Journal of Artificial Intelligence Research 64 (2019) 861-893 Submitted 10/18; published 04/19

Pitfalls and Best Practices in Algorithm Configuration

Katharina Eggensperger EGGENSPK@CS.UNI-FREIBURG.DE
Marius Lindauer LINDAUER@CS.UNI-FREIBURG.DE
Frank Hutter FH@CS.UNI-FREIBURG.DE
Institut für Informatik, Albert-Ludwigs-Universität Freiburg,
Georges-Köhler-Allee 74, 79110 Freiburg, Germany

Abstract

Good parameter settings are crucial to achieve high performance in many areas of artificial intelligence (AI), such as propositional satisfiability solving, AI planning, scheduling, and machine learning (in particular deep learning). Automated algorithm configuration methods have recently received much attention in the AI community since they replace tedious, irreproducible and error-prone manual parameter tuning and can lead to new state-of-the-art performance. However, practical applications of algorithm configuration are prone to several (often subtle) pitfalls in the experimental design that can render the procedure ineffective. We identify several common issues and propose best practices for avoiding them. As one possibility for automatically handling as many of these as possible, we also propose a tool called `GenericWrapper4AC`.

1. Introduction

To obtain peak performance of an algorithm, it is often necessary to tune its parameters. The AI community has recently developed automated methods for the resulting *algorithm configuration (AC)* problem to replace tedious, irreproducible and error-prone manual parameter tuning. Some example applications, for which automated AC procedures led to new state-of-the-art performance, include satisfiability solving (Hutter, Babić, Hoos, & Hu, 2007a; Hutter et al., 2017), maximum satisfiability (Ansótegui, Gabàs, Malitsky, & Sellmann, 2016), scheduling (Chiarandini, Fawcett, & Hoos, 2008), mixed integer programming (Hutter, Hoos, & Leyton-Brown, 2010a; López-Ibáñez & Stützle, 2014), evolutionary algorithms (Bezerra, López-Ibáñez, & Stützle, 2016), answer set solving (Gebser et al., 2011), AI planning (Vallati, Fawcett, Gerevini, Hoos, & Saetti, 2013) and machine learning (Thornton, Hutter, Hoos, & Leyton-Brown, 2013; Feurer, Springenberg, & Hutter, 2015).

Although the usability of AC systems improved over the years (e.g., *SpySMAC*, Falkner, Lindauer, & Hutter, 2015), we still often observe fundamental issues in the design and execution of experiments with algorithm configuration methods by both experts and new users. The goals of this work are therefore to:

- highlight the many pitfalls we have encountered in AC experiments (run by ourselves and others);
- present best practices to avoid most of these pitfalls; and

- propose a unified interface between an AC system and the algorithm it optimizes (the so-called target algorithm) that directly implements best practices related to properly measuring the target algorithm's performance with different parameter settings.

Providing recommendations and best practices on how to empirically evaluate algorithms and avoid pitfalls is a topic of interest cutting across all of artificial intelligence, including, e.g., evolutionary optimization (Weise, Chiong, & Tang, 2012), algorithms for NP-complete problems (Gent et al., 1997), and reinforcement learning (Henderson et al., 2018) to mention only a few. Running and comparing implementations of algorithms is the most commonly used approach to understand the behaviour of the underlying method (McGeoch, 1987). There is a rich literature on how to best conduct such empirical studies (Hooker, 1995; Gent et al., 1997; Howe & Dahlman, 2002; McGeoch, 2002, 2012), and for some journals abiding by such guidelines is even mandatory in order to publish research (Dorigo, 2016; Laguna, 2017). Research in AC depends even more on proper empirical methodology than the rest of artificial intelligence, since AC systems need to *automatically* evaluate the empirical performance of different algorithm variants in their inner loop in order to find configurations with better performance. Nevertheless, many of the underlying characteristics of empirical evaluations still remain the same as for other domains, and our guidelines thus share many characteristics with existing guidelines and extend them to the setting faced in AC.

The structure of this work is as follows. First, we provide a brief overview of AC, including some guidelines for new users, such as why and when to use AC, and how to set up effective AC experiments (Section 2). Afterwards, we describe common pitfalls in using AC systems and recommendations on how to avoid them. We first discuss pitfalls concerning the interface between AC systems and target algorithms (Section 3), followed by pitfalls regarding over-tuning (Section 4). Throughout, we illustrate pitfalls by AC experiments on propositional satisfiability solvers (Biere, Heule, van Maaren, & Walsh, 2009) as a prototypical AC example, but insights directly transfer to other AC problems.[1] From our own experiences, we provide further general recommendations for effective configuration in Section 5. We end by presenting a package to provide an interface between AC systems and target algorithms that aims to improve the reliability, reproducibility and robustness of AC experiments (Section 6).

