An experimental study of adaptive capping in irace

Leslie Pérez Cáceres*[1], Manuel López-Ibáñez[2], Holger Hoos[3], and Thomas Stützle[1]

[1] IRIDIA, Université Libre de Bruxelles, Brussels, Belgium
{leslie.perez.caceres,stuetzle}@ulb.ac.be
[2] Alliance Manchester Business School, University of Manchester, UK
manuel.lopez-ibanez@manchester.ac.uk
[3] Computer Science Department, University of British Columbia, Vancouver, Canada
hoos@cs.ubc.cs

Abstract. The irace package is a widely used for automatic algorithm configuration and implements various iterated racing procedures. The original irace was designed for the optimisation of the solution quality reached within a given running time, a situation frequently arising when configuring algorithms such as stochastic local search procedures. However, when applied to configuration scenarios that involve minimising the running time of a given target algorithm, irace falls short of reaching the performance of other general-purpose configuration approaches, since it tends to spend too much time evaluating poor configurations. In this article, we improve the efficacy of irace in running time minimisation by integrating an adaptive capping mechanism into irace, inspired by the one used by ParamILS. We demonstrate that the resulting irace$_{\mathsf{cap}}$ reaches performance levels competitive with those of state-of-the-art algorithm configurators that have been designed to perform well on running time minimisation scenarios. We also investigate the behaviour of irace$_{\mathsf{cap}}$ in detail and contrast different ways of integrating adaptive capping.

1 Introduction

Algorithm configuration is the task of finding parameter settings (a configuration) of a target algorithm that achieve high performance for a given class of problem instances [6,8]. The appropriate choice of parameter settings is often crucial for obtaining good performance, particularly when dealing with computationally challenging (e.g., $\mathcal{NP}$-hard) problems. This choice usually depends on the set or distribution of problem instances to be solved as well as on the execution environment. Therefore, using appropriately chosen parameter values is not only essential for reaching peak performance, but also for conducting fair performance comparisons between different algorithms for the same problem.

Traditionally, algorithm configuration has been performed manually, relying on experience and intuition about the behaviour of a given algorithm. However, typical manual configuration processes are time-consuming and tedious; furthermore, they often leave the performance potential of a given target algorithm unrealised. In light of this, several automated algorithm configuration

* Corresponding author.

approaches have been developed and are now used increasingly widely. Prominent examples of general-purpose algorithm configuration procedures include ParamILS [13], SMAC [12], GGA++ [1] and irace [4,18,17]. The key idea behind these and other configuration procedures is to view algorithm configuration as a stochastic optimisation problem that can be solved by effectively searching the space of configurations of a given target algorithm A. The performance metrics most commonly optimised in this context are the solution quality reached by A within a certain time budget and the running time of A for finding a solution (of a certain quality) to a given problem instance.

The irace software is an automatic configurator based on the iterated F-race procedure [4,7] and recent improvements [18,17]. It was initially developed for configuring metaheuristic algorithms that optimise solution quality. In contrast, minimisation of the running time of a given target algorithm was a major focus in the development of ParamILS and SMAC, and both of them include an *adaptive capping* [13] mechanism that is specifically designed to improve efficiency when dealing with this performance objective. The key idea behind adaptive capping is to reduce the time wasted in the evaluation of poorly performing configurations by bounding the maximum running time permitted for each such evaluation. This bound is calculated based on the best-performing configuration found so far. The use of adaptive capping allows the configurator to prune poorly performing target algorithm configurations early and to quickly focus the configuration budget on promising areas of the space of configurations being searched.

In this work, we improve the efficacy of irace on algorithm configuration scenarios involving running time minimisation. We adapt the ideas of the adaptive capping mechanism into the underlying iterated racing procedure and define an additional dominance criterion based on the performance of the elite configurations obtained by irace. As a first step, we show that by extending irace with adaptive capping, resulting in our new $\text{irace}_{\text{cap}}$ method, we can significantly increase its performance on well-known and difficult configuration scenarios. An additional analysis of various parameters of $\text{irace}_{\text{cap}}$ gives further insights into the importance of the statistical testing procedures and other aspects of irace. A final comparison with other state-of-the-art configuration procedures for running time minimisation, namely ParamILS and SMAC, shows that $\text{irace}_{\text{cap}}$ reaches highly competitive performance and, thus, broadens the range of configuration scenarios for which irace can be considered a possible method of choice.

The remainder of this article is structured as follows. First, we describe irace and the adaptive capping mechanism used by ParamILS (Sections 2 and 3). Next, in Section 4, we describe how we integrated adaptive capping into irace, and we experimentally analyse the resulting $\text{irace}_{\text{cap}}$ in Section 5. In Section 6, we compare $\text{irace}_{\text{cap}}$ to state-of-the-art configurators for minimising running time, and we conclude in Section 7.

2 Elitist iterated racing in irace

irace is an iterated racing procedure [7] for automatic algorithm configuration. It explores the parameter space of a target algorithm by iteratively sampling parameter configurations and applying a racing procedure to select the best-performing configurations. The racing procedure considers a sequence of problem

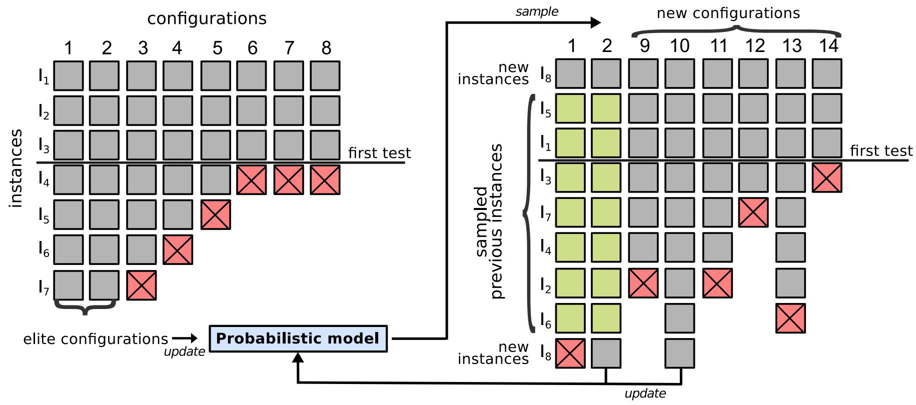

Fig. 1: Illustration of the 1st and 2nd iteration of a run of irace using $T^{\text{first}} = 3$, $T^{\text{each}} = 1$ and $T^{\text{new}} = 1$.

instances on which the candidate configurations are evaluated. At each stage of the race, all candidate configurations are run on a specific problem instance; at the end of the stage, configurations that perform statistically worse than others are eliminated from the race, while all others proceed to the next stage. Once a race is terminated, the best configurations, called *elite*, are used to update the sampling model from which new configurations are generated. The elite configurations are carried over to the next iteration to continue their evaluation within a new race together with the newly generated configurations. One iteration of irace comprises the process of (i) generation of candidate configurations, (ii) execution of the racing procedure, and (iii) update of the probabilistic model.

The irace package [18,17] is an implementation of irace that is publicly available as an R package. Recently, version 2.0 of the software was released, which implements an elitist racing procedure [17]. Differently from the non-elitist racing procedure on which the first version of irace is based, elitist irace evaluates configurations on a set of problem instances that increases in size in every iteration of irace. In particular, in elitist irace, an elite configuration carried over from the previous iteration cannot be eliminated until a better configuration is evaluated on the same instances as the elite one, including all instances on which the elite configuration was previously evaluated and at least one new instance.

In more detail, elitist irace works as follows (see Figure 1). In the first iteration, configurations are sampled uniformly at random from the given configuration space. These configurations are evaluated on T^{first} instances, after which the first statistical test is applied, and the configurations that are significantly worse performing than the best ones are eliminated. This elimination test is performed every T^{each} instances until the termination criterion of the iteration is met. The surviving configurations at the end of the iteration (elite configurations) are used to update a probabilistic model from which new configurations are sampled. The set of configurations evaluated in the next iteration is composed of the elite configurations and newly sampled configurations (non-elite ones). Algorithm 1 shows the pseudo-code of the race performed at each iteration of irace. Instances are evaluated following an execution order that is built interleaving new and old instances (procedure generateInstancesList in line 1).

Algorithm 1: Racing procedure in elitist irace

Inputs are a set of newly generated configurations (Θ^{new}), a user-provided maximum execution time ($b^{\max}$), the number of new initial instances (T^{new}), the list of unseen instances ($\mathcal{I}^{\text{new}}$), a set of elite configurations (Θ^{elite}), the list of instances on which Θ^{elite} were previously evaluated ($\mathcal{I}^{\text{old}}$), and a Boolean predicate $\mathsf{isElite}(\theta, I)$ that returns true if configuration $\theta \in \Theta^{\text{elite}}$ was previously evaluated on instance $I \in \mathcal{I}^{\text{old}}$. In the first iteration of irace, Θ^{elite} and $\mathcal{I}^{\text{old}}$ are empty and all entries of $\mathsf{isElite}(\cdot,\cdot)$ are set to false.

```
Input: $\Theta^{\text{elite}}, \Theta^{\text{new}}, b^{\max}, T^{\text{new}}, \mathcal{I}^{\text{old}}, \mathcal{I}^{\text{new}}, \mathsf{isElite}(\cdot,\cdot)$
Output: Best configurations found in the race.
begin
1       $\mathcal{I} \leftarrow$ generateInstancesList $(T^{\text{new}}, \mathcal{I}^{\text{old}}, \mathcal{I}^{\text{new}})$
2       $i \leftarrow 1$
3       $\Theta^i \leftarrow \Theta^{\text{new}} \cup \Theta^{\text{elite}}$
4       while $\neg$ termination() do
            # execute elites only when needed; $\mathcal{I}[i]$ is the $i$th entry of instance list $\mathcal{I}$
5           $\Theta^{\text{exe}} \leftarrow \Theta^i \setminus \{\theta \in \Theta^{\text{elite}} \mid \mathsf{isElite}(\theta, \mathcal{I}[i])\}$
6           execute $(\Theta^{\text{exe}}, b^{\max}, \mathcal{I}[i])$
7           $\mathsf{isElite}(\theta, \mathcal{I}[i]) \leftarrow$ false $\quad \forall \theta \in \Theta^{\text{elite}}$
8           if mustTest($i$) then
9               $\Theta^{i+1} \leftarrow$ eliminationTest$(\Theta^i, \{\mathcal{I}[1], \ldots, \mathcal{I}[i]\})$
                # keep configurations that are still elite
10              $\Theta^{i+1} \leftarrow \Theta^{i+1} \cup \{\theta \in \Theta^{\text{elite}} \mid \bigvee_{I \in \mathcal{I}^{\text{old}}} \mathsf{isElite}(\theta, I)\}$
11          else
12              $\Theta^{i+1} \leftarrow \Theta^i$
13          $i \leftarrow i + 1$
14      return $\Theta^i$
```

More precisely, the instance list includes T^{new} previously unseen instances, followed by the list of previously evaluated instances ($\mathcal{I}^{\text{old}}$), and finally, enough new instances to complete the race. $\mathcal{I}^{\text{old}}$ is randomly shuffled to avoid a bias that could result from always using the same instance order. A race may terminate even before evaluating all instances in $\mathcal{I}^{\text{old}}$ (e.g. when a minimum number of configurations is reached), and, as a result, each elite configuration may be evaluated on some instances in $\mathcal{I}^{\text{old}}$. When the race finishes, irace therefore memorises which configuration has been evaluated on which instance. (Line 7 updates the elite status, and line 10 tests this condition.) In line 6, the configurations are evaluated on instance $\mathcal{I}[i]$ with a maximum execution time of $b^{\max}$. If an elite configuration was already previously evaluated on $\mathcal{I}[i]$ (i.e., $\mathcal{I}[i] \in \mathcal{I}^{\text{old}}$), its result on that instance is read from memory. When the statistical elimination test is applied, only non-elite configurations (Θ^{new}) may be eliminated and elite ones are kept until they become non-elite. A configuration becomes non-elite if all instances in $\mathcal{I}^{\text{old}}$ on which it has previously been evaluated have been seen in a race. Finally, the race returns the best configurations found, which will become elite in the next iteration. For more details about irace, see [17].

3 ParamILS and adaptive capping

ParamILS [13] is an iterated local search [19] procedure that searches in a parameter space defined by categorical parameters only; for configuring numerical parameters with ParamILS, these need to be discretised. ParamILS uses a first-improvement local search algorithm that explores, in random order, the one-exchange neighbourhood of the current configuration.

There are two versions of ParamILS, BasicILS and FocusedILS, which differ in the number of instances evaluated when comparing two configurations [13]. BasicILS compares configurations by evaluating them on a fixed number N of instances, while FocusedILS varies the number of instances according to the quality of the configurations to be tested. The number of instances used in the comparison is adjusted based on the **dominance criterion**, by which a configuration θ_j is dominated by a configuration θ_i if (1) θ_i has been evaluated in at least as many instances as θ_j and (2) the aggregated performance of θ_i is better or equal than the one of θ_j on the N_j instances on which θ_j has been evaluated. When no dominance can be established between two configurations, the number of instances seen by the configuration with less instances evaluated is increased until both configurations have seen the same number of evaluations. The execution of a configuration on each instance is always bounded by a defined maximum execution time (cut-off time).

The **adaptive capping** technique further bounds the execution of a configuration by using the running time of good configurations as a bound in running time that is often less than the user-specified cut-off time. Using this technique can significantly reduce the time wasted in the evaluation of poor performing configurations. Adaptive capping adjusts the bound on running time according to the number of instances to be used in the comparison, and for this reason, it can be sensitive to the ordering of the given instances. There are two types of adaptive capping: trajectory preserving and aggressive capping [13]. The first of these bounds the running time of new configurations using the performance of the currently best configuration of each ParamILS iteration as reference, while the second additionally uses the performance of the overall best configuration multiplied by a factor, set to two by default, for bounding. This factor controls the aggressiveness of the capping strategy. Further details on adaptive capping can be found in [13].

4 Adaptive capping in irace

In this section, we describe a new version of irace that adopts the ideas underlying adaptive capping in the racing procedure. This new version, $\text{irace}_{\text{cap}}$, introduces two new components to the algorithm: (1) the **adaptive running time bound**, used to limit the running time of new configurations on previously seen and initial instances, and (2) **dominance elimination**, a procedure that discards poorly performing configurations. Algorithm 2 shows the outline of the racing procedure implemented in $\text{irace}_{\text{cap}}$; it follows the same structure as elitist irace, described in Section 2. The elite configurations are first run on the set of initial instances before the start of the race (Line 2). Line 6 calculates an initial running time bound based on the running times of the elite configurations. Let p_i^j be the average computation time of a configuration θ_j up to instance $\mathcal{I}[i]$ in the current iteration. Then, the bound b_i for running new configurations on instance $\mathcal{I}[i]$ is equal to $\text{median}_{\theta_j \in \Theta^{\text{elite}}}\{p_i^j\}$. (Median is chosen to be consistent with the elimination based on dominance described next.) The bound b_i can be computed

Algorithm 2: Racing procedure in $\mathsf{irace}_{\mathsf{cap}}$

For the description of the inputs, see Algorithm 1.

```
Input: $\Theta^{\text{elite}}, \Theta^{\text{new}}, b^{\max}, T^{\text{new}}, \mathcal{I}^{\text{old}}, \mathcal{I}^{\text{new}}, \mathsf{isElite}(\cdot,\cdot)$
Output: Best configuration set found in the race.
begin
1     $\mathcal{I} \leftarrow$ generateInstancesList $(T^{\text{new}}, \mathcal{I}^{\text{old}}, \mathcal{I}^{\text{new}})$
2     execute $(\Theta^{\text{elite}}, b^{\max}, \{\mathcal{I}[1], \dots, \mathcal{I}[T^{\text{new}}]\})$
3     $i \leftarrow 1$
4     $\Theta^i \leftarrow \Theta^{\text{new}} \cup \Theta^{\text{elite}}$
5     while $\neg$ termination() do
6         $b_i \leftarrow$ calculateEliteBound $(\Theta^{\text{elite}}, \{\mathcal{I}[1], \dots, \mathcal{I}[i]\})$
7         $\Theta^{\text{exe}} \leftarrow \Theta^i \setminus \{\theta \in \Theta^{\text{elite}} \mid \mathsf{isElite}(\theta, \mathcal{I}[i])\}$
8         execute $(\Theta^{\text{exe}}, b_i, \mathcal{I}[i])$
9         $\mathsf{isElite}(\theta, \mathcal{I}[i]) \leftarrow$ false   $\forall \theta \in \Theta^{\text{elite}}$
          # dominance criterion elimination
10        $\Theta^{i+1} \leftarrow$ eliminationDominance $(\Theta^i, \{\mathcal{I}_1, \dots, \mathcal{I}[i]\})$
          # statistical test elimination
11        if mustTest $(i)$ then
12            $\Theta^{i+1} \leftarrow$ eliminationTest $(\Theta^{i+1}, \{\mathcal{I}_1, \dots, \mathcal{I}[i]\})$
          # keep configurations that are still elite
13        $\Theta^{i+1} \leftarrow \Theta^{i+1} \cup \{\theta \in \Theta^{\text{elite}} \mid \bigvee_{I \in \mathcal{I}^{\text{old}}} \mathsf{isElite}(\theta, I)\}$
14        $i \leftarrow i + 1$
15    return $\Theta^i$
```

only for previously evaluated instances (including the initial instances); for any other instance, we set the bound to the cut-off time, $b^{\max}$. This running time bound provides a reference of the minimum performance new configurations should obtain in order to compete with the current elite configurations.

The maximum running time k_i^j for each configuration θ_j on instance $I[i]$ is computed by procedure execute in line 8 using the value of b_i as follows:

$$k_i^{\prime j} = b_i \cdot i + b^{\min} - p_{i-1}^j \cdot (i-1) \tag{1}$$

$$k_i^j = \begin{cases} b^{\max} & \text{if } k_i^{\prime j} > b^{\max}, \\ \min\{b_i, b^{\max}\} & \text{if } k_i^{\prime j} \leq 0, \\ k_i^{\prime j} & \text{otherwise;} \end{cases} \tag{2}$$

where $b^{\min}$ is a constant that represents a minimally measurable running time different from zero (set to a default value of 0.01). Intuitively, k_i^j is the time remaining for a configuration θ_j to improve over the median elite configuration.

We implemented a dominance-based elimination procedure inspired by the domination criterion described in Section 3. We compare the median performance of the elite configurations set (Θ^{elite}) on the list of instances $\{\mathcal{I}[1], \dots, \mathcal{I}[i]\}$ considered so far with the performance of the new configurations as follows:

$$\text{Median}_{\theta_s \in \Theta^{\text{elite}}}\{p_i^s\} \; + \; b^{\min} \; < \; p_i^j \tag{3}$$

where p_i^s is the mean running time of configuration θ_s on instances $\{\mathcal{I}[1], \dots, \mathcal{I}[i]\}$, and $b^{\min}$ is the constant defined in Eq. (2). Other choices than the median are possible and may be considered in future work. We eliminate configurations as soon as they become dominated, that is, the dominance-based elimination is applied after every instance seen within an iteration of irace.

5 Experiments

In this section, we study the impact of introducing the previously described capping procedure into irace. We compare the performance of the final configurations obtained by elitist irace and $\text{irace}_{\text{cap}}$ using different settings.

5.1 Experimental setup

In our performance assessments of $\text{irace}_{\text{cap}}$, we use five configuration scenarios taken from previous experimental studies of other automatic algorithm configuration methods, in particular, ParamILS and SMAC. These scenarios use CPLEX [16], Lingeling [5] and Spear [3] as target algorithms, and involve parameter spaces with 74, 137 and 26 parameters, respectively. Their principal characteristics are as follows:

CPLEX - Regions100 [13,12]. 5 seconds cut-off time, 18 000 seconds total configuration budget, and a training and testing set of 1000 mixed integer programming (MIP) instances each. The instances encode a combinatorial auction winner determination problem with 100 goods and 500 bids.

CPLEX - Regions200 [13,11]. 300 seconds cut-off time, 172 800 seconds total configuration budget, and a training and testing set of 1000 MIP instances each. These instances are encodings of a combinatorial auction winner determination problem with 200 goods and 1000 bids.

CPLEX - Corlat [11]. 300 seconds cut-off time, 172 800 seconds total configuration budget, and a training and testing set of 1000 MIP instances each.

Lingeling [14]. 300 seconds cut-off time, 172 800 seconds total configuration budget, and a training and testing set of 299 and 302 SAT instances, respectively. These instances were obtained from the 2014 Configurable SAT Solver Competition (CSSC) [14].

Spear [13]. 300 seconds cut-off time, 172 800 seconds total configuration budget, and a training and testing set of 302 SAT-encoded software verification instances each.

The instance files for these scenarios are also available from the Algorithm Configuration Library (AClib) [15]. AClib specifies a cut-off time of 10 000 seconds for the CPLEX scenarios, which stems from their initial use in conjunction with the CPLEX auto-tuning tool. Following the experiments in [11, Section 5]), we use a cut-off time of 300 seconds.[4] Another minor difference is that we used

[4] A higher cut-off time, as used in AClib, would be detrimental for configuration procedures such as $\text{irace}_{\text{cap}}$, as time-outs would very strongly impact the number of configurations that can be evaluated. On the other hand, there are various techniques, such as early termination of ongoing runs or the initial use of smaller maximum cut-off times, to address this problem. In the literature, the use of smaller cut-off times has been suggested as a possible remedy [12, footnote 9].

version 12.4 of CPLEX, which was installed on our system, while AClib proposes to use version 12.6. However, there is no obvious reason to suspect that the particular version of CPLEX should affect our conclusions on the effect of capping inside irace, and we do not directly compare to results for the original AClib scenarios. Moreover, although both irace and SMAC are able to handle non-discrete parameter spaces, for ParamILS, all parameters have to be discretised, with all possible values specified explicitly in the scenario definition. There is some evidence that the use of non-discrete parameter spaces, where possible, leads to improved results [12], thus giving an advantage to both irace and SMAC over ParamILS, unrelated to the capping mechanism, which is the focus of our comparison presented in Section 6. To avoid this bias, we only consider the variants of the scenarios where all parameters are discretised and explicitly specified.

In all our experiments, we used the t-test to eliminate configurations within irace, as previously recommended for running time minimisation [21]. The comparisons presented in the following are based on 20 independent runs of all configuration procedures; multiple independent configurator runs are performed due to the inherent randomness of the configuration procedures and the configuration scenarios. The experiments were run on one core of a dual-processor 2.1 GHz AMD Opteron system with 16 cores per CPU, 16 MB cache and 64 GB RAM, running Cluster Rocks 6.2, which is based on CentOS 6.2.

In our empirical analysis of $\text{irace}_{\text{cap}}$, we use mean running time as the performance criterion to be optimised by irace. Runs that time out due to reaching the cut-off time are then counted at this maximum cut-off time. In the literature, unsuccessful runs are often more strongly penalised, computing effectively the number of timed out runs multiplied by a penalty factor p_f plus the mean computation time of the successfully terminated runs. In fact, the penalty factor p_f converts the bi-objective problem of minimising the number of timed-out runs and mean time of successful runs into a single-objective problem. In this section, runs of irace attempt to minimise mean running time (with $p_f = 1$), and we therefore assessed the performance of the resulting target algorithm configurations using this performance metric. In the supplementary material, we additionally present results for evaluating configurations using $p_f = 10$ and $p_f = 100$. In the literature, $p_f = 10$ is commonly used and referred to as PAR10; consequently, in Section 6, all configurator runs and target algorithm evaluations are performed using PAR10 scores.

5.2 Experimental results

We first compare the results obtained by elitist irace and $\text{irace}_{\text{cap}}$, using their respective default settings. Table 1 presents performance statistics over the 20 runs of both irace versions. The implementation of the proposed capping procedure proves to be beneficial for the scenarios used in these experiments. For the Regions 100, Regions 200, Corlat, and Spear scenarios, the results obtained by $\text{irace}_{\text{cap}}$ are significantly better than those of elitist irace, while for the Lingeling scenario, the results are not significantly different (however, $\text{irace}_{\text{cap}}$ still achieves a better mean than irace).

The elimination criterion of the capping procedure in $\text{irace}_{\text{cap}}$ only considers aggregated running time rather than its statistical distribution, and it is not ob-

Table 1: Summary statistics of the distribution of observed mean running time and percentage of timed out evaluations of 20 runs of irace$_{\mathsf{cap}}$ and elitist irace (irace) on test sets for the various configuration scenarios. We show the first and second quartile ($q25$ and $q75$, respectively), the median, the mean, the standard deviation (sd) and the variation coefficient (sd/mean). Wilcoxon test p-values are reported in the last line. Statistically significantly better results (at $\alpha = 0.05$) are indicated in bold-face and lowest mean running times in italics.

	Regions 100		Regions 200		Corlat		Lingeling		Spear	
	irace$_{\mathsf{cap}}$	irace	irace$_{\mathsf{cap}}$	irace	irace$_{\mathsf{cap}}$	irace	irace$_{\mathsf{cap}}$	irace	irace$_{\mathsf{cap}}$	irace
%timeout	0.08	0.085	0.01	0.015	0.695	1.205	8.377	8.659	0.397	3.328
q25	0.327	0.374	9.487	10.983	8.616	13.526	42.379	44.274	3.028	4.776
mean	**0.338**	0.395	**10.498**	13.231	**11.899**	15.935	*45.501*	46.923	**4.116**	13.068
median	0.332	0.401	10.469	12.871	9.688	14.911	44.453	47.034	3.765	14.617
q75	0.34	0.413	10.75	14.256	13.941	18.436	48.996	49.738	4.242	19.993
sd	0.018	0.033	1.335	2.908	5.645	4.325	3.799	3.658	1.848	8.092
sd/mean	0.054	0.082	0.127	0.22	0.474	0.271	0.083	0.078	0.449	0.619
p-value	5.7e-06		0.0001049		0.0055809		0.2942524		0.0002613	

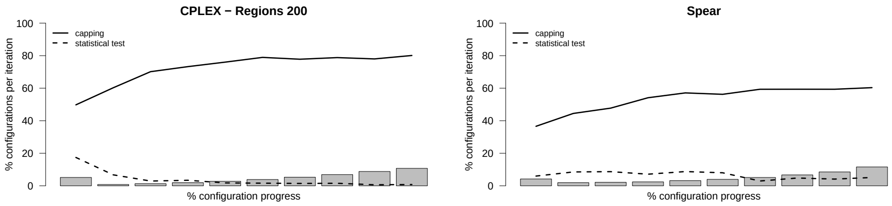

Fig. 2: Mean percentage of configurations selected for elimination by the capping procedure and the statistical test (solid and dashed lines respectively), and mean percentage of initial configurations that become elite configurations at the end of the iteration (bars). Means obtained across 20 independent runs of irace$_{\mathsf{cap}}$ on the Regions 200 and Spear scenarios.

vious whether this renders the criterion always stricter than the statistical test at the core of irace, which could, in principle, render the latter superfluous. Figure 2 shows the mean percentage of live configurations selected to be eliminated by the capping procedure and the statistical test (lines), and the mean percentage of initial configurations that become elite configurations at the end of the iteration (bars). (For results on all other scenarios, see Figure A.2.) The capping procedure selects more configurations for elimination than the statistical test in all stages of the search, while the statistical test is mainly able to eliminate configurations in the initial phases of the search. As the race progresses, capping elimination quickly becomes mainly responsible for eliminating configurations, illustrating the importance of introducing it into irace.

The capping mechanism of irace$_{\mathsf{cap}}$ and the increased elimination of configurations induce a highly intensified search. On average, irace$_{\mathsf{cap}}$ performs in part many more iterations than irace and shows a lower average number of elite configurations per iteration. This results in an increased number of configurations sampled overall and instances used for evaluation (Table 2).

Table 2: Statistics over 20 independent runs of $\mathsf{irace}_{\mathsf{cap}}$ and irace: mean number of iterations performed (iterations), mean number of instances used in the evaluation (instances), mean overall sampled configurations (candidates), mean elite configurations per iteration (elites) and mean total executions (executions).

	Regions 100		Regions 200		Corlat		Lingeling		Spear	
mean	$\mathsf{irace}_{\mathsf{cap}}$	irace	$\mathsf{irace}_{\mathsf{cap}}$	irace	$\mathsf{irace}_{\mathsf{cap}}$	irace	$\mathsf{irace}_{\mathsf{cap}}$	irace	$\mathsf{irace}_{\mathsf{cap}}$	irace
iterations	253.5	28.3	85.8	17.1	68.7	13	27.4	10.5	67.0	7.6
instances	258.6	47.6	91.1	36.7	75.1	29.4	35.5	26.3	83.2	16.3
candidates	27914	1136	5191	285	5318	242	2595	214	11193	718
elites	1.09	6.88	1.25	7.12	1.82	7.80	3.28	8.90	2.26	5.99
executions	30604	8362	6779	2770	8873	3147	5218	2878	28039	6109

5.3 Additional analysis of $\mathsf{irace}_{\mathsf{cap}}$

In what follows, we examine in more detail the impact of some specific parameter settings of $\mathsf{irace}_{\mathsf{cap}}$ on its performance. For the sake of conciseness, we will only discuss overall trends; detailed results are found in supplementary material [20].

Instance order. The order of the instances may introduce a bias in irace when the configuration scenario involves a heterogeneous instance set. By default, irace shuffles the order of the training instances. Without this shuffling, irace evaluates the instances in the order provided by the user. Since the set of previously used instances is evaluated in every iteration, elitist irace randomly permutes the order of previously seen instances ($\mathcal{I}^{\text{old}}$) before each iteration to further avoid any bias that the previous order may introduce. Table A.2 compares the results obtained by $\mathsf{irace}_{\mathsf{cap}}$ with and without this instance reshuffling. For most benchmark scenarios, disabling instance reshuffling produces better mean results and fewer timed-out runs; for Regions 200 and Lingeling, these differences are statistically significant. The main exception is the Spear scenario, where reshuffling leads to much improved results; this is probably due to the fact that this scenario contains a very heterogeneous instances set.

These results suggest that the impact of reshuffling depends on the given configuration scenario; we conjecture that for more heterogeneous instance sets, reshuffling the instance set becomes increasingly important. Investigating this conjecture in detail is an interesting direction for future work.

Confidence level of statistical test. The dominance criterion eliminates more configurations than the statistical test. Lowering the confidence level of the statistical test should lead to an even higher elimination rate of the latter and possibly improve the efficacy of the overall configuration process. We explored this possibility by lowering the confidence level in $\mathsf{irace}_{\mathsf{cap}}$ from its default value of 0.95 to 0.75. Table A.3 in the supplementary material shows the impact of this change. The effects on the elimination of configurations can be observed in Figure A.5 in the supplementary material. As expected, the statistical test eliminates more configurations when setting the confidence level to 0.75. This also results in a small increase in the overall number of configurations evaluated and a reduction of the mean number of elite configurations (see Table A.4 in the supplementary material). The more aggressive test slightly improved the performance for three scenarios, yielding significantly better results for Regions 200. In contrast, a confidence level of 0.75 results in slightly worse performance on the Spear scenario, indicating that the eliminations performed with lower confidence can be premature.

If we completely disable statistical testing (confidence level 1.0), the performance of $\text{irace}_{\text{cap}}$ improves on Regions 100 and Regions 200, as seen in Table A.5 in the supplementary material. This suggests that the statistical test can prematurely eliminate configurations based on an incorrect criterion. Despite this, we still recommend keeping the default confidence level of 0.95, as a safe-guard that may be useful for configuration scenarios with possibly very different properties from the ones we are testing here.

Log-transformation of running times. When used for running time minimisation, irace makes use of the t-test for elimination. However, the potentially very large variability of target algorithm running times [10] often renders the distribution of running times far from normal, a situation that may be alleviated by using a transformation of running times – in particular, a logarithmic transformation. Applying this transformation has, however, only a significant effect on the Regions 200 scenario. Increasing the difference between the performance of configurations makes the elimination more aggressive and, as seen in other experiments, the Regions 200 scenario benefits greatly of this increased intensification. For the other scenarios, the impact on performance is negligible, as seen in Table A.6 in the supplementary material, probably due to the minor impact of the statistical test on the elimination of configurations.

Number of initial new instances. Finally, we performed experiments to evaluate the impact of adding new instances at the beginning of each race. If no new instances are added at the start, then new configurations can only become elite by performing better on exactly the same instances on which the current elites performed well in previous races. Even though the new configurations may be better on instances not seen yet, they may be eliminated before seeing them, unless those new instances are evaluated at the start. On the other hand, there are no running times available for new instances; this issue is addressed by first running the elite configurations on the new instances to avoid wasting too much computation time on possibly poor newly sampled candidate configurations. Table A.7 in the supplementary material shows the results for setting the number of new initial instances (T^{new}) to 0 (new instances are never added at the start of each race), 1 (the default setting) and 5. As previously observed for elitist irace [17], a larger value of T^{new} improves the performance of $\text{irace}_{\text{cap}}$ for the Spear scenario. While for the other scenarios, the differences are minor, there appears to be a tendency for the default value of 1 (or perhaps even a slightly larger value, such as 2 or 3) to result in the most robust behaviour.[5]

[5] Setting T^{new} to 0 may be beneficial for scenarios with a very large cut-off time, as used by default in AClib for the CPLEX scenarios. This should help to aggressively bound the running time at the start of each race, by using the running times of the elite configurations, thus avoiding the high cost of evaluating possibly poor configurations with a very large cut-off time.

Table 3: Statistics over the mean PAR10 performance and percentage of timed-out instances from 20 runs of irace$_{\mathsf{cap}}$, SMAC and ParamILS. Wilcoxon test p-values (significance 0.05). Significantly better results in bold and best mean in cursive.

		q25	mean	median	q75	sd	sd/mean	%timeout
Regions 100 p-value: 0.5958195	ParamILS	0.318	0.38	0.37	0.416	0.066	0.173	0.130
	SMAC	0.45	0.478	0.473	0.499	0.055	0.116	0.045
	irace$_{\mathsf{cap}}$	0.32	*0.372*	0.365	0.395	0.057	0.154	0.095
Regions 200 p-value: 0.03276825	ParamILS	9.412	11.656	10.359	13.606	3.348	0.287	0.005
	SMAC	14.205	17.917	16.452	21.925	5.419	0.302	0.045
	irace$_{\mathsf{cap}}$	8.854	**9.926**	9.349	10.533	1.459	0.147	0.005
Corlat p-value: 0.0083084	ParamILS	30.924	193.303	48.772	74.309	360.42	1.865	5.945
	SMAC	30.866	45.847	39.855	63.805	20.903	0.456	1.005
	irace$_{\mathsf{cap}}$	12.24	**27.974**	26.763	33.902	19.426	0.694	0.62
Lingeling p-value: 0.0362339	ParamILS	250.115	292.529	298.942	327.679	51.497	0.176	9.023
	SMAC	266.792	283.907	289.153	298.64	25.121	0.088	8.758
	irace$_{\mathsf{cap}}$	244.313	**263.651**	259.768	271.119	31.736	0.12	8.113
Spear p-value: $9.5e-06$	ParamILS	3.037	88.083	12.094	40.877	188.929	2.145	2.815
	SMAC	1.6	**3.416**	1.746	2.511	3.733	1.093	0.05
	irace$_{\mathsf{cap}}$	5.666	23.741	22.3	25.872	21.512	0.906	0.662

6 Comparison to other configurators

We compare the results obtained by irace$_{\mathsf{cap}}$ with two other automatic configurators available in the literature, ParamILS and SMAC. Both have been widely used in the literature for running time minimisation. SMAC and ParamILS, as well as irace, were run using default settings. We chose not to include instance features in the configuration process and use only fully discretised configuration spaces; this was done to isolate as much as possible the impact of the new capping mechanism in irace, and to examine whether it would become competitive with other configurators that already used this technique. Considering features or non-discrete parameter spaces would introduce additional factors that are likely to affect performance beyond the impact of capping. Nevertheless, SMAC can also use instance features in the configuration process, which may improve its results; therefore, the results obtained here should be considered with caution for those scenarios in cases where these features are available. Yet, identifying how much of the improvement is due to instance features or due to differences in the capping methods between SMAC and other configurators would require a more extensive analysis that is left for future research. Additionally, SMAC and irace can handle real-valued parameters and, as already shown for SMAC in [12], doing so may further improve performance.

As mentioned previously, we ran irace$_{\mathsf{cap}}$, SMAC and ParamILS using the PAR10 evaluation on the scenarios described in Section 5. Table 3 shows the mean PAR10 execution times obtained from 20 runs of the configurators. In the on-line supplementary material, we present results with other penalty factors

Table 4: Statistics over the mean PAR10 performance for the best-out-of-ten runs sampled from the 20 original runs of irace$_{\mathsf{cap}}$, SMAC and ParamILS. Wilcoxon test p-values ($\alpha = 0.05$). Significantly better results are shown in bold-face and best mean values in italics.

		q25	mean	median	q75	sd	sd/mean
Regions 100 p-value: 8.83e-05	ParamILS	0.303	**0.305**	0.303	0.307	0.003	0.011
	SMAC	0.386	0.392	0.386	0.391	0.012	0.031
	irace$_{\mathsf{cap}}$	0.312	0.314	0.314	0.314	0.002	0.007
Regions 200 p-value: 0.0001417	ParamILS	8.589	8.871	8.999	9.139	0.266	0.03
	SMAC	9.82	11.2	10.106	12.989	1.784	0.159
	irace$_{\mathsf{cap}}$	8.456	**8.523**	8.518	8.581	0.069	0.008
Corlat p-value: 0.0010241	ParamILS	8.284	10.58	9.58	9.58	4.163	0.393
	SMAC	17.801	19.898	19.832	19.832	3.016	0.152
	irace$_{\mathsf{cap}}$	7.959	**8.194**	7.959	8.256	0.627	0.076
Lingeling p-value: 0.6341078	ParamILS	210.753	*218.874*	210.753	223.379	13.256	0.061
	SMAC	230.175	241.659	236.26	257.77	12.85	0.053
	irace$_{\mathsf{cap}}$	220.093	220.212	220.093	220.212	0.212	0.001
Spear p-value: 9e-05	ParamILS	1.911	2.075	2.012	2.073	0.27	0.13
	SMAC	1.454	**1.462**	1.454	1.463	0.018	0.013
	irace$_{\mathsf{cap}}$	2.154	2.497	2.184	2.497	0.581	0.233

from $\{1, 10, 100\}$. The table shows the p-values obtained from the Wilcoxon signed-rank test comparing the performance of the two configurators with the lowest mean PAR10 score. irace$_{\mathsf{cap}}$ obtains the statistically significantly lowest mean on the Regions 200, Corlat, and Lingeling scenarios, while SMAC obtains the statistically significantly lowest mean on the Spear scenario. On the Regions 100 scenario, irace$_{\mathsf{cap}}$ obtains the lowest mean performance value, though its performance is not statistically different from that of ParamILS.

It is known that trajectory-based local search methods, such as ParamILS, can exhibit high performance variability over multiple independent runs due to search stagnation. A common practice for dealing with this situation, and for reducing the overall wall-clock time of the configuration process by means of parallelisation, is to perform multiple independent configurator runs concurrently and to return the best configuration found in any of these. This may not always be feasible when the average running times of configurations on instances are high, e.g., in the range of hours, in which case the parallelisation features of irace would be very useful. Nevertheless, we mimic this commonly applied approach and compare the performance of ParamILS, SMAC and irace$_{\mathsf{cap}}$ based on the following resampling approach: From the 20 values of mean PAR10 performance previously obtained for each configurator on the test set of each scenario, we sample 10 values (uniformly at random and without repetition) and take the best of these samples. This is equivalent to running the configurator 10 times and determining the best of the configurations thus obtained. We repeat this process 20 times to obtain 20 replicates of the experiment. Table 4 shows the results thus obtained. ParamILS benefits most from multiple independent runs, achieving the statistically significantly best performance on the Regions 100 scenario and the best mean performance (though not statistically significantly different from that of irace$_{\mathsf{cap}}$) on Lingeling. irace$_{\mathsf{cap}}$ produces the statistically significantly best results on Regions 200 and Corlat, while SMAC shows the best mean performance for the Spear scenario.

7 Conclusions

In this work, we have extended irace, an automatic algorithm configuration procedure primarily designed for solution quality optimisation, with an adaptive capping mechanism. We have demonstrated that this results in substantial improvements in the efficacy of irace for running time minimisation, and our new irace$_{\mathsf{cap}}$ configurator reaches state-of-the-art performance on prominent configuration scenarios. This considerably broadens the range of configuration scenarios on which irace should be seen as one of the methods of choice.

In future work, it would be interesting to explore which characteristics of a configuration scenario makes it particularly amenable to different variants of adaptive capping. Furthermore, we would like to investigate under which circumstances irace$_{\mathsf{cap}}$ performs better (or worse) than other state-of-the-art configurators, notably SMAC [12], ParamILS [13] and GGA++ [1]. We see this as an important step towards automatic selection of the configurator expected to perform best on a given scenario. This could improve the state of the art in automatic algorithm configuration and further boost the appeal of the programming by optimisation (PbO) software design paradigm [9], which crucially depends on maximally effective configurators.

Acknowledgments. This research was supported through funding through COMEX project (P7/36) within the Interuniversity Attraction Poles Programme of the Belgian Science Policy Office. Thomas Stützle acknowledges support from the Belgian F.R.S.-FNRS, of which he is a Senior Research Associate. Holger Hoos acknowledges support through an NSERC Discovery Grant.

References

1. Ansótegui, C., Malitsky, Y., Samulowitz, H., Sellmann, M., Tierney, K.: Model-based genetic algorithms for algorithm configuration. IJCAI-15, pp. 733–739. IJCAI/AAAI Press, Menlo Park, CA (2015)
2. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: CP 2009, LNCS, vol. 5732, pp. 142–157. Springer (2009)
3. Babić, D., Hutter, F.: Spear theorem prover. In: SAT'08: Proceedings of the SAT 2008 Race (2008)
4. Balaprakash, P., Birattari, M., Stützle, T.: Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In: HM 2007, LNCS, vol. 4771, pp. 108–122 (2007)

5. Biere, A.: Yet another local search solver and Lingeling and friends entering the SAT competition 2014. In: Belov, A., et al. (eds.) Proc. of SAT Competition 2014. Science Series of Publications B, vol. B-2014-2, pp. 39–40. University of Helsinki (2014)
6. Birattari, M.: The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective. Ph.D. thesis, Université Libre de Bruxelles, Belgium (2004)
7. Birattari, M., Yuan, Z., Balaprakash, P., Stützle, T.: F-race and iterated F-race: An overview. In: Bartz-Beielstein, T., et al. (eds.) Experimental Methods for the Analysis of Optimization Algorithms, pp. 311–336. Springer (2010)
8. Hoos, H.H.: Automated algorithm configuration and parameter tuning. In: Hamadi, Y., et al. (eds.) Autonomous Search, pp. 37–71. Springer (2012)
9. Hoos, H.H.: Programming by optimization. Communications of the ACM 55(2), 70–80 (2012)
10. Hoos, H.H., Stützle, T.: Stochastic Local Search—Foundations and Applications. Morgan Kaufmann Publishers (2005)
11. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: CPAIOR, LNCS, vol. 6140, pp. 186–202. Springer (2010)
12. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: LION 5, LNCS, vol. 6683, pp. 507–523. Springer (2011)
13. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306 (2009)
14. Hutter, F., Lindauer, M.T., Balint, A., Bayless, S., Hoos, H.H., Leyton-Brown, K.: The configurable SAT solver challenge (CSSC). Artificial Intelligence 243, 1–25 (2017)
15. Hutter, F., López-Ibáñez, M., Fawcett, C., Lindauer, M.T., Hoos, H.H., Leyton-Brown, K., Stützle, T.: AClib: a benchmark library for algorithm configuration. In: LION, LNCS, vol. 8426, pp. 36–40. Springer (2014)
16. IBM: ILOG CPLEX optimizer. `http://www.ibm.com/software/integration/optimization/cplex-optimizer/`
17. López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., Birattari, M.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives 3, 43–58 (2016)
18. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package, iterated race for automatic algorithm configuration. Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011)
19. Lourenço, H.R., Martin, O., Stützle, T.: Iterated local search: Framework and applications. In: Gendreau, M., et al. (eds.) Handbook of Metaheuristics, pp. 363–397. Springer, 2 edn. (2010)
20. Pérez Cáceres, L., López-Ibáñez, M., Hoos, H.H., Stützle, T.: An experimental study of adaptive capping in irace: Supplementary material. `http://iridia.ulb.ac.be/supp/IridiaSupp2016-007/` (2017)
21. Pérez Cáceres, L., López-Ibáñez, M., Stützle, T.: An analysis of parameters of irace. In: EvoCOP 2014, LNCS, vol. 8600, pp. 37–48. Springer (2014)