of the tabu search. We used a maximum of 5000 iterations. TSBPP is a deterministic algorithm. This scenario also has the smallest parameter space. Because of that, we set a budget of only 500 executions for irace. We used the instances of the two-dimensional bin packing problem (2BPP) proposed by Berkey and Wang (1987) and Martello and Vigo (1998), which are divided in ten different classes. A complete description of these instances can be found in Lodi et al. (1999). All 500 instances were used for the test phase, and 20 out of them were selected for the training phase (we randomly selected two instances of each class). Additional information about these and other instances, as well as other approaches to solve the BPP can be found in Delorme et al. (2018).

HHBQP This scenario consists in a hybrid heuristic (HH) algorithm to solve the unconstrained binary quadratic programming (BQP) (see Kochenberger et al., 2014 and Beasley, 1998). This algorithm was automatically generated by De Souza and Ritt (2018a) using a grammar-based automatic algorithm design technique based on algorithmic components from the literature (Palubeckis, 2006; Glover et al., 2010; Wang et al., 2012). The source code is available in De Souza and Ritt (2018b).

The scenario has 14 parameters, with 7 being conditional. We used a time limit of 20 s for the training executions, 30 s for the executions in the test phase, and a budget of 2000 total executions for irace. For the test phase, we used the 10 instances of size 2500 of Beasley (1998). They can be downloaded from Wiegele (2007b) and more details can be found in Wiegele (2007a). For the training phase, we randomly generated 9 instances with the same structure as Beasley's instances. We generated three instances for each size $n \in \{2000, 2500, 3000\}$, with a density of 0.1 and integer coefficients uniformly sampled within $[-100, 100]$.

LKH This scenario concerns the configuration of the Lin–Kernighan–Helsgaun (LKH) algorithm for the symmetric traveling salesperson problem (TSP). The LKH algorithm consists in an iterated local search based on the Lin–Kernighan heuristic (Lin and Kernighan, 1973). The algorithm and an effective implementation are described in Helsgaun (2000, 2009, 2018a). The source code is available in Helsgaun (2018b). We used LKH version 2.0.9.

The LKH scenario has 21 unconditional parameters. We set a time limit of 10 s as termination criterion, and a budget of 2000 total executions for irace. We used Euclidean TSP instances of sizes 1000, 1500, 2000, 2500, and 3000. We randomly generated 50 training instances (10 for each size) and 250 test instances (50 for each size), using the *portgen* instance generator from the 8th DIMACS Implementation Challenge (Johnson et al., 2001).

SCIP This scenario consists in the configuration of the Solving Constraint Integer Programs (SCIP), an open-source exact solver for mixed integer programming (Achterberg, 2009). We configure SCIP for solving the combinatorial auction winner determination problem (De Vries and Vohra, 2003). SCIP was previously configured in López-Ibáñez and Stützle (2014). We used the same version of SCIP (2.0.2) linked with the linear programming solver SoPlex 1.5.0. We also set the maximum memory to be used by SCIP to 350 MB.

We considered the same 207 unconditional categorical parameters used in López-Ibáñez and Stützle (2014). We set a time limit of 30 s for each execution of SCIP, and a budget of 2000 total executions for irace. We used the MIP-encoded instances introduced in Leyton-Brown et al. (2000). We randomly selected 50 training instances and 50 test instances with 200 goods and 1000 bids. Some configurations of SCIP produce infeasible solutions. We assign those configurations the worst possible cost value and do not use them to determine the performance envelopes.

Most of the experiments were performed using a single core of a computer with an 8-core AMD FX-8150 processor running at 3.6 GHz and 32 GB main memory, under Ubuntu Linux. The experiments of the SCIP scenario were performed using a single core of a computer with a 12-core AMD Ryzen 9 3900X processor running at 3.8 GHz and 32 GB main memory, under Ubuntu Linux.

5.2. Evaluation of capping methods

The first experiment consists in the evaluation of all capping methods. For each method, we executed irace 20 times and computed the total effort used for the configuration process. Then, we executed the first ranked configuration of each irace execution on the set of test instances with 5 replications, and computed the average cost deviation from the best known solutions.

The results are shown in Table 3. In the method description (column "Capping method"), the first letter indicates the envelope type (P for profile-based or A for area-based) of the capping method, followed by the strategy (E for elitist or D for adaptive). In the case of elitist methods, the next two letters represent the aggregation functions (B, W or M) used, respectively, for aggregating over multiple replications of a configuration (AR) and for aggregating over multiple configurations (AC). In the case of methods with a user-defined parameter (p or a_g), its value is given at the end of the description. For example, the method PEMW.1 represents the profile-based envelope, using the elitist strategy, model-based function M to aggregate replications, worst function W to aggregate configurations, and 0.1 for the parameter p required by the M function. Column "r. e.". presents the average relative effort

Table 3
Average relative effort and average solution cost deviation for each capping method. The three best values of each column are shown in bold.

Capping method	ACOTSP		HEACOL		TSBPP		HHBQP		LKH		SCIP	
	r. e.	r. d.	r. e.	r. d.	r. e.	r. d.	r. e.	a. d.	r. e.	r. d.	r. e.	r. d.
No capping	100.0	**0.33**	100.0	**4.14**	100.0	1.31	100.0	49.72	100.0	**0.04**	100.0	**0.04**
PEWW	40.3	0.37	38.7	4.22	87.4	1.25	55.7	65.16	37.8	**0.04**	69.0	0.05
PEBB	**22.3**	0.52	**25.2**	4.48	61.9	1.27	**25.1**	58.38	**24.3**	0.08	**21.7**	0.11
PEMW.1	74.5	**0.34**	77.9	**4.10**	95.1	**1.24**	82.6	72.44	60.2	**0.04**	89.9	**0.03**
PEMW.3	51.4	0.35	61.5	4.16	92.1	1.28	66.7	63.52	46.4	**0.04**	75.8	0.05
PEMW.5	30.2	0.44	**27.3**	4.48	**54.5**	1.35	40.0	72.16	35.4	0.05	54.4	0.07
PEMB.1	50.0	**0.32**	58.5	**4.11**	86.7	**1.22**	53.7	67.19	40.4	**0.04**	28.6	0.12
PEMB.3	**26.9**	0.46	31.5	4.23	74.6	1.30	37.2	55.75	**31.0**	0.05	**23.6**	0.12
PEMB.5	**21.8**	0.48	**23.5**	4.49	**44.6**	1.33	**25.8**	61.40	**24.0**	0.07	**21.6**	0.12
PD.2	65.0	0.38	62.4	4.28	92.6	**1.24**	78.4	**36.06**	66.2	**0.04**	65.3	0.15
PD.4	63.7	0.38	61.9	4.27	88.3	1.28	74.6	44.71	62.1	**0.04**	63.9	0.16
PD.6	55.7	0.40	56.5	4.30	80.7	1.29	65.7	48.87	53.6	0.05	58.6	0.16
PD.8	43.7	0.41	47.9	4.39	73.9	1.37	54.3	57.26	39.8	0.05	48.6	0.16
AEWW	73.2	0.35	72.8	4.18	87.6	1.28	82.7	46.97	52.5	**0.04**	94.0	**0.04**
AEBB	47.3	0.38	53.0	4.18	58.6	1.35	**34.1**	68.56	33.8	0.06	62.9	0.08
AD.2	82.8	0.35	84.6	4.16	85.6	1.30	85.2	**37.03**	78.7	**0.04**	83.4	**0.03**
AD.4	77.9	0.35	81.8	4.16	71.5	1.26	72.9	**37.48**	72.4	**0.04**	80.3	0.06
AD.6	69.7	0.35	74.0	**4.14**	58.5	1.34	58.2	37.71	59.5	**0.04**	71.1	0.08
AD.8	55.3	0.36	62.1	4.21	**52.6**	1.45	41.9	54.13	43.5	**0.04**	61.6	0.10

required when using each capping method in comparison to the effort required when configuring without capping. Column "r. d.". presents the average relative deviation from the best known solutions, obtained by executing the best found configurations on the test instances. In the case of HHBQP, we report the average absolute deviation (column "a. d.".) from the best known solutions, following the practice of the literature of UBQP (Palubeckis, 2006; Glover et al., 2010; Wang et al., 2012). We say that a capping method presents better quality than another if it presents a smaller deviation. We highlighted the three best values of relative effort and deviation for each configuration scenario.

We observe that all capping methods reduce some amount of effort. The reduction ranges from about 5% (method PEMW.1 on TSBPP) to about 78% (method PEMB.5 on ACOTSP and SCIP, and method PEBB on SCIP) of the effort required when configuring without capping. The resulting configurations present competitive quality in comparison to the one obtained without using capping. As expected, we can also observe that more aggressive methods (e.g. PEBB, PEMB.5 and PEMW.5) save more effort, but usually in exchange of producing worse configurations. This is the case when using best (B) instead of worst (W) aggregation functions, as well as using more aggressive (higher) values for the parameter p of the exponential model and the aggressiveness goal a_g.

We also measured the number of training instances used by each run of irace. For ACOTSP, HEACOL, TSBPP, LKH and SCIP, irace used an average of 35%, 71%, 61%, 37% and 49% of the available instances, respectively, with no more than 10% of variation over the different capping methods. Thus, irace never needed to perform more than one replication per training instance, and methods W and B have no effect when aggregating replications. The model-based method M has an effect even with a single replication, as explained in Section 4.1.1. For scenario HHBQP, irace does use all training instances and thus sometimes performs more than one replication per instance. In this case methods W and B for aggregating replications lead to different results.

Fig. 9 presents the trade-off between the average relative effort and the average solution cost deviation of each capping method. It also presents the Pareto frontier defined by the non-dominated methods, considering relative effort and deviation as two distinct objectives of the capping methods that must be minimized. We can see that most of the Pareto-optimal methods are profile-based and elitist, except in the HHBQP scenario, where profile- and area-based adaptive methods are the majority in the Pareto frontier.

We also observe that all capping methods present competitive results in the quality of the configurations found. The difference of the observed relative deviations from the best known solutions in comparison to those obtained without capping is always less than 1% (and less than 100 of absolute deviation for HHBQP, whose objective values are in the order of magnitude of 10^6). Even the most aggressive methods (PEBB, PEMB.5, PEMW.5) produce acceptable configurations, while saving more than half of the total configuration effort. We would expect no capping would produce the best results, however, in some cases the tuning process with capping found a better final configuration (e.g. capping method PEMW.1 in scenarios HEACOL, TSBPP, and SCIP). Since capped runs are penalized by returning the current cost, this can lead irace to discard the corresponding configuration earlier and focus the sampling of the parameter space on better configurations.

Table 4
Results of the multiple comparison Dunn's test, adjusted with the Bonferroni method.

Capping method	No capping		Capping method	No capping	
	ACOTSP	SCIP		ACOTSP	SCIP
PEWW	ns	ns	PD.4	ns	***
PEBB	***	ns	PD.6	**	***
PEMW.1	ns	ns	PD.8	***	**
PEMW.3	ns	ns	AEWW	ns	ns
PEMW.5	***	ns	AEBB	ns	ns
PEMB.1	ns	*	AD.2	ns	ns
PEMB.3	***	ns	AD.4	ns	ns
PEMB.5	***	*	AD.6	ns	ns
PD.2	ns	**	AD.8	ns	ns

Comparing the quality of the configurations produced using each capping method with those obtained using no capping. Symbol "ns" means no significant difference, while the asterisks denote the order of the p-values: (*) $p \leq 0.01$, (**) $p \leq 0.001$ and (***) $p \leq 0.0001$.

We have performed a non-parametric Kruskal–Wallis test to check whether any capping method statistically dominates another in terms of the quality of the produced configurations (Montgomery, 2012). The results indicate statistically significant differences in the ACOTSP, HEACOL, LKH and SCIP scenarios (p-values of 1.01×10^{-23}, 1.82×10^{-7}, 1.93×10^{-3} and 2.97×10^{-22}, respectively), while the null hypothesis could not be rejected in the TSBPP and HHBQP scenarios (p-values of 0.99 and 0.09, respectively). We have also performed a post-hoc analysis using the Dunn's multiple comparisons test (Dunn, 1964) to assess the pairwise differences of the capping methods. We used a significance level of 0.01, and the Bonferroni correction method (Dunn, 1961) to control the familywise error rate. The HEACOL and LKH scenarios presented statistical differences among distinct capping methods, but

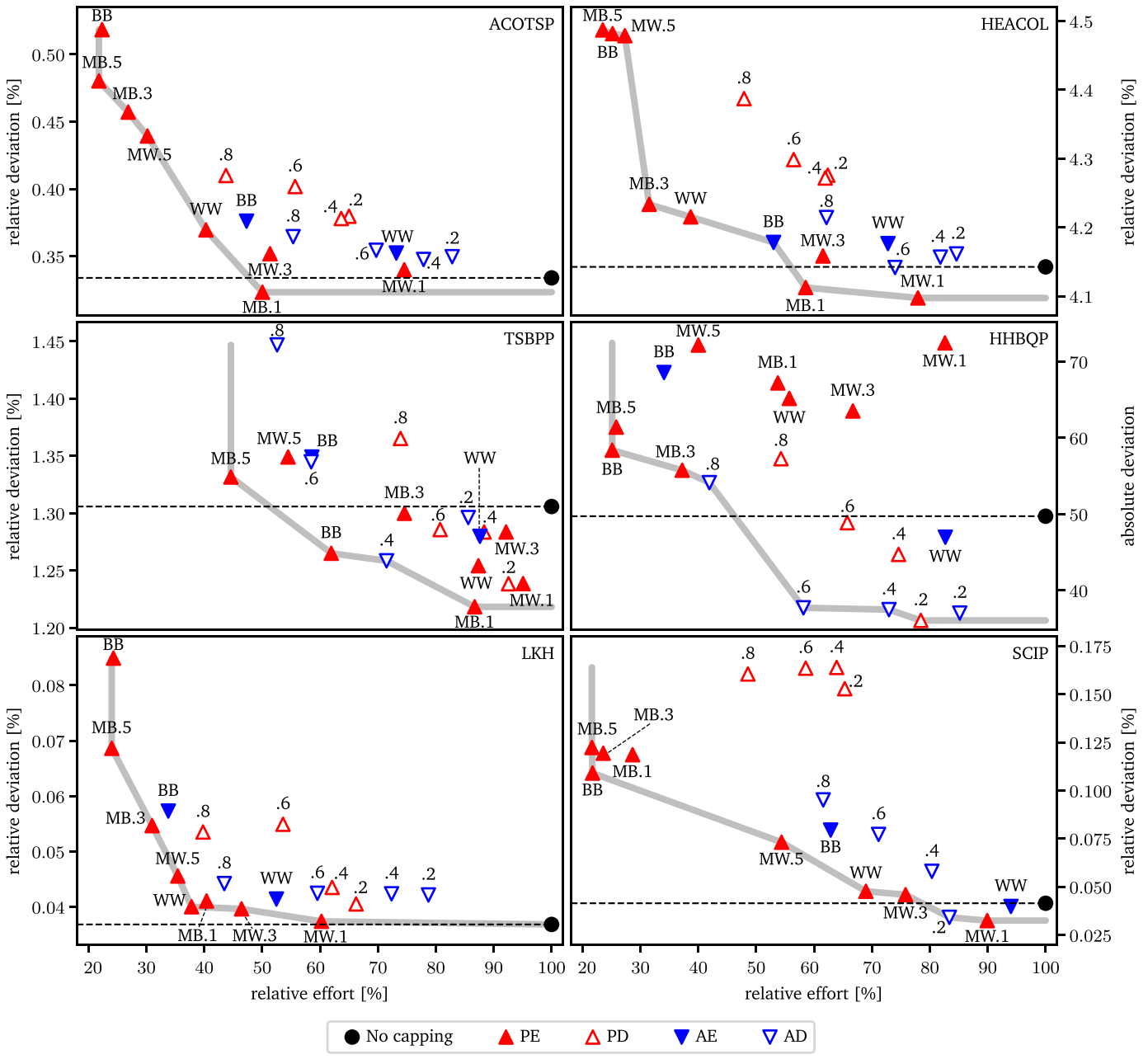


Fig. 9. Normalized configuration effort and quality of the resulting configurations for all capping methods.

no difference was identified between any capping method and configuring with no capping. For ACOTSP and SCIP, however, some of the most aggressive methods presented statistically worse configurations than those obtained by configuring with no capping. Table 4 presents these results, and their statistical significance. We can see that the differences are statistically more significant for ACOTSP. For the area-based methods, even the more aggressive approaches (e.g. AEBB and AD.8) produce solutions which are statistically not worse than those found without capping.

5.3. Recommended methods

Based on the results discussed above, we can identify specific recommendations presented in Table 5. First, we selected two conservative, robust capping methods (PEMW.1 and AD.4, which have low p and a_g values, respectively, thus making them more conservative) that maintain the quality of the final configurations, but still save a reasonable effort. Then, we selected two moderately aggressive methods (PEMB.1, i.e., PEMB with the least aggressive p value, and PEWW) that save more effort, but sometimes find worse configurations than tuning with no capping. Table 5 shows the average relative effort of those methods for all six configuration scenarios, as well as their quality loss on each configuration scenario. The quality loss is the difference between the average cost deviation obtained using the capping method and using no capping. We observe that all recommended methods reduce the tuning effort by at least 20%, but still produce solutions of acceptable quality. In some cases, the use of capping produced better configurations than tuning with no capping. We also note that the selected methods cover the different capping components, i.e. profile-based elitist and area-based adaptive methods. In this section, we analyze in detail the behavior of the recommended capping methods. We used the visualization tool acviz (De Souza et al., 2021b).

Table 5
Recommended capping methods.

Category	Capping method	Relative effort [%]	Quality loss					
			ACOTSP	HEACOL	TSBPP	HHBQP	LKH	SCIP
Conservative	PEMW.1	80.0	0.01	−0.04	−0.07	22.72	0.00	−0.01
	AD.4	76.1	0.02	0.02	−0.05	−12.24	0.00	0.02
Aggressive	PEMB.1	53.0	−0.01	−0.03	−0.09	17.47	0.00	0.08
	PEWW	54.8	0.04	0.08	−0.06	15.44	0.00	0.01

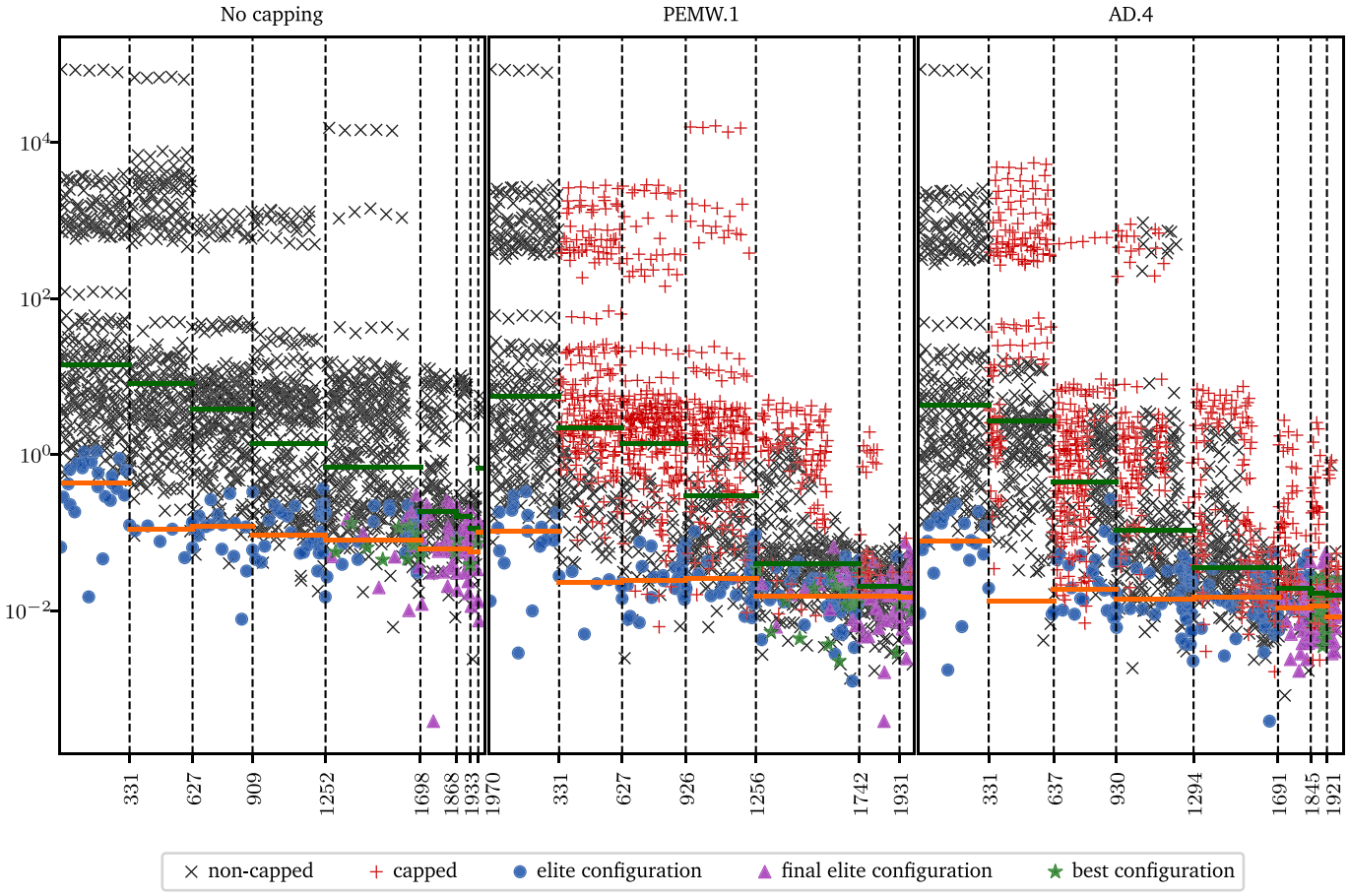


Fig. 10. Evolution of the automatic configuration of ACOTSP using the conservative capping methods.

Figs. 10 and 11 present the evolution of the configuration process of ACOTSP with no capping, when using the conservative methods PEMW.1 and AD.4 (Fig. 10) or the aggressive methods PEWW and PEMB.1 (Fig. 11). We selected one of the irace executions at random to produce this figure (the other executions present a similar behavior). We plot the quality over the executions. Since we used a budget of 2000 total executions in the ACOTSP scenario, the x axis ranges from 1 to 2000. The quality is the relative deviation (on a logarithmic scale) from the best known solutions. A vertical dashed line marks the beginning of each irace iteration with the respective budget used so far. We also indicate if the execution was capped (+) or not (×), and the execution of elite configurations. A blue circle represent the execution of a configuration selected as elite in the corresponding iteration. A purple triangle represents the execution of a configuration which became elite in the last iteration, i.e. a final elite configuration. Executions of the best final configuration are represented by a green star. This is the configuration used to evaluate the quality of the irace run. In the case of capped executions (+ marker), we executed them again until the cut-off effort, i.e. without capping it, to obtain the quality of the complete execution. Therefore, the quality values of capped executions are those which would be obtained if they were not capped. Finally, the horizontal lines in each iteration represent the median relative deviations of all executions (green) and of the executions of elite configurations (orange) obtained in that iteration.

The behavior of how the observed execution quality changes over iterations is very similar for the different scenarios. We can see that the capping methods are effective in identifying poor performers and then stop executions that will not find the best solutions. Almost all executions capped by the analyzed methods turned out to be bad performers when executed until completion, as we observe in the final quality of capped executions in Figs. 10 and 11. We can also analyze the aggressiveness of the recommended methods by looking at the amount of capped executions, which is clearly bigger in methods PEWW and PEMB.1. Besides that, we observe that the separation between capped and non-capped executions is higher in the conservative methods, indicating more tolerance in allowing configurations to run until completion. On the other hand, in some iterations the aggressive methods turn out to cap almost all executions of non-elite configurations. Finally, for users seeking a conservative method, we recommend to use PEMW.1, for those seeking an aggressive method, we recommend to use PEMB.1.

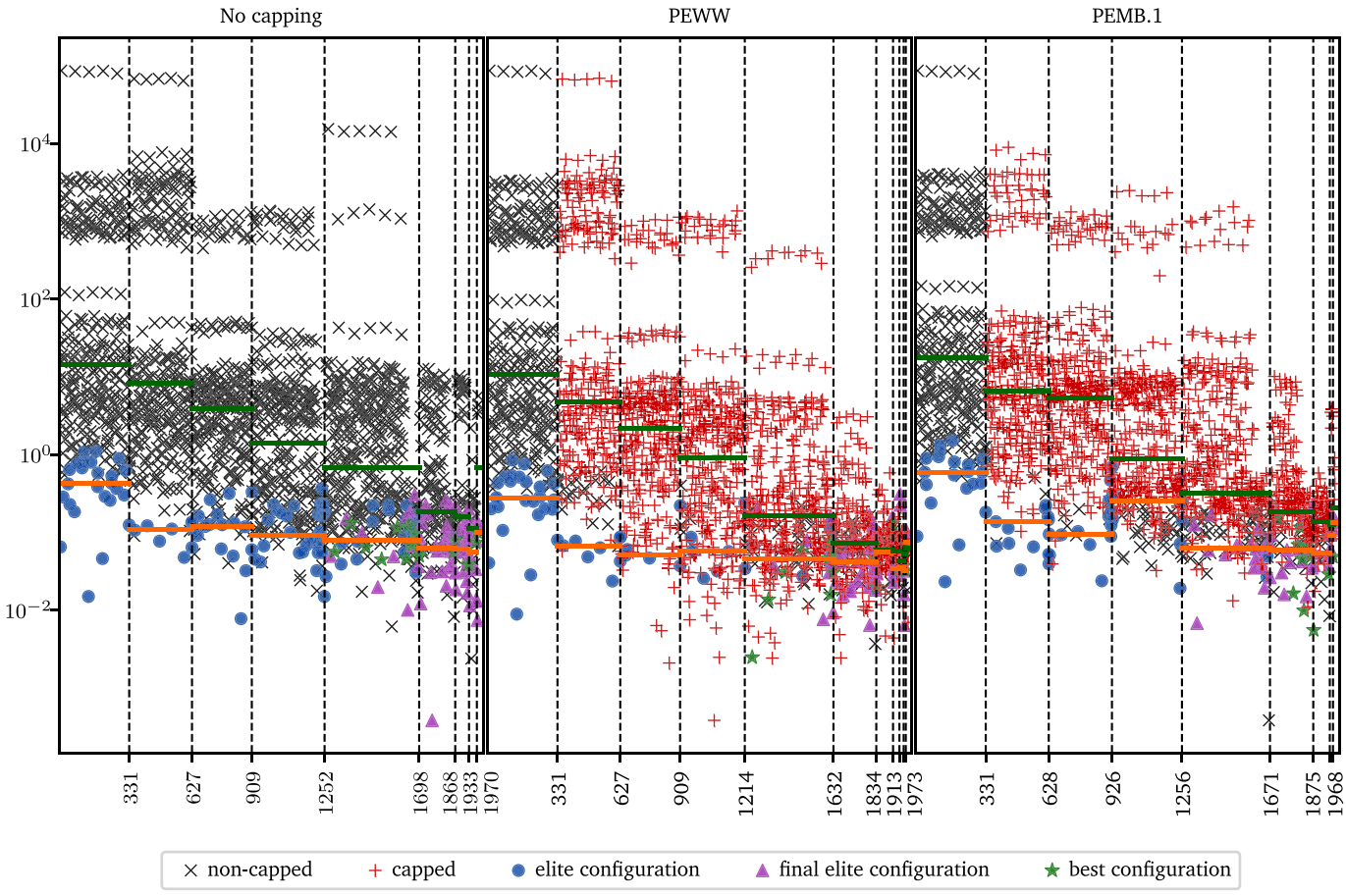


Fig. 11. Evolution of the automatic configuration of ACOTSP using the aggressive capping methods.

5.4. Time as budget

A common scenario is using a time limit for the budget of the configuration process. We designed an experiment to evaluate the benefits of using capping in these conditions. We defined a tight configuration time limit of 21000 s for ACOTSP, 3200 s for HEACOL, 700 s for TSBPP, 7000 s for HHBQP, 3500 s for LKH, and 21000 s for SCIP. This means around 350 non-capped executions for ACOTSP, HEACOL, HHBQP, LKH and SCIP, which had a budget of 2000 executions in previous experiments, and around 100 non-capped executions for TSBPP, which had a budget of 500 executions. These budget values make the configuration a challenging task.

When using the total execution time as budget, irace first estimates the time required by a single execution by evaluating a few random configurations. Based on this estimated time, it plans the iterations to be performed and the number of executions in each of them. Every time an iteration finishes, the time required by each previous execution is used to update the time estimate and re-plan the next iterations. When using capping, the time saved by early stopping poorly performing executions becomes available to evaluate more configurations in the next iterations. The redistribution of the saved time is performed by irace considering the average time used so far in each execution. Thus, irace implicitly uses the amount of capping done to plan the next iterations. Thus, when using capping, we expect that the time saved is used to further explore the configuration space.

We evaluated all capping methods in the described scenario with 5 independent runs of irace. Table 6 shows the percentage of increase in the number of executions, generated configurations and evaluated instances, in comparison to configuring with no capping. We observe that the capping methods can help irace to make better use of the available budget, since poor performers are discarded early and the saved time can be used to further explore the configuration space. When using capping, irace samples more configurations and performs more executions during the tuning process. Besides that, it uses more instances to evaluate the quality of the configurations. For example, the increase in the number of configurations ranges from around 30% (less aggressive method AD.2) to around 2250% (more aggressive method PEMB.5). The corresponding increase in the number of total executions exceeds 1500% in the most aggressive methods.

Table 7 presents the mean relative deviation (and the mean absolute deviation for HHBQP) from the best known solutions when configuring with each capping method and using the budget as a total configuration time. These values were determined by running 5 replications of the best configuration found in each irace run on the set of test instances. We highlight in bold the deviation values less than the one obtained by configuring with no capping. For most of the scenarios, the configurations found by using capping performed better than those obtained without capping. In HEACOL, TSBPP, HHBQP and LKH, the use of capping allowed irace to find configurations competitive to those obtained in the experiment with executions as budget (Table 3 and Fig. 9), but using less computational effort.

Given the above results, we recommend using AEBB for scenarios where the configuration budget is defined relative to the time of the target algorithm. Although the percentage increase in configurations and executions achieved by AEBB is more modest than for other methods (Table 6), these additional executions lead to consistent improvements in quality for all scenarios evaluated here (Table 7). Thus, the AEBB capping method improves the quality of the automatic tuning of optimization algorithms in this type of scenario.

Table 6
Percentage of increase in the number of executions, configurations generated, and instances evaluated when running irace with total execution time as budget and using each capping method, in comparison with using no capping.

Capping method	Exec. [%]	Config. [%]	Inst. [%]
PEWW	193.2	203.2	19.0
PEBB	1598.2	2023.5	63.8
PEMW.1	49.2	50.7	1.0
PEMW.3	106.2	102.5	17.6
PEMW.5	264.7	298.6	29.1
PEMB.1	258.6	305.4	25.4
PEMB.3	627.4	753.2	45.3
PEMB.5	1792.1	2256.6	66.7
PD.2	61.1	56.4	12.8
PD.4	78.8	71.9	18.2
PD.6	108.7	95.3	22.0
PD.8	202.6	193.1	25.8
AEWW	55.1	55.1	4.6
AEBB	222.5	252.4	28.6
AD.2	33.9	31.1	7.6
AD.4	51.8	48.0	8.5
AD.6	92.4	85.2	24.5
AD.8	171.1	177.6	20.4

Table 7
Average deviation of the resulting configurations with total execution time as budget. The values better than configuring with no capping are shown in bold.

Capping method	ACOTSP	HEACOL	TSBPP	HHBQP	LKH	SCIP
No capping	0.46	4.48	1.31	206.91	0.11	0.12
PEWW	0.51	**4.12**	1.31	**81.37**	**0.05**	**0.09**
PEBB	0.60	**4.24**	**1.24**	**117.81**	**0.06**	**0.11**
PEMW.1	**0.41**	**4.29**	1.31	**98.58**	**0.05**	**0.09**
PEMW.3	**0.41**	**4.20**	1.31	**78.77**	**0.06**	**0.05**
PEMW.5	0.56	**4.18**	1.31	**89.93**	**0.06**	**0.09**
PEMB.1	**0.40**	**4.21**	1.31	**80.94**	**0.07**	0.17
PEMB.3	0.54	**4.18**	**1.24**	**93.26**	**0.07**	0.13
PEMB.5	0.53	**4.40**	1.31	**62.43**	**0.10**	0.12
PD.2	**0.43**	**4.18**	1.38	**72.82**	**0.10**	0.23
PD.4	**0.41**	**4.26**	1.40	**165.60**	**0.05**	0.24
PD.6	**0.42**	**4.25**	1.31	**35.62**	**0.09**	0.24
PD.8	0.59	**4.30**	1.40	**39.64**	**0.07**	0.20
AEWW	**0.40**	**4.17**	**1.24**	**60.30**	**0.08**	0.13
AEBB	**0.40**	**4.20**	**1.24**	**33.21**	**0.05**	**0.10**
AD.2	**0.39**	**4.09**	**1.24**	**84.22**	**0.06**	0.13
AD.4	**0.38**	**4.28**	**1.24**	**65.77**	**0.10**	0.17
AD.6	**0.44**	**4.15**	**1.24**	**67.05**	**0.08**	0.18
AD.8	**0.42**	**4.34**	**1.24**	**83.33**	**0.06**	0.23

6. Concluding remarks

We have proposed capping methods that speed up the automatic configuration of optimization algorithms. Previous methods (Hutter et al., 2009; Pérez Cáceres et al., 2017b) were designed for the configuration of decision algorithms and are not suitable for optimization scenarios. The methods described in this article use the previous executions to compute a performance envelope, which is used to evaluate new executions and cap those with unsatisfactory performance. The experimental results show the effectiveness of the capping methods to reduce the computational effort of the automatic algorithm configuration, while keeping the quality of the resulting configurations.

We identified two conservative (PEMW.1 and AD.4) and two aggressive (PEMB.1 and PEWW) methods, which have been shown to be robust and present good trade-offs between the saved effort and the quality of the final configurations. Their average effort savings ranges from 20% to 45% of the configuration time with no capping, and the resulting configurations are comparable in terms of quality.

We also evaluated the proposed capping methods with the total execution time as configuration budget. In this case, the capping methods helped to discard poorly performing configurations and allow irace to use the saved time to better explore promising regions of the parameter space. We recommend AEBB for this type of scenario as it improves the results over no capping in all benchmarks. These results indicate that capping can also be helpful to scale the automatic configuration techniques to challenging scenarios (Mascia et al., 2013; Styles and Hoos, 2013).

Although the optimization scenarios are the primary focus of the proposed capping methods, they can be used for the automatic configuration of algorithms in other domains. If we can measure the performance profile of an execution, we can apply the proposed approaches to the configuration process. For example, if we can monitor how close a decision algorithm is to its termination, we can use previous executions to compute a performance envelope and use it to evaluate new executions. We can combine the approaches proposed here with existing capping methods for decision algorithms and discard configurations before exhausting the cut-off time, if the observed performance profile is unsatisfactory. We also want to explore the use of the capping methods in searching good parameter settings for computational simulations. In such scenarios, the evaluation of configurations are costly and the use of capping can reduce the required computational effort for the automatic tuning. Finally, future work should also explore how the choice of capping method and its parameters relates to scenario features.

CRediT authorship contribution statement

Marcelo de Souza: Conceptualization, Methodology, Software, Investigation, Writing – original draft. **Marcus Ritt:** Conceptualization, Methodology, Validation, Investigation, Writing – review & editing, Supervision. **Manuel López-Ibáñez:** Conceptualization, Methodology, Validation, Investigation, Writing – review & editing.

Acknowledgments

This research has been supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. M. de Souza acknowledges the support of the Santa Catarina State University, Brasil. M. Ritt acknowledges the support of CNPq, Brasil (grant 437859/2018-5) and Google Research Latin America (grant 25111). M. López-Ibáñez is a "Beatriz Galindo" Senior Distinguished Researcher (BEAGAL 18/00053) funded by the Spanish Ministry of Science and Innovation (MICINN). This research is partially funded by TAILOR ICT-48 Network (No 952215) funded by EU Horizon 2020 research and innovation programme. Funding for open access charge: Universidad de Málaga / CBUA.

References

Achterberg, T., 2009. SCIP: Solving constraint integer programs. Math. Program. Comput. 1 (1), 1–41.

Ansótegui, C., Malitsky, Y., Samulowitz, H., Sellmann, M., Tierney, K., 2015. Model-based genetic algorithms for algorithm configuration. In: Yang, Q., Wooldridge, M. (Eds.), Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-15. IJCAI/AAAI Press, Menlo Park, CA, pp. 733–739. http://dx.doi.org/10.5555/2832249.2832351.

Ansótegui, C., Sellmann, M., Tierney, K., 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent, I.P. (Ed.), Principles and Practice of Constraint Programming, CP 2009. In: Lecture Notes in Computer Science, vol. 5732, Springer, Heidelberg, Germany, pp. 142–157. http://dx.doi.org/10.1007/978-3-642-04244-7_14.

Balaprakash, P., Birattari, M., Stützle, T., 2007. Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In: Bartz-Beielstein, T., Blesa, M.J., Blum, C., Naujoks, B., Roli, A., Rudolph, G., Sampels, M. (Eds.), Hybrid Metaheuristics. In: Lecture Notes in Computer Science, vol. 4771, Springer, Heidelberg, Germany, pp. 108–122. http://dx.doi.org/10.1007/978-3-540-75514-2_9.

Beasley, J.E., 1998. Heuristic Algorithms for the Unconstrained Binary Quadratic Programming Problem. Technical Report, The Management School, Imperial College, London, England.

Berkey, J.O., Wang, P.Y., 1987. Two-dimensional finite bin-packing algorithms. J. Oper. Res. Soc. 38 (5), 423–429. http://dx.doi.org/10.2307/2582731.

Birattari, M., 2003. The race Package for R: Racing Methods for the Selection of the Best. Technical Report TR/IRIDIA/2003-037, IRIDIA, Université Libre de Bruxelles, Belgium.

Birattari, M., 2009. Tuning Metaheuristics: A Machine Learning Perspective. In: Studies in Computational Intelligence, vol. 197, Springer, Berlin, Heidelberg, http://dx.doi.org/10.1007/978-3-642-00483-4.

Blum, C., Calvo, B., Blesa, M.J., 2015. FrogCOL and FrogMIS: New decentralized algorithms for finding large independent sets in graphs. Swarm Intell. 9 (2–3), 205–227. http://dx.doi.org/10.1007/s11721-015-0110-1.

Branke, J., Elomari, J., 2011. Simultaneous tuning of metaheuristic parameters for various computing budgets. In: Krasnogor, N., Lanzi, P.L. (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2011. ACM Press, New York, NY, pp. 263–264. http://dx.doi.org/10.1145/2001858.2002006.

De Souza, M., Ritt, M., 2018a. Automatic grammar-based design of heuristic algorithms for unconstrained binary quadratic programming. In: Evolutionary Computation in Combinatorial Optimization. Springer International Publishing, pp. 67–84. http://dx.doi.org/10.1007/978-3-319-77449-7_5.

De Souza, M., Ritt, M., 2018b. Hybrid heuristic for unconstrained binary quadratic programming – source code of HHBQP. https://github.com/souzamarcelo/hhbqp.

De Souza, M., Ritt, M., López-Ibáñez, M., 2020. CAPOPT: Capping methods for the automatic configuration of optimization algorithms. URL https://github.com/souzamarcelo/capopt.

De Souza, M., Ritt, M., López-Ibáñez, M., 2021a. Capping methods for the automatic configuration of optimization algorithms – supplementary material. https://github.com/souzamarcelo/supp-cor-capopt.

De Souza, M., Ritt, M., López-Ibáñez, M., Pérez Cáceres, L., 2021b. ACVIZ: A tool for the visual analysis of the configuration of algorithms with irace. 8, pp. 1–9. http://dx.doi.org/10.1016/j.orp.2021.100186,

De Vries, S., Vohra, R.V., 2003. Combinatorial auctions: A survey. INFORMS J. Comput. 15 (3), 284–309.

Delorme, M., Iori, M., Martello, S., 2016. Bin packing and cutting stock problems: Mathematical models and exact algorithms. European J. Oper. Res. 255 (1), 1–20. http://dx.doi.org/10.1016/j.ejor.2016.04.030.

Delorme, M., Iori, M., Martello, S., 2018. BPPLIB: A library for bin packing and cutting stock problems. Optim. Lett. 12 (2), 235–250. http://dx.doi.org/10.1007/s11590-017-1192-z.

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

Dunn, O.J., 1961. Multiple comparisons among means. J. Amer. Statist. Assoc. 56 (293), 52–64.

Dunn, O.J., 1964. Multiple comparisons using rank sums. Technometrics 6 (3), 241–252.

Franzin, A., Stützle, T., 2019. Revisiting simulated annealing: A component-based analysis. Comput. Oper. Res. 104, 191–206. http://dx.doi.org/10.1016/j.cor.2018.12.015.

Galinier, P., Hao, J.-K., 1999. Hybrid evolutionary algorithms for graph coloring. J. Comb. Optim. 3 (4), 379–397. http://dx.doi.org/10.1023/A:1009823419804.

Glover, F., Lü, Z., Hao, J.-K., 2010. Diversification-driven tabu search for unconstrained binary quadratic problems. 4OR 8 (3), 239–253. http://dx.doi.org/10.1007/s10288-009-0115-y.

Helsgaun, K., 2000. An effective implementation of the Lin-Kernighan traveling salesman heuristic. European J. Oper. Res. 126, 106–130.

Helsgaun, K., 2009. General *k*-opt submoves for the Lin-Kernighan TSP heuristic. Math. Program. Comput. 1 (2–3), 119–163.

Helsgaun, K., 2018a. Efficient recombination in the Lin-Kernighan-Helsgaun traveling salesman heuristic. In: Auger, A., Fonseca, C.M., Lourenço, N., Machado, P., Paquete, L., Whitley, D. (Eds.), Parallel Problem Solving from Nature - PPSN XV. In: Lecture Notes in Computer Science, vol. 11101, Springer, Cham, pp. 95–107. http://dx.doi.org/10.1007/978-3-319-99253-2_8.

Helsgaun, K., 2018b. Source code of the Lin-Kernighan-Helsgaun traveling salesman heuristic. http://webhotel4.ruc.dk/~keld/research/LKH/.

Hoos, H.H., Stützle, T., 2005. Stochastic Local Search—Foundations and Applications. Morgan Kaufmann Publishers, San Francisco, CA.

Hutter, F., Hoos, H.H., Leyton-Brown, K., 2010. Automated configuration of mixed integer programming solvers. In: Lodi, A., Milano, M., Toth, P. (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010. In: Lecture Notes in Computer Science, vol. 6140, Springer, Heidelberg, Germany, pp. 186–202. http://dx.doi.org/10.1007/978-3-642-13520-0_23.

Hutter, F., Hoos, H.H., Leyton-Brown, K., 2011. Sequential model-based optimization for general algorithm configuration. In: Coello Coello, C.A. (Ed.), Learning and Intelligent Optimization, 5th International Conference, LION 5. In: Lecture Notes in Computer Science, vol. 6683, Springer, Heidelberg, Germany, pp. 507–523. http://dx.doi.org/10.1007/978-3-642-25566-3_40.

Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T., 2009. ParamILS: An automatic algorithm configuration framework. J. Artificial Intelligence Res. 36, 267–306. http://dx.doi.org/10.1613/jair.2861.

Hutter, F., López-Ibáñez, M., Fawcett, C., Lindauer, M.T., Hoos, H.H., Leyton-Brown, K., Stützle, T., 2014. AClib: A benchmark library for algorithm configuration. In: Pardalos, P.M., Resende, M.G.C., Vogiatzis, C., Walteros, J.L. (Eds.), Learning and Intelligent Optimization, 8th International Conference, LION 8. In: Lecture Notes in Computer Science, vol. 8426, Springer, Heidelberg, Germany, pp. 36–40. http://dx.doi.org/10.1007/978-3-319-09584-4_4.

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

Karapetyan, D., Parkes, A.J., Stützle, T., 2018. Algorithm configuration: Learning policies for the quick termination of poor performers. In: Battiti, R., Brunato, M., Kotsireas, I., Pardalos, P.M. (Eds.), Learning and Intelligent Optimization, 12th International Conference, LION 12. In: Lecture Notes in Computer Science, vol. 11353, Springer, Cham, Switzerland, pp. 220–224. http://dx.doi.org/10.1007/978-3-030-05348-2_20.

KhudaBukhsh, A.R., Xu, L., Hoos, H.H., Leyton-Brown, K., 2016. SATenstein: Automatically building local search SAT Solvers from Components. Artificial Intelligence 232, 20–42. http://dx.doi.org/10.1016/j.artint.2015.11.002.

Kochenberger, G.A., Hao, J.-K., Glover, F., Lewis, M., Lü, Z., Wang, H., Wang, Y., 2014. The unconstrained binary quadratic programming problem: A survey. J. Comb. Optim. 28 (1), 58–81. http://dx.doi.org/10.1007/s10878-014-9734-0.

Lewis, R.M.R., 2016a. A Guide to Graph Colouring: Algorithms and Applications. Springer, Cham, http://dx.doi.org/10.1007/978-3-319-25730-3.

Lewis, R.M.R., 2016b. Suite of graph colouring algorithms – supplementary material to the book "A guide to graph colouring: Algorithms and applications". http://rhydlewis.eu/resources/gCol.zip.

Leyton-Brown, K., Pearson, M., Shoham, Y., 2000. Towards a universal test suite for combinatorial auction algorithms. In: Jhingran, A., et al. (Eds.), ACM Conference on Electronic Commerce (EC-00). ACM Press, New York, NY, pp. 66–76. http://dx.doi.org/10.1145/352871.352879.

Lin, S., Kernighan, B.W., 1973. An effective heuristic algorithm for the traveling salesman problem. Oper. Res. 21 (2), 498–516.

Lodi, A., Martello, S., Vigo, D., 1999. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. INFORMS J. Comput. 11 (4), 345–357. http://dx.doi.org/10.1287/ijoc.11.4.345.

Lodi, A., Martello, S., Vigo, D., 2004a. TSpack: A unified tabu search code for multi-dimensional bin packing problems. Ann. Oper. Res. 131 (1–4), 203–213. http://dx.doi.org/10.1023/B:ANOR.0000039519.03572.08.

Lodi, A., Martello, S., Vigo, D., 2004b. Two- and three-dimensional bin packing – source code of TSpack. http://or.dei.unibo.it/research_pages/ORcodes/TSpack.html.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., Birattari, M., 2016a. The irace package: Iterated racing for automatic algorithm configuration. Oper. Res. Perspect. 3, 43–58. http://dx.doi.org/10.1016/j.orp.2016.09.002.

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

López-Ibáñez, M., Stützle, T., 2012. The automatic design of multi-objective ant colony optimization algorithms. IEEE Trans. Evol. Comput. 16 (6), 861–875. http://dx.doi.org/10.1109/TEVC.2011.2182651.

López-Ibáñez, M., Stützle, T., 2014. Automatically improving the anytime behaviour of optimisation algorithms. European J. Oper. Res. 235 (3), 569–582. http://dx.doi.org/10.1016/j.ejor.2013.10.043.

López-Ibáñez, M., Stützle, T., Dorigo, M., 2018. Ant colony optimization: A component-wise overview. In: Martí, R., Pardalos, P.M., Resende, M.G.C. (Eds.), Handbook of Heuristics. Springer International Publishing, pp. 371–407. http://dx.doi.org/10.1007/978-3-319-07124-4_21.

Martello, S., Vigo, D., 1998. Exact solution of the two-dimensional finite bin packing problem. Manage. Sci. 44 (3), 388–399. http://dx.doi.org/10.1287/mnsc.44.3.388.

Mascia, F., Birattari, M., Stützle, T., 2013. Tuning algorithms for tackling large instances: An experimental protocol. In: Pardalos, P.M., Nicosia, G. (Eds.), Learning and Intelligent Optimization, 7th International Conference, LION 7. In: Lecture Notes in Computer Science, vol. 7997, Springer, Heidelberg, Germany, pp. 410–422. http://dx.doi.org/10.1007/978-3-642-44973-4_44.

Montgomery, D.C., 2012. Design and Analysis of Experiments, eighth ed. John Wiley & Sons, New York, NY.

Pagnozzi, F., Stützle, T., 2019. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. European J. Oper. Res. 276, 409–421. http://dx.doi.org/10.1016/j.ejor.2019.01.018.

Palubeckis, G., 2006. Iterated tabu search for the unconstrained binary quadratic optimization problem. Informatica 17 (2), 279–296. http://dx.doi.org/10.15388/Informatica.2006.138.

Pérez Cáceres, L., López-Ibáñez, M., Hoos, H.H., Stützle, T., 2017a. An experimental study of adaptive capping in irace. In: Battiti, R., Kvasov, D.E., Sergeyev, Y.D. (Eds.), Learning and Intelligent Optimization, 11th International Conference, LION 11. In: Lecture Notes in Computer Science, vol. 10556, Springer, Cham, Switzerland, pp. 235–250. http://dx.doi.org/10.1007/978-3-319-69404-7_17.

Pérez Cáceres, L., López-Ibáñez, M., Hoos, H.H., Stützle, T., 2017b. An experimental study of adaptive capping in irace: Supplementary material. http://iridia.ulb.ac.be/supp/IridiaSupp2016-007/.

Pérez Cáceres, L., López-Ibáñez, M., Stützle, T., 2014. An analysis of parameters of irace. In: Blum, C., Ochoa, G. (Eds.), Proceedings of EvoCOP 2014 – 14th European Conference on Evolutionary Computation in Combinatorial Optimization. In: Lecture Notes in Computer Science, vol. 8600, Springer, Heidelberg, Germany, pp. 37–48. http://dx.doi.org/10.1007/978-3-662-44320-0_4.

Pérez Cáceres, L., López-Ibáñez, M., Stützle, T., 2015. Ant colony optimization on a limited budget of evaluations. Swarm Intell. 9 (2–3), 103–124. http://dx.doi.org/10.1007/s11721-015-0106-x.

Pérez Cáceres, L., Pagnozzi, F., Franzin, A., Stützle, T., 2017c. Automatic configuration of GCC using irace. In: Lutton, E., Legrand, P., Parrend, P., M., Nicolas, Schoenauer, M. (Eds.), EA 2017: Artificial Evolution. In: Lecture Notes in Computer Science, vol. 10764, Springer, Heidelberg, Germany, pp. 202–216. http://dx.doi.org/10.1007/978-3-319-78133-4_15.

Stützle, T., 2002. ACOTSP: A software package of various ant colony optimization algorithms applied to the symmetric traveling salesman problem. URL http://www.aco-metaheuristic.org/aco-code.

Stützle, T., López-Ibáñez, M., Pellegrini, P., Maur, M., Montes de Oca, M.A., Birattari, M., Dorigo, M., 2012. Parameter adaptation in ant colony optimization. In: Hamadi, Y., Monfroy, E., Saubion, F. (Eds.), Autonomous Search. Springer, Berlin, Germany, pp. 191–215. http://dx.doi.org/10.1007/978-3-642-21434-9_8.

Styles, J., Hoos, H.H., 2013. Ordered racing protocols for automatically configuring algorithms for scaling performance. In: Blum, C., Alba, E. (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2013. ACM Press, New York, NY, pp. 551–558. http://dx.doi.org/10.1145/2463372.2463438.

Trick, M.A., 2018. Graph coloring instances. https://mat.gsia.cmu.edu/COLOR/instances.html.

Wang, Y., Lü, Z., Glover, F., Hao, J.-K., 2012. Path relinking for unconstrained binary quadratic programming. European J. Oper. Res. 223 (3), 595–604. http://dx.doi.org/10.1016/j.ejor.2012.07.012.

Wiegele, A., 2007a. Biq Mac Library – a Collection of Max-Cut and Quadratic 0-1 Programming Instances of Medium Size. Technical Report, Institut für Mathematik, Alpen-Adria-Universität Klagenfurt, URL http://biqmac.aau.at/biqmaclib.pdf.

Wiegele, A., 2007b. Biq mac library – binary quadratic and max cut library. http://biqmac.aau.at/biqmaclib.html.

Zilberstein, S., 1996. Using anytime algorithms in intelligent systems. AI Mag. 17 (3), 73–83. http://dx.doi.org/10.1609/aimag.v17i3.1232, arXiv:http://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1232.