2. Background: Algorithm Configuration

The algorithm configuration problem can be briefly described as follows: given an algorithm $\mathcal{A}$ to be optimized (the so-called *target algorithm*) with parameter configuration space $\mathbf{\Theta}$, a set of instances Π, and a cost metric $c : \mathbf{\Theta} \times \Pi \rightarrow \mathbb{R}$, find a configuration $\theta^* \in \mathbf{\Theta}$ that minimizes the cost metric c across the instances in Π:

$$\theta^* \in \underset{\theta \in \mathbf{\Theta}}{\arg\min} \sum_{\pi \in \Pi} c(\theta, \pi). \tag{1}$$

[1] For these pitfalls, we do not distinguish between decision and optimization problems as the application domain of AC. Although configurators usually take into account whether and how the metric to be optimized relates to runtime, all presented pitfalls can happen in both types of application domain.

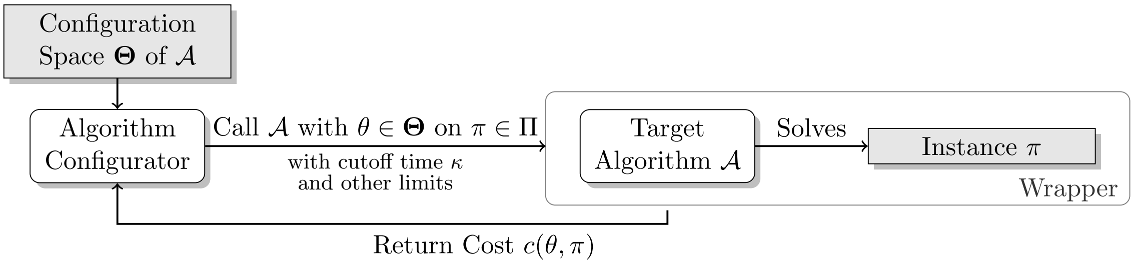

Configuration Space $\mathbf{\Theta}$ of $\mathcal{A}$
Algorithm Configurator
Call $\mathcal{A}$ with $\theta \in \mathbf{\Theta}$ on $\pi \in \Pi$
with cutoff time κ and other limits
Target Algorithm $\mathcal{A}$
Solves
Instance π
Wrapper
Return Cost $c(\theta, \pi)$

Figure 1: Workflow of Algorithm Configuration

A concrete example for this algorithm configuration problem would be to find a parameter setting $\theta \in \mathbf{\Theta}$ of a solver $\mathcal{A}$ for the propositional satisfiability problem (SAT) (such as *glucose*, Audemard & Simon, 2009 or *lingeling*, Biere, 2013) on a set of CNF instances Π (e.g., SAT-encoded hardware or software verification instances) that minimizes $\mathcal{A}$'s average runtime c. Another example would be to find a hyperparameter setting for a machine learning algorithm that minimizes its error c on a given dataset (Snoek, Larochelle, & Adams, 2012; Feurer et al., 2015); in this latter example, c would be validation error, either measured via k-fold inner cross-validation (giving rise to k instances for algorithm configuration) or a single validation set (in which case there is just a single instance for algorithm configuration).

The general workflow of a sequential algorithm configuration procedure (short: *configurator*) is shown in Figure 1. In each step, the configurator picks a configuration $\theta \in \mathbf{\Theta}$ and an instance $\pi \in \Pi$, triggers a run of algorithm $\mathcal{A}$ with configuration θ on instance π with a maximal runtime cutoff κ (and other resource limitations that apply, such as a memory limit), and measures the resulting cost $c(\theta, \pi)$. As detailed in Section 6, this step is usually mediated by a target-algorithm specific *wrapper*. The configurator uses this collected data about the target algorithm's performance to find a well-performing configuration, typically operating as an anytime procedure until its configuration budget is exhausted (e.g., a maximal number of target algorithm calls or a time budget)[2]; when terminated, it returns its current incumbent, i.e., the best found configuration so far.

2.1 Why and When to Consider AC?

Algorithm configuration should always be considered if (i) the empirical performance of an algorithm is relevant and (ii) the algorithm has performance-relevant parameters. This is quite obvious for most empirical studies showing that a new algorithm $\mathcal{A}$ establishes a new state-of-the-art performance on benchmark problem X. However, in this setting it is also important to tune the parameters of all algorithms to compare against — without this, a comparison would not be fair because one of the algorithms may only perform best because

[2] Alternatively, the termination criterion could be defined as stopping when no (or only little) further improvement is expected. Although this is a common choice for some other anytime algorithms, such as gradient descent, we often observe that AC trajectories are step functions with long periods of time between finding improving configurations, complicating the prediction of whether improvements will still happen. For these reasons and to enable an efficient use of resources, we chose to treat the budgets as discussed in the text.

its parameters were tuned with the most effort (Hooker, 1995). Indeed, as shown several times in the AC literature, optimized configurations often perform much better than default ones; in some cases, the default configuration may even be worse than one drawn uniformly at random (e.g., see Figure 6a).

There are several other advantages of AC compared to manual parameter tuning (cf. López-Ibáñez, Dubois-Lacoste, Caceres, Birattari, & Stützle, 2016), including:

Reproducibility Automated algorithm configuration is often more reproducible than doing manual parameter tuning. Manual parameter tuning strongly depends on the experience and intuition of an expert for the algorithm at hand and/or for the given instance set. This manual procedure can often not be reproduced by other users. If algorithm developers also make their configuration spaces available (e.g., as the authors of *Lingeling*, Biere, 2014, and *Clasp*, Gebser, Kaufmann, & Schaub, 2012, do), reproducing the performance of an algorithm using AC is feasible.

Less human-time Assuming that a reasonable configuration space is known, applying algorithm configuration is often much more efficient than manual parameter tuning. While ceding this tedious task to algorithmic approaches can come at the cost of requiring more computational resources, these tend to be quite cheap compared to paying a human expert and are increasingly widely available.

More thoroughly tested Since humans are impatient by nature (e.g., during the development of algorithms), they often focus on a rather small subset of instances to get feedback fast and to evaluate another configuration. Compared to humans, configurators often evaluate (promising) configurations more thoroughly on more instances.

More configurations evaluated Because of similar reasons as above, humans tend to evaluate far less configurations than most configurators would do.

However, there are also two major limitations of AC, which must be considered:

Homogeneous instances To successfully apply AC, the instances have to be similar enough such that configurations that perform well on subsets of them also tend to perform well on others; we call such instance sets *homogeneous*. If the instances are not homogeneous, it is harder to find a configuration that performs well on average; it is even possible that a configurator returns a configuration θ that performs worse than the default one (although θ may appear to perform better based on the instances the configurator could consider within its limited budget). Unfortunately, so far, none of the existing AC tools implement an automatic check whether the given instances are sufficiently homogeneous. For heterogeneous instance sets, portfolio approaches (Xu, Hutter, Hoos, & Leyton-Brown, 2008; Kadioglu, Malitsky, Sabharwal, Samulowitz, & Sellmann, 2011; Malitsky, Sabharwal, Samulowitz, & Sellmann, 2012; Lindauer, Hoos, Hutter, & Schaub, 2015) or instance-specific algorithm configuration (Xu, Hoos, & Leyton-Brown, 2010; Kadioglu, Malitsky, Sellmann, & Tierney, 2010) provide alternative solutions.

Specialization From the restriction to homogeneous instances, the second limitation of AC follows: the optimized configurations (returned by a configurator) are always specialized to the instance set and cost metric at hand. It is hard to obtain a robust configuration on a large variety of heterogeneous instances. (In fact, it is not even guaranteed that a single configuration with strong performance on all instances exists.)

2.2 Setting up AC Experiments

In the following, we describe the typical steps to set up and run AC experiments, and provide pointers to the pitfalls and best practices discussed later.

1. Define an instance set of interest, which should be homogeneous (see Section 5.3) and representative of future instances (see Section 5.2);
2. Split your instances into a training and test instances (see Section 5.1); the test instances are later used to safeguard against over-tuning effects (see Section 4.2);
3. Define the ranges of all performance-relevant parameters giving rise to the configuration space (see Sections 5.6 and 5.7);
4. Implement the interface between your algorithm and the configurator; take Pitfalls 1-4 into consideration (Section 3);
5. Choose your preferred configurator (e.g., *ParamILS*, *GGA++*, *irace* or *SMAC*; see Section 2.3)
6. Define the resource limitations your algorithm (cutoff time and memory limit) and the configurator (configuration budget) should respect (see Section 5.4);
7. Define your cost metric to be optimized; if the cost metric is runtime, configurators typically optimize PAR10 as the metric of interest, which is the penalized average runtime (in CPU seconds) counting runs exceeding the cutoff time κ as $10 \cdot \kappa$; furthermore please consider Pitfalls 2 and 3 (see Section 3.2) and recommendations in Section 5.8 for runtime optimization; if the cost metric is related to the quality of the solution, e.g. the error of a model on a dataset, configurators typically minimize validation error.
8. Run the AC experiments on the training instances and obtain the final incumbent—consider to use parallel runs (Section 5.5);
9. Evaluate the default configuration and the optimized configuration on the test instances, to obtain an unbiased estimate of generalization performance on new instances, and to assess over-tuning effects (Section 4);
10. Optionally, use further tools to obtain visualizations and gain more insights from the AC experiments, e.g., CAVE (Biedenkapp, Marben, Lindauer, & Hutter, 2018).

As an exemplary application where AC yields dramatic speedups, we ran *SMAC* to optimize 75 parameters of the configurator *Clasp* (Gebser et al., 2012) to solve N-Rooks (Manthey & Steinke, 2014a) instances. We will return to this scenario in more detail in Subsection 4.2. Here, we used a training set of 484 instances and a test set of 351 instances to evaluate the best found configurations over time. We used a cutoff of 300 seconds, within which the default configuration solves 82% of all training instances. Figure 2 reports results from 16 independent *SMAC* runs, showing that AC using an adequate setup can robustly yield large speedups compared to not tuning the algorithm.

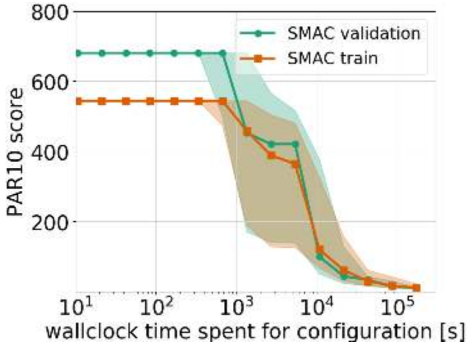


Figure 2: Exemplary application of AC, optimizing 75 parameters of *Clasp* to solve N-Rooks problems. At each time step t, we show the penalized average runtime (PAR10) score on the training set (orange) and test set (green) of the incumbent configuration at time t. I.e., at each time step, we take the best configuration found so far (the one the configurator would return if stopped at that time), ran the algorithm with it on the training and test set and recorded its PAR10 score. We show the median and quartiles of repeating this process 16 times using different random seeds.

2.3 Approaches for Solving the AC problem

For subproblems of the AC problem that deal neither with instances nor with capped and censored runs, there exist several approaches in the fields of parameter tuning, hyperparameter optimization and expensive black-box optimization. Prominent examples include Bayesian optimization (Mockus, Tiesis, & Zilinskas, 1978; Shahriari, Swersky, Wang, Adams, & de Freitas, 2016), sequential parameter optimization (Bartz-Beielstein, Lasarczyk, & Preuss, 2010), evolution strategies (Hansen, 2006), and combinations of several classical search strategies (Ansel et al., 2014).

For solving the full AC problem, there are several configurators. *ParamILS* (Hutter, Hoos, Leyton-Brown, & Stützle, 2009) uses local search in the configuration space, employing a racing strategy to decide which of two configurations performs better without running both of them on all instances. Recently, Cáceres and Stützle (2017) also proposed to use variable neighborhood search instead of the iterated local search used in *ParamILS*. *irace* (López-Ibáñez et al., 2016) uses iterative races via F-race (Birattari, Stützle, Paquete, & Varrentrapp, 2002) on a set of sampled configurations to determine the best one. *SMAC* (Hutter, Hoos, & Leyton-Brown, 2011) and its distributed version *dSMAC* (Hutter, Hoos, & Leyton-Brown, 2012) use probabilistic models of algorithm performance, so-called empirical performance models (Hutter, Xu, Hoos, & Leyton-Brown, 2014b), to guide the search for good configurations by means of an extension of Bayesian Optimization (Brochu, Cora, & de Freitas, 2010). *GGA* (Ansótegui, Sellmann, & Tierney, 2009) represents parameters as genes and uses a genetic algorithm with a competitive and a non-competitive gender; its newest version *GGA++* (Ansótegui, Malitