Universiteit Leiden
The Netherlands

Sampling strategies in automated algorithm configuration

Anastacio, M.I.A.

6

Instance selection within automated algorithm configuration

In the previous chapter, we showed that selecting on which instances to run two algorithms and performing statistical tests allows us to make an accurate decision regarding which algorithm performs best on a set of instances in a fraction of the time it would require for running them on all instances.

Based on this evidence that selecting instances can unlock significant speed-ups in comparing the running times of algorithms, we extend this method to compare two configurations of the same algorithm and integrate it into an automated algorithm configurator, specifically, SMAC. In this chapter[1], we study the potential of this approach applied in the context of automated algorithm configuration (AAC). We adapt the previously studied selection methods to leverage the empirical performance models used in model-based configurators, and we also introduce two methods inspired by the active learning literature.

First, we test those methods on setup of experiments representing two situations requiring performance comparison that arise during the configuration process. Our empirical evaluation on six benchmarks shows that, depending on the problem in-

[1]Parts of this chapter have been published as Anastacio et al. (2022)

stances and their running time distributions, a decision can be reached 5 to 3000 times faster than with uniform random sampling, the method used in current state-of-the-art configurators. Then, we integrate the best-performing methods into the model-based configurator SMAC and evaluate the resulting configurator on five running time optimisation configuration scenarios from the literature. On two out of those, we almost double the performance improvement achieved by vanilla SMAC, allowing in one case to reach previously unseen performances (compared to Chapter 3). Additionally, we perform an ablation study to confirm that the instance selection mechanism is indeed responsible for this improvement and confirm that the selection of instances has very substantial potential in improving the state of the art in automated algorithm configuration.

6.1 Introduction

Comparing the performance of two configurations of a given algorithm is a key element of procedures for solving the AAC problem, since such comparisons are performed many times during the configuration process. However, in an automated algorithm configurator, the most computationally expensive task is to evaluate the quality of candidate parameter configurations. Executing time-consuming runs of the target algorithm on different problem instances to determine which parameter settings achieve the best performance requires substantial resources, and time is often wasted on less promising configurations as well as on instances that require a long running time to solve, regardless of the configuration utilised. With the increasing focus on sustainability, the computational resources and the environmental impact associated with the use of AI methods should be put under scrutiny, providing additional incentives to configure algorithms, but also to reduce the computational cost of AAC (see *e.g.* Tornede et al., 2023).

Several lines of research attempt to tackle this problem, mainly focusing on the idea of discarding configurations that are not sufficiently promising. For anytime algorithms, such as machine learning methods, there has been work on early stopping less promising runs based on learning curves (see *e.g.* Domhan et al., 2015; Luo et al., 2019), while adaptive capping mechanisms, such as the ones included in paramILS and irace (Hutter et al., 2009; López-Ibáñez et al., 2016), permit the early stopping of evaluations of configurations unlikely to be competitive with previously evaluated ones. Those lines of research are focused on the idea of discarding configurations deemed insufficiently promising.

On the other hand, in Chapter 5 (see also Matricon et al., 2021), inspired by the field of active learning, we explored the idea of selecting on which instances to compare two given algorithms. We introduced the per-set efficient algorithm selection problem (PSEAS) problem, which appears during AAC, albeit in a slightly different form. Rather than selecting an algorithm, the configurator needs to select a specific configuration of an algorithm among others.

In the following, Building on research from several areas, we aim to identify instances that help discriminate between the compared configurations. We argue that carefully selecting instances and avoiding long evaluations that provide only a limited amount of information allows the configurator to decide faster whether or not it should reject less promising configurations.

6.1.1 Background

In a configurator, the most expensive part is to evaluate the quality of numerous parameter configurations, executing time-consuming runs of the target algorithm on different problem instances to determine which parameter settings achieve the best performance. For anytime algorithms, such as machine learning methods, there has been work on early stopping less promising runs based on the learning curve (*e.g.* Domhan et al., 2015; Luo et al., 2019), *i.e.,* the curve representing the evolution of the performance of the model on the training (or validation) set during training. In contrast, adaptive capping, such as the mechanism included in paramILS (Hutter et al., 2009) and irace (López-Ibáñez et al., 2016), allows for early stopping the evaluation when a configuration is already deemed not to be competitive with the current incumbent (*i.e.,* , the best currently known). Those lines of research are focused on the idea of discarding configurations deemed insufficiently promising.

On the other hand, the per-set efficient algorithm selection problem (PSEAS) as defined in Chapter 5 appears during AAC, albeit in a slightly different form. Rather than selecting an algorithm, the configurator needs to select one of many configurations of a given target algorithms. In a configurator, the prior information on which we can base our instance selection comes in the form of prior runs of other configurations on the instances and of instance features. The latter are used in particular when learning a surrogate model, which is why we limit ourselves to model-based configurators. Because we have a model, our work relates to the questions addressed by active learning methods, which we therefore incorporate into our study.

6.1.2 Research questions

The work presented in this chapter addresses two main research questions, each subdivided into two questions addressed through our experiments.

RQ7 *How can we smartly select on which instances to run our evaluation to lower the time spent evaluating bad configurations?*

We evaluated our methods outside the configurator on artificially generated running time data. We defined two phases of comparison within the configuration process. The first phase takes place on a subset of instances on which configurations have been run before, and the second on instances never seen before. We conducted separate experiments to evaluate the performance of the selection methods within these two phases, aiming to answer the following questions:

RQ 7.1. How does the selection method perform when comparing a new configuration to the incumbent on the subset of instances for which we already collected information throughout the configuration run, as seen in phase 1?

RQ 7.2. How does the selection method perform when comparing a new configuration to the incumbent on all instances, selecting instances for which we did not collect information throughout the configuration run, as seen in phase 2?

To answer those questions, we adapted the two best-performing selection methods from Chapter 5 (also Matricon et al., 2021) with the performance model used in model-based configurators, added two methods inspired by the active learning literature (Gu et al., 2014), and evaluated them on five benchmarks. We designed two sets of experiments, showing that, depending on the problem instances and their running time distribution, the decision to stop evaluating a less promising configuration could be reached 5 to 3000 times faster than with random sampling, the method currently used in most state-of-the-art configurators.

RQ8 *How can instance selection boost the configurator performance or speed?*

We included the best-performing methods in a state-of-the-art configurator and evaluated their performance. Since SMAC (Hutter et al., 2011b) does not include any statistical test to decide when to discard configurations as soon as there is sufficient evidence for doing so, we implemented this test, as well as the selection mechanisms that showed the best result on our initial evaluation. We conducted an ablation study to evaluate the impact of the test we newly introduced without instance selection. This allows us to answer the following questions:

RQ 8.3. Do sophisticated instance selection mechanisms allow us to improve over picking instances uniformly at random?

RQ 8.4. How does the introduction of a statistical test during the comparison impact the performance of the configurator – in our case SMAC?

We evaluated the resulting configurator on five configuration scenarios from the literature and found that the method shows great potential, almost doubling the improvement reached by vanilla SMAC on two out of five scenarios and reaching previously unseen performances (compared to Chapter 3) on one of them (EAX solver on `rue-1000-3000`). Our ablation study confirmed that the origin of the improvement lies in the selection method or in the combined effect of both new mechanisms.

6.2 Comparison of two configurations

We want to efficiently compare two configurations of a single algorithm. This problem is similar to the comparison of two algorithms studied in Chapter 5. However, when comparing two algorithms, the distribution of running times for each algorithm is potentially completely different, with each algorithm taking advantage of different elements or structures in the problem instances. Comparing configurations of a single algorithm requires to detect smaller changes, potentially simple shifts of the distribution of running time. It is this important to evaluate if the methods are able to detect those changes. To do so, we need to gather enough statistical evidence while using the least possible amount of computing time.

6.2.1 Instance selection

Following the definition of AAC in Chapter 2, $\mathcal{I}$ is the finite set of instances and Ω is the set of valid configurations of the algorithm at hand. At a given step, we have partial running time information on $\mathcal{I}_{\text{known}} \subseteq \mathcal{I}$ for configurations in $\Omega_{\text{known}} \subseteq \Omega$, which means that for $\omega \in \Omega_{\text{known}}$, there exists information about the performance of ω on at least one instance of $\mathcal{I}_{\text{known}}$.

When comparing a challenger configuration ω_{ch} to the incumbent configuration ω_{inc}, instance selection appears in two forms. In Algorithm 6.1, a high-level description of how the comparison is conducted in SMAC (corresponding to line 10 of Algorithm 2.4 in Section 2.2.2), these are found in lines 6 and 10 (coloured purple), but the same mechanisms arises in any configurator. The first of these, which we name *phase 1*, corresponds to the PSEAS problem (Chapter 5), where we already know the performance of ω_{inc} on a set of instances $\mathcal{I}_{\text{known}}$ and want to determine whether ω_{ch} performs better on this set. The second, which we name *phase 2*, corresponds to a case where we know the performance of both ω_{inc} and ω_{ch} on $\mathcal{I}_{\text{known}}$ and want to evaluate both configurations on additional instances from $\mathcal{I} \setminus \mathcal{I}_{\text{known}}$. This can happen, for example, by steadily increasing the size of $\mathcal{I}_{\text{known}}$ at each iteration of the configuration process, or only when we do not have sufficient information to decide which one is the best. Since with our method we would have already discarded ω_{ch} if it was worse than ω_{inc}, and considering our goal of lowering the number of evaluations of the target algorithm, we will focus on this second case in the following.

In both cases, we seek to iteratively choose an instance $I \in \mathcal{I}_{\text{choose}} \subseteq \mathcal{I}$ and gather performance information on it until we satisfy a stopping condition.

Algorithm 6.1 Intensification for one challenger (based on SMAC Hutter et al. (2011b))

Input ω_{inc}: the incumbent configuration, ω_{ch}: the challenger configuration, $n_{\max}$: maximum number of new instances on which to run ω_{inc}, $\mathcal{I}$: the training instances, $\mathcal{I}_{\text{known}}$: the instances on which ω_{inc} was run.
Output ω_{ch}: the best found configuration.

if Insufficient runs for configuration ω_{inc} **then**
    execute a run of ω_{inc} on an instance not run yet, sampled uniformly at random
end if
$N \leftarrow 1$
while there are instances on which ω_{inc}, but not ω_{ch}, has been run **do**
    execute runs of ω_{ch} **on N instances from $\mathcal{I}_{\text{known}}$ on which ω_{ch} has not been run**
    if ω_{ch} is worse than ω_{inc} **then**
        return ω_{inc}
    else $N \leftarrow N \cdot 2$
    end if
end while
while ω_{inc} and ω_{ch} cannot be distinguished **do**
    if $N_{\text{run}} < n_{\max}$ **then**
        run ω_{inc} and ω_{ch} on **an instance from $\mathcal{I} \setminus \mathcal{I}_{\text{known}}$**
        $N_{\text{run}} \leftarrow N_{\text{run}} + 1$
    else
        return ω_{inc}
    end if
end while
return best of ω_{ch}, ω_{inc}

In phase 1, $\mathcal{I}_{\text{known}}$ is the subset of instances on which we have run our incumbent ω_{inc} so far, and ω_{inc} is the best performing configuration known to us on $\mathcal{I}_{\text{known}}$. At the first step, we have no performance information regarding ω_{ch}. At each step, we select an instance I from $\mathcal{I}_{\text{choose}}$, run ω_{ch} on I and add it to $\mathcal{I}_{\text{selected}}$. At any step, $\mathcal{I}_{\text{selected}} \subseteq \mathcal{I}_{\text{known}}$ and $\mathcal{I}_{\text{choose}} = \mathcal{I}_{\text{known}} \setminus \mathcal{I}_{\text{selected}}$. During this phase, we want to discard ω_{ch}, given sufficient evidence that it performs worse than ω_{inc}.If ω_{ch} performs as well or better than ω_{inc}, we would need to run it on all instances of $\mathcal{I}_{\text{known}}$ before applying the second phase or continuing the configuration process, thus we do not discard ω_{inc} early. Moreover, ω_{inc} already showed evidence that it performs better than previously tested challengers and thus comes with stronger evidence of good performance. Thus, our stopping criterion is either to have $\mathcal{I}_{\text{choose}} = \emptyset$, or to be confident that ω_{ch} is worse than ω_{inc}. We consider that, to select instances, we have access to an empirical prediction model trained on all pairs $(\omega, I) \in \Omega_{\text{known}} \times \mathcal{I}_{\text{known}}$, such that $\mathcal{M}(\omega, I)$ is known, and predicting the performance for any pair of instance and configuration.

In phase 2, we also have a subset $\mathcal{I}_{\text{known}} \subset \mathcal{I}$, but unlike in phase 1, there is no asymmetry between ω_{inc} and ω_{ch}. We know their running time on all instances of

$\mathcal{I}_{\text{known}}$ and both can be discarded given sufficient evidence. The goal is to be able to decide which of ω_{inc} and ω_{ch}, whose performance on $\mathcal{I}_{\text{known}}$ cannot be distinguished reliably, actually is to be preferred; to achieve this, we can select instances from $\mathcal{I}_{\text{choose}} = \mathcal{I} \backslash \mathcal{I}_{\text{known}}$ and iteratively add them to $\mathcal{I}_{\text{known}}$. Since no configuration has been run on any of the instances in $\mathcal{I}_{\text{choose}}$, we predict the performance of ω_{inc} and ω_{ch} with a predictive model trained on the performance of the configurations from Ω_{known} on the instances from $\mathcal{I}_{\text{known}}$. To do so, we require instance features, as defined in previous work for a broad range of problems, *e.g.* for Boolean satisfiability (SAT) (Xu et al., 2008), mixed integer programming (MIP) (Xu et al., 2011) or traveling salesperson problem (TSP) (Mersmann et al., 2013). In this phase, the stopping criterion is either to be able to clearly separate the performance of ω_{inc} and ω_{ch} on $\mathcal{I}_{\text{known}}$, or to reach a predefined maximum number $n_{\max} \in \mathbb{N}$ of instances added during the process.

In Chapter 5, we have used selection methods to decide which of two given solvers for an NP-hard problem performs best. To do so, each instance is assigned a score designed to reflect the relevance of choosing that instance both in terms of information obtained and cost incurred. The highest-scoring instance is chosen iteratively until one solver is deemed to have shown better performance than the other. Since we are working with similar types of solvers, we expect that a similar approach would be promising in our situation. We assign scores to instances from $\mathcal{I}_{\text{choose}}$ and select iteratively the highest scoring instance $I^* \in \operatorname{argmax}_{I \in \mathcal{I}_{\text{choose}}} score(I)$. We adapted two of the best methods tested on PSEAS to support the partial-information context. Note that these methods do not take advantage of the model in phase 1, while in phase 2, they are using the predictions given by the model as if they were ground truth. We did not adapt the information-based method, as it relies on assumptions regarding the performance distribution that could not be made in the current context. For PSEAS, this method relied on distributions estimated on the runs of Ω_{known} on all the instances from $\mathcal{I}$. In our current context, we only have information on $\mathcal{I}_{\text{known}}$, which is influenced heavily by the selection method itself; the estimated distributions would hence be strongly biased. Considering work from the active learning literature applicable to a random forest regression model – the model used in SMAC and previously demonstrated to be most effective for empirical performance prediction (Hutter et al., 2014b) –, we chose to adapt the work of Gu *et al.*, which considers active learning for terrain classification using random forests (Gu et al., 2014). Other works (*e.g.* Bhosle and Kokare, 2020; Ayerdi and Graña, 2015) have used similar ideas, focusing on the uncertainty of the model, so we also include a measure solely based on uncertainty.

Baseline: Uniform random sampling

This is equivalent to assigning every instance the same score, and thus sampling an instance uniformly at random.

Discrimination

This method, originally inspired by the work of Gent et al. (2014), aims to choose the instance that best discriminates between the best and other configurations. We say that a configuration ω is ρ-*dominated* on an instance I, for a given $\rho > 1$, if there exists another configuration ω' such that $\mathcal{M}(\omega', I) \leq \rho \cdot \mathcal{M}(\omega, I)$. Thus, we define the *discrimination quality* of an instance I, denoted $Q(I)$, as the fraction of known configurations that are ρ-dominated on this instance. The score is then defined as the discrimination quality divided by the mean running time of the instance:

$$score(I) = \frac{Q(I)}{Mean(I)}.$$

Variance

This approach is based on the intuition that an instance with high variance is likely to discriminate between two configurations. To also take into account the cost of running this instance, we divide the variance by the mean running time of the instance. Note that, according to Section 5.4.3, the underlying distribution of running times follows a Cauchy distribution and would thus not have a well-defined mean or variance. However, due to running times being bounded by 0 and the cutoff time, it is a truncated Cauchy distribution, which is well-behaved and has a mean and a variance. Our score is thus the relative variance

$$score(I) = \frac{Var(I)}{Mean(I)}.$$

Uncertainty-Diversity-Density (UDD)

This method is inspired by the work of Gu et al. (2014) from the active learning literature mentioned earlier. We decided to take the core ideas for their classification model and adapt it to our regression model. We named this approach UDD, because it is based on a combination of three scores: uncertainty, diversity and density. All three scores are scaled and translated to the interval $[0; 1]$ before computing $score(I)$

as

$$\begin{aligned} score(I) = &\ Uncertainty(I) \\ &+ \alpha \cdot Diversity(I) \\ &+ \beta \cdot Density(I). \end{aligned}$$

$Uncertainty(I)$ is the variance of the random forest on running time predictions for instance I and $Diversity(I) = -min_{I' \in \mathcal{I}_{\text{known}}} \mathbb{D}(I, I')$, where $\mathbb{D}$ is a distance function over instances. Intuitively, the closer I is to instances from $\mathcal{I}_{\text{known}}$, the more unlikely it is to provide additional information. Finally, $Density(I) = \frac{1}{k} \cdot \sum_{I' \in \mathcal{N}_k(I, \mathbb{D})} \mathbb{D}(I, I')^2$ where $k \in \mathbb{N}$ is a parameter, $\mathbb{D}$ is a the same distance function as for diversity and $\mathcal{N}_k(I, \mathbb{D}, \mathcal{I}_{\text{choose}})$ returns the k closest neighbours of I in $\mathcal{I}_{\text{choose}} \setminus \{I\}$ according to $\mathbb{D}$. Intuitively, if an instance I is close to many instances from $\mathcal{I}_{\text{choose}}$, then running I should also provide information about these other instances.

Uncertainty

This corresponds to UDD with α and β set to zero, which is reminiscent of the variance method applied to the predictions of a model instead of measured performance values.

6.2.2 Stopping criterion

At each phase, we need to decide when we consider that sufficient statistical evidence has been gathered. In Algorithm 6.1, this decision appears in lines 7 and 12. Based on previous work (Matricon et al., 2021; López-Ibáñez et al., 2016), we use a *Wilcoxon matched-pairs signed-rank test* (Conover, 1998) with a significance level of 0.05.

6.3 Setup of experiments

The final goal of this chapter is to include the instance selection methods presented in the previous section in a configurator at both phases and to assess the impact of these modifications on the performance. However, directly including them without decomposing the mechanism into smaller, more easily analysed components would give us little to no information about which of the components shows the desired impact.

Following our research questions, our evaluation is divided into two main sections, each subdivided into two research questions. We evaluated our methods outside the configurator on artificially generated running time data, conducting experiments to

evaluate the performance of the selection methods, separately for phase 1 and phase 2 defined earlier. Then, we included the best performing methods in the state-of-the-art configurator SMAC and conducted an ablation study to evaluate the impact of the statistical test we newly introduced with and without instance selection.

6.3.1 Datasets

We used configuration scenarios taken from the Algorithm configuration library AClib (Hutter et al., 2014a) or derived from these. Table 6.1 reminds the features of the considered scenarios; more details are shown in Chapter 2.

Evaluation outside of a configurator

We evaluated our method on two NP-hard problems that have been well-studied in the algorithm configuration literature: SAT and MIP. For each, we chose two widely used datasets from AClib and added a more recent and harder dataset to test the limits of our methods.

For SAT, we used CF and IBM from AClib and generated a new set of cryptography instances based on the work of Nejati and Ganesh (2019). Specifically, we used the sha256 encoding, 16 to 60 rounds and an input size of 2^n with $n \in \mathbb{N}, n \leq 10$. For this last dataset, we set the cutoff time to 5000 seconds, such that 70% of the instances can be solved by the default configuration before reaching this time limit. Based on the results of the SAT competition 2020 (Balyo et al., 2020), we decided to configure Kissat (Biere et al., 2020), the best SAT solver currently available.

For MIP, we used RCW2, REG200 from AClib and added a more difficult dataset based on the work of König et al. (2021), which is comprised of challenging neural

Table 6.1: Benchmark instance sets characteristics.

Name	Origin	Training size	Testing size	Features	Clusters
CF	generated	298	301	113	14
IBM	real-world	382	302	113	21
Crypto	generated	225	225	103	8
CLS	generated	50	50	148	3
RCW2	real-world	495	495	148	6
REG200	generated	999	999	148	2
MIPverify	generated	92	92	206	5
rue-1000-3000	generated	50	250	64	9

network verification problems. For this last dataset, we set the cutoff-time to 9000 seconds, such that 70% of the instances can be solved by the default configuration before reaching this time limit . For these scenarios, we chose CPLEX, since it is well known in the literature and also prominently used in AClib.

Evaluation inside a configurator

We chose 2 TSP scenarios and 3 MIP scenarios. Because we had to run many different versions of the configurators, we selected well-studied scenarios with a relatively low time budget. *For TSP,* we used two datasets from AClib (EAX and LKH on rue-1000-3000) due to their short configuration time (see *e.g.* Pushak and Hoos, 2020). *For MIP,* we used three datasets from AClib (REG200, CLS and RCW2) well-known from the literature (see *e.g.* Hutter et al., 2010b). For this scenario, we use a cutoff time of 300 seconds (following Cáceres et al. (2017)), since our method is more suited for situations in which runs might be cut off upon reaching the time limit. Using a cutoff time of 10 000 seconds results in all runs completing before the cutoff is reached.

6.3.2 Implementation details

Our implementations are available on GitHub[2]. The UDD method requires a distance function $\mathbb{D}$ in the instance space; we compute this using the same procedure as in Chapter 5, where we find weights for instance features and compute a weighted feature distance between instances. Since the discrimination and UDD methods have parameters, we tuned those with a simple grid search on a separate scenario (Kissat with the SWGCP dataset from AClib). For discrimination, we evaluated values in $[1.01; 2]$ with a step size of 0.11 and found that $\rho = 1.12$ performed well on both phases. For UDD, we evaluated values in $[0; 2]$ with a step size of 0.21 for both values independently and found that $\alpha = 0.2$ and $\beta = 1.4$ performed well on both levels.

Evaluation outside the configuration process

To carry out our empirical investigation, a dataset of configurations and their associated performance scores were required. To obtain such a set, we generated 100 random configurations uniformly at random for each solver and ran them on all instances of the datasets included in the respective configuration scenarios. This allowed us to

[2]The implementation of the first set of experiments is found at `github.com/Theomat/MPSEAS` and that of the second set of experiments is found at `github.com/ADA-research/SMACIS`

collect performance data on many pairs of problem instances and algorithm configurations. We used the same random forest model as in SMAC (Hutter et al., 2011b) as an empirical performance model (EPM). We trained this EPM on the previously described performance dataset. To evaluate how efficient our methods will be along a configuration run, we trained the EPM on various amounts of performance data: the number of known configurations is in $[10, 20, 30, 40, 50]$ and the amount of known instances is a fraction of $[0.1, 0.2, 0.3, 0.4, 0.5]$ of the full dataset.

Evaluation inside the configuration process

This evaluation required us to include the selection method in the configurator. As our first evaluation is based on the inner working of SMAC, we included the methods in SMAC3 version 1.1.1. However, in principle, a similar mechanism could be used in any configuration procedure.

We included in SMAC a *Wilcoxon matched-pairs signed-ranks test* (Conover, 1998) with a significance level of 0.05 between the runtime of the incumbent ω_{inc} and the challenger ω_{ch} to decide if the challenger can be discarded. Remember that in phase 1, we only discard the challenger in the presence of sufficient evidence that it performs worse than the incumbent and never decide to replace the incumbent based on the test. This means that we take the risk of discarding good configurations in case of error, but would not risk replacing the incumbent with a worse configuration (on known instances).

Due to the large number of statistical tests involved, we need to account for the problem of multiple testing. First, we do not perform tests before running ω_{ch} on at least 5 instances based on the recommended smallest number of samples for the statistical test to be effective (Conover, 1998). Moreover, we use batches to lower the number of tests performed. For each test, we apply a Bonferroni correction (Dunn, 1961), which means that we divide the significance threshold by the number of tests to be performed (given by the size of $\mathcal{I}_{\text{known}}$ divided by our batch size) before comparing it to our confidence threshold. Note that if we use a fixed batch size, the larger the number of instances in $\mathcal{I}_{\text{known}}$, the lower the p-value would need to be to reject the null-hypothesis. Along the configuration process, more instances are added to $\mathcal{I}_{\text{known}}$, and it would become very unlikely to reject a new incumbent, whilst the time to compare configurations will become larger due to the number of runs required. This phenomenon would counteract our goal to lower the comparison computation cost. Thus, we decide to set our batch size relatively to the size of $\mathcal{I}_{\text{known}}$. Based on the results of Matricon et al. (2021), we decided to test every 20% of $\mathcal{I}_{\text{known}}$;

this corresponds to an amount of instances above which the Wilcoxon test had high accuracy in their reported results for most of the selection methods.

Due to the large computation time required to evaluate every possible combination of methods between phase 1 and phase 2, we had to carefully select a subset of possible experiments; specifically, we only considered the best-performing methods from the first set of experiments at each phase of the configuration. Because our method involves adding a Wilcoxon test to stop comparisons early, we also evaluate its impact separately, to gain further insights into the observed behaviour. To compare the performance of two versions of the configurator, we want to look at the expected best performance of the best found configuration. Since a user would typically run the configurator several times and select the configuration found to perform best on the given training data (which corresponds to the so-called standard protocol), we apply the following protocol: we run the configuration 8 times with seeds from 1 to 8, repeatedly sample 5 runs uniformly at random from that set of 8, and identify the best of these according to performance on the training set. We used 1000 such samples to estimate the probability distribution of the quality of the result produced by each configurator on each configuration scenario. We then compared the medians of these empirical distributions. This is similar to procedures used in the literature (see *e.g.* Pushak and Hoos, 2020; Anastacio and Hoos, 2020b).

6.3.3 Execution environment

The first set of experiments was run on the high-performance compute cluster *Grace*, hosted by Leiden University, running CentOS Linux operating system version 8.5. Each node is equipped with two Intel Xeon E5-2683 CPUs with 16 cores and 40MB cache each, as well as 94GB RAM. The second set of experiments was run on the high-performance cluster *Kathleen*, hosted by RWTH Aachen University, running Rocky Linux operating system version 9.3. Each node is equipped with two AMD EPYC 7543 CPUs with 32 cores and 256 MB of cache each, as well as 1 TB of RAM.

6.4 Evaluation outside the configuration process

This section describes the results obtained from our first set of experiments. The goal of these experiments was to evaluate how well the selection methods perform at both phases described in Section 6.2, independently of the whole configuration procedure around. We show aggregated results here, but the raw results and scripts to generate

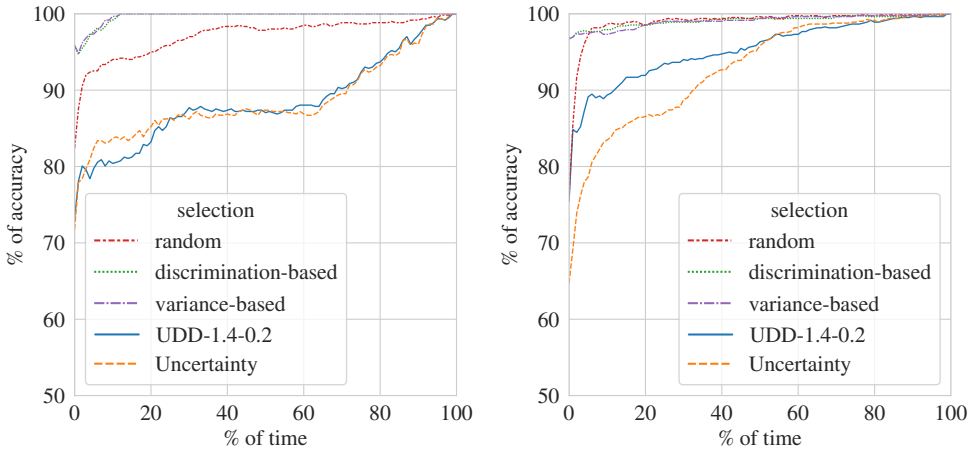
more visualisations are available in our git repository.

6.4.1 Compare configurations on known instances

To answer the first question – *How does the selection method perform to compare a new configuration to the incumbent on the subset of instances for which we already collected information throughout the configuration run, as seen in phase 1*? – we consider phase 1 (see In phase 1,). We populate $\mathcal{I}_{\text{known}}$ and Ω_{known} with instances and configurations, respectively, selected uniformly at random. We note that this does not reflect accurately the way these sets are populated during a configuration run since each configurator will bias their search according to their configuration sampling mechanisms. We choose $\omega_{\text{inc}} \in \text{argmin}_{\omega \in \Omega_{\text{known}}} \mathcal{M}(\omega, \mathcal{I}_{\text{known}})$ and train the random forest model on all the available data. Then we pick configurations from $\Omega \setminus \Omega_{\text{known}}$ as ω_{ch} and run our iterative process; we refer to this as one run. We stop when we have run all instances of $\mathcal{I}_{\text{selected}}$. After each new instance is added, we report the percentage of time that has been spent up to that point to evaluate $\mathcal{M}(\omega_{\text{ch}}, \mathcal{I}_{\text{selected}})$ compared to running it on all instances of $\mathcal{I}_{\text{known}}$, and we perform a *Wilcoxon matched-pairs signed-ranks test* (Conover, 1998) with a significance level of 0.05 to decide if the challenger can be discarded. If $\mathcal{M}(\omega_{\text{ch}}, \mathcal{I}_{\text{selected}}) > \mathcal{M}(\omega_{\text{inc}}, \mathcal{I}_{\text{selected}})$ and the statistical test indicated statistical significance, ω_{ch} is discarded. We compare the resulting decision to the ground truth given by comparing $\mathcal{M}(\omega_{\text{ch}}, \mathcal{I}_{\text{known}})$ to $\mathcal{M}(\omega_{\text{inc}}, \mathcal{I}_{\text{known}})$ to assess the accuracy of the decision. For a given pair of $(\mathcal{I}_{\text{known}}, \Omega_{\text{known}})$, we performed 10 independent runs, using different pseudo-random number seeds, and report the average over those runs.

Figure 6.1 shows the collected accuracy over the time spent to make the comparison for two examples. Figure 6.1a is a case in which the discrimination and variance methods are significantly more accurate than the three others at any given time, while UDD and uncertainty show lower accuracy than random sampling. Figure 6.1b is a case in which discrimination and variance methods start with an advantage over random but quickly reach the same accuracy; once again, UDD and uncertainty perform substantially worse.

Figure 6.2 summarises the previously described curves by computing the area under the curve (AUC) for all tested amounts of prior data; the higher the AUC, the faster and more accurately the decision can be taken. This visualisation allows us to examine how the methods compare, but also illustrates the impact of the prior data used on the empirical performance model. In all our scenarios, we can see a clear correlation



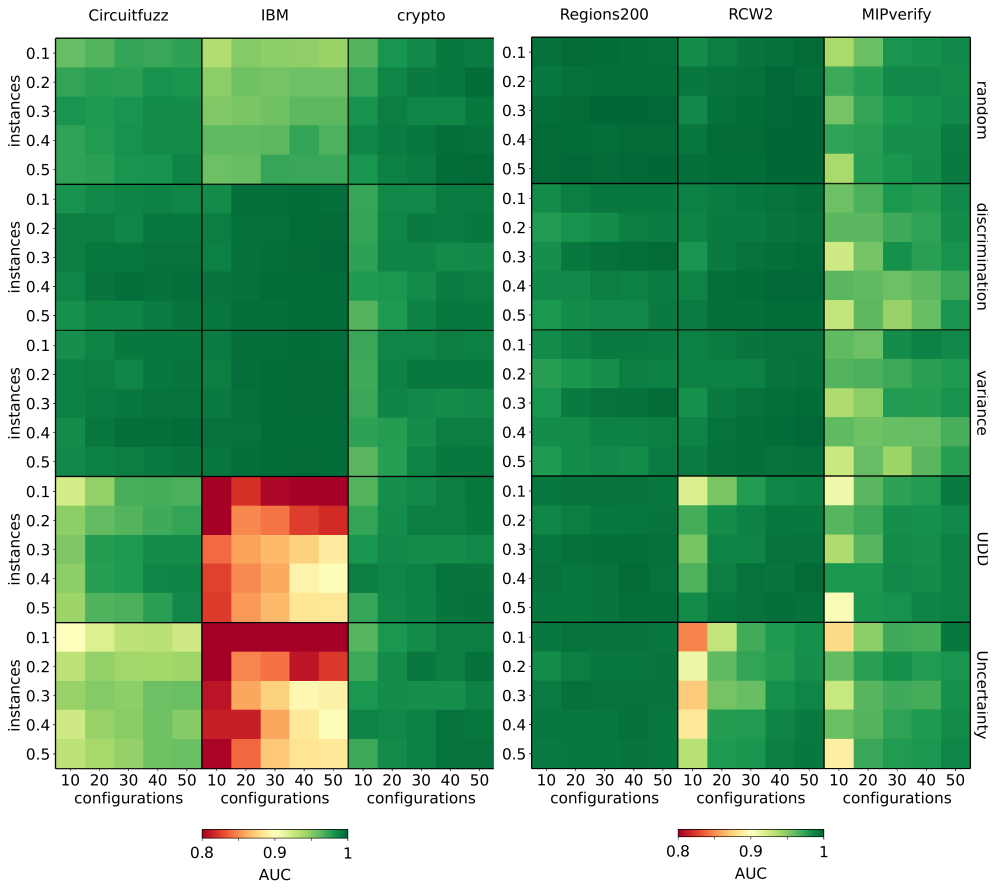
(a) Kissat on IBM, 50% instances, 50 configurations

(b) Cplex on RCW2, 10% instances, 20 configurations

Figure 6.1: Mean accuracy of the Wilcoxon test ($p < 0.05$) on which among ω_{ch} and ω_{inc} performs best *vs* the percentage of time spent on evaluations (100% means that all instances of $\mathcal{I}_{\text{known}}$ have been run)

between the amount of configurations in Ω_{known} and the AUC. This would allow the selection method to become increasingly efficient over the course of the configuration run, thereby avoiding wasted time in the final steps. On the other hand, adding more instances does not seem to improve performance consistently. This is in line with the expectation that our instance sets are built to be homogeneous; thus, adding more instances will be unlikely to improve the model substantially.

Regarding the selection methods, randomly sampling instances performs well, but in most cases, the discrimination and variance approaches are superior. The IBM dataset is unusual in this context, in that the UDD and uncertainty methods perform notably worse than random sampling. This could be explained by the large variation in the running time required to solve the respective problem instances: There are many instances requiring long running times, but also many that are solved within a second or less. This means that selecting the wrong instance can have a dramatic effect on overall running time. This would explain why random sampling does not perform as well on this scenario as on the others, but also why adding more instances in the prior data improves the performance of our selection methods for this scenario.

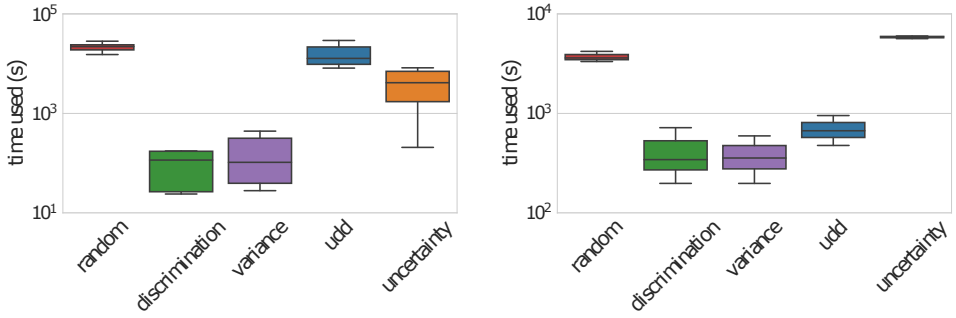


(a) Kissat scenarios **(b)** Cplex scenarios

Figure 6.2: Area under the curve of the mean accuracy of the Wilcoxon test ($p < 0.05$) on which among ω_{ch} and ω_{inc} performs best against the time spent on evaluations.

6.4.2 Compare configurations on unknown instances

To answer the second question – *How does the selection method perform to compare a new configuration to the incumbent on all instances, selecting instances for which we did not collect information throughout the configuration run as seen in phase 2?* – we consider phase 2 (see In phase 2,). We populate $\mathcal{I}_{\text{known}}$ and Ω_{known} with instances and configurations, respectively, selected uniformly at random. The random forest model is trained on all the performance data regarding all pairs of instance and configuration from those two sets. We choose $\omega_{\text{inc}} \in \text{argmin}_{\omega \in \Omega_{\text{known}}} \mathcal{M}(\omega, \mathcal{I}_{\text{selected}})$ and collect all $\omega_{\text{ch}} \in \Omega \setminus \Omega_{\text{known}}$, such that the performance of ω_{inc} and ω_{ch} cannot be distinguished



(a) Kissat – crypto, 30% instances, 20 configurations

(b) cplex – RCW2, 40% instances, 40 configurations

Figure 6.3: Time used (in seconds) before deciding that one configuration is better than the other based on a Wilcoxon test ($\alpha = 0.05$) or reaching a maximum of 10 instance selected

on the instances of $\mathcal{I}_{\text{known}}$ by a Wilcoxon test with a significance level of 0.05. We then apply our selection methods to select up to $n_{\max} = 10$ instances on which we run both configurations until they can be distinguished using the previous test.

For each method and each considered pair of $(\mathcal{I}_{\text{known}}, \Omega_{\text{known}})$, we gather the time used to decide between the two configurations at hand, *i.e.* the sum of the running times of ω_{inc} and ω_{ch} on $\mathcal{I}_{\text{selected}}$. Figure 6.3 shows the running times obtained for two example scenarios.

To evaluate the performance of the selection methods, we computed the median time used to run the instances selected by each of the methods for each prior data, see Table 6.2. The statistical significance of differences in the medians was tested with a permutation test (significance level of 0.05). In most cases, random is outperformed by all other methods, with some exceptions (uncertainty performs worse on RCW2, and random is best on MIPverify). The data shows that discrimination and variance

Table 6.2: Median time in seconds for each method over every tested prior data, with lowest medians boldfaced (statistical significance according to a permutation test with $\alpha = 0.05$).

	ibm	kissat cf	crypto	reg200	cplex RCW2	MIPverify
random	1557	979.7	21243	576.8	4138	**29470**
discrimination	0.086	143.6	419.3	**96.66**	364.7	44390
variance	0.776	**95.16**	**372.2**	109.5	**342.0**	41365
udd	880.9	393.2	13483	379.7	1299	**28845**
uncertainty	**0.033**	330.8	2361.9	152.7	5974	39801

outperform the other methods in almost all cases, with variance providing a speedup ranging from a factor of 5.8 up to 3000 compared to random. We note that the high speedups observed for the IBM dataset are linked to high variance in the running time distribution of the instances, which range from milliseconds to the timeout of 300 seconds.

6.4.3 Discussion

The results shown in this section indicate that the best-performing methods to discriminate between two configurations of the same algorithm within a limited amount of time are the ones based on the running time variance on each instance and on their discrimination power. We notice that both methods inspired by the active learning literature are not performing as well. Whilst we wanted to assess these methods on our problem, this was to be expected, since they were designed with a different goal in mind. Indeed, the field of active learning focuses on improving the accuracy of the model, whereas we only use a model to avoid having to run each configuration on each instance. Improving the accuracy of this model can serve our purpose, but it is not our final goal. We note that the experiments reported here made use of randomly chosen configurations of a given algorithm. As a result, the variation in running times between these configurations is much larger than that expected during an actual configuration run, which focuses on high-performance configurations. While this certainly does not invalidate our results, it implies that we should not expect speedups as large as the ones observed in Table 6.2 when including our methods inside a configurator.

6.5 Evaluation inside the configuration process

As previously shown, applying instance selection and performing a statistical test allows us to spend less time on comparing the performance of two configurations through two expected mechanisms: early stopping of the evaluation of less promising configurations and performance comparisons on less time-consuming instances. In this section, we include the instance selection mechanism inside a model-based configurator in order to evaluate whether the previously observed results can be translated to the performance of the configurator itself. To do so, we expanded the prominent sequential model-based configurator SMAC. However, since SMAC does not include a statistical test, the two aspects of our methods have to be evaluated separately. First, we evaluate SMAC-IS (SMAC with Instance Selection), a version of SMAC3 in which

we added at both phases of the both parts of the instance selection method, namely a *Wilcoxon matched-pairs signed-ranks test* (Conover, 1998) with a significance level $\alpha = 0.05$ to decide if the challenger configuration should be dropped earlier, and an instance selection method to decide on which instance the next run should happen. To compare the performance of SMAC3 to SMAC-IS, we followed the procedure described in Section 6.3 and obtained for each scenario and configurator a distribution of best configurations. The following results are based on those distributions.

6.5.1 Impact of the instance selection methods

To answer our first question, "*Do sophisticated instance selection mechanisms allow us to improve over picking instances uniformly at random?*", we implemented the instance selection mechanisms inside SMAC at the two phases identified earlier and named this new version SMAC-IS. Table 6.3 shows the median performance values of the best configurations distribution. We validate the statistical significance of the differences with a Mann-Whitney U-test with a significance level $\alpha = 0.05$. Moreover, we show in bold methods that perform better than SMAC, our baseline (the performance of SMAC can bee seen in Table 6.4).

Compared to vanilla SMAC, SMAC-IS showed improved behaviour for three of our five scenarios. In particular, the EAX on rue-1000-3000 scenario (Table 6.3a), displays improvements with most instance selection methods; at best, from a default performance of 120.82 seconds, SMAC-IS reaches a median of 65.68 seconds, while SMAC could only reach 92.93 seconds. A similarly impressive improvement was achieved for CPLEX on RCW2 (Table 6.3d), on which, from a default value of 115.95 seconds, SMAC-IS reaches a median of 57.63 seconds, while SMAC could only reach 83.96 seconds.

For CPLEX on REG200 (Table 6.3e), SMAC-IS improves slightly over SMAC, but for CPLEX on CLS (Table 6.3c) it does not, despite being able to find a better configuration than the default. At the other end of the spectrum, for LKH on rue-1000-3000 (Table 6.3b) SMAC-IS returns configurations that perform even worse than the default values in half of the cases.

6.5.2 Impact of the statistical test

To evaluate the impact of the statistical test, we examined the performance of SMAC-IS with random sampling at both phases, which corresponds to vanilla SMAC with a Wilcoxon test to discriminate between the performance of the incumbent and of

Table 6.3: Median performance of SMAC-IS with the selection methods random (rand), variance-based (var) and discrimination-based (disc) at both phases. Boldfaced values are better than those for vanilla SMAC. The lowest median is underlined. All underlined medians are significantly different from others based on a Mann-Whitney U-test ($\alpha = 0.05$).

(a) eax rue-1000-3000

		phase 2		
		rand	var	disc
phase 1	rand	**89.87**	**71.87**	**72.72**
	var	121.69	**87.14**	95.55
	disc	**89.80**	**87.34**	**65.68**

(b) lkh rue-1000-3000

		phase 2		
		rand	var	disc
phase 1	rand	233.13	228.74	229.48
	var	229.39	229.00	243.04
	disc	228.76	185.62	229.19

(c) cplex cls

		phase 2		
		rand	var	disc
phase 1	rand	1.79	1.73	1.67
	var	1.61	1.66	1.68
	disc	1.66	1.69	1.65

(d) cplex RCW2

		phase 2		
		rand	var	disc
phase 1	rand	113.73	113.54	113.94
	var	**57.63**	113.98	114.27
	disc	86.06	114.86	114.48

(e) cplex regions200

		phase 2		
		rand	var	disc
phase 1	rand	3.68	2.95	3.35
	var	3.25	3.77	3.74
	disc	**2.79**	**2.78**	3.05

the challenger configuration at both phases of the configuration. We dub this variant SMAC-W (SMAC with Wilcoxon test). We show the median of those distributions in Table 6.4a. Similarly to the previous results, we validated the statistical significance of the differences using a Mann-Whitney U-test (with $\alpha = 0.05$) and detected statistical significance for all observed differences.

In all except one scenario, the use of a statistical test for early stopping of the comparison has an adverse effect. To further investigate these results, we examined the frequency at which the challenger replaces the incumbent and the number of instances on which the configurations are evaluated. These results are shown in Table 6.4b. We note that the number of accepted incumbents during a run is significantly lower when using the test. Vanilla SMAC accepts the incumbent up to twice as often than SMAC-W for CPLEX on CLS. Moreover, since incumbents get rejected more quickly, the number of instances on which the configurations are evaluated does not increase

Table 6.4: Comparison of SMAC and SMAC-W, respectively, without and with a Wilcoxon test to decide whether a challenger configuration should be kept longer.

(a) Median PAR10 of the best found configurations. The lowest medians are underlined, all are statistically significantly lower according to a Mann-Whitney U-test (with $\alpha = 0.05$).

scenario		default	SMAC	SMAC-W
CPLEX	CLS	1.72	<u>1.31</u>	1.79
	RCW2	115.97	<u>83.96</u>	113.73
	REG200	6.13	<u>2.84</u>	3.68
EAX	rue-1000-3000	120.82	92.93	<u>89.87</u>
LKH		229.22	<u>157.83</u>	233.13

(b) Mean number of changes in incumbent and number of instances in $\mathcal{I}_{\text{known}}$ at the end of the configuration procedure

scenario		changes		instances	
		SMAC-W	SMAC	SMAC-W	SMAC
CPLEX	CLS	3.0	7.1	50	50
	RCW2	3.1	4.2	495	495
	REG200	4.5	5.6	816	823
EAX	rue-1000-3000	4.9	7.6	332	294
LKH		3.1	3.5	432	685

as quickly in SMAC-W as in SMAC. We can expect that running on a smaller number of instances prevents the configurator from seeing the full range of instances on which the algorithm should perform well, leading to overfitting. This is especially evident for the LKH scenario, on which the expected performance of SMAC-W is worse than the default on the test set. We also noticed that the only case in which the number of instances seen during configuration is higher for SMAC-W corresponds to the only scenario in which SMAC-W performs better than SMAC. Based on those results, we can answer our research question and state that in most cases, adding a Wilcoxon test to SMAC hinders its performance.

6.5.3 Discussion

Since the instance selection mechanism did not allow us to improve over SMAC on all scenarios, we looked into the characteristics of each scenario to better understand what might allow instance selection to reach its full potential.

When we look at each selection phase separately, there is no clear trend in terms of

which method performs best at any of those. One expectation was that for scenarios with a low running time, the overhead induced by our methods would hinder the process. Still, SMAC-IS performed slightly better than SMAC on one out of our two scenarios with short running times (CPLEX on CLS and REG200), so this hypothesis does not hold in our experiments. Another expectation was that the homogeneity of the dataset would strongly impact the ability to select the right instances and to decide accurately which configurations to drop. However, the best and worst outcomes were obtained on the same dataset, rue-1000-3000, on which we found 9 clusters of instances when applying a simple mean shift algorithm, which is the highest number among our datasets. Moreover, two seemingly homogeneous datasets, namely CLS and REG200, show very different outcomes. However, the number of clusters does not capture how far those clusters are from each other, which would impact the difficulty of selecting representative instances.

Thus, based on our results, we do not see a clear trend regarding what kinds of scenarios would benefit (or not) from our instance selection mechanism. We note, however, that in two out of five scenarios, we were able to nearly double the improvement obtained by SMAC. This improvement demonstrates that in some scenarios, selecting the instances on which to run the configurations at hand can significantly improve the performance of a general-purpose algorithm configurator.

6.6 Conclusion

Inspired by the success of instance selection when comparing algorithms (see Chapter 5), we adapted four methods from several fields (Matricon et al., 2021; Gu et al., 2014) that could be applied to select instances in the context of automated algorithm configuration (AAC). We identified two steps of AAC procedures at which the selection mechanism could be applied and designed two sets of experiments to assess the performance gains that are thus obtainable. In the first, we considered a situation in which the performance of an incumbent configuration on a set of instances is known, and we want to determine whether the challenger configuration, whose performance is unknown, performs better on this set. In the second, two similarly performing configurations have to be evaluated on unknown instances. Our results show that in both cases, there is considerable potential in the use of those methods, in particular the ones based on the variability in running time or on discrimination power.

Based on those encouraging results, we included the two best selection mechanisms identified in the first phase of our study at both identified steps of the configuration

process within the prominent and state-of-the-art SMAC3 configuration system. On half of the considered scenarios, selecting on which instances to run the first and second phase, on top of performing a Wilcoxon test to decide when to stop the comparison between the current incumbent and a challenger configuration, makes it possible to reach better performing configurations within the same configuration budget, sometimes reaching major improvement compared to SMAC and all previously evaluated configurators according to the results shown in Chapter 3. However, we have not yet found a straightforward way to decide which instance selection method to apply or which scenarios have the potential to benefit from them. Moreover, we studied the impact of solely adding the Wilcoxon test and found that, in most scenarios, using the test degrades the configuration process of SMAC. We note that on the scenarios we have studied, use of the test lowers the number of accepted challengers, likely discarding well-performing configurations by mistake, and tends to slow down the addition of new instances to the pool of instances on which configurations are evaluated. This second point could potentially lead to a form of over-fitting. Those observations confirm that the selection mechanism is responsible for the observed improvements.

This work opens the door to a more principled approach for deciding on which instances the configurations should be evaluated. While more research is needed to determine which specific method to apply in practice, selecting instances during automated algorithm configuration shows great potential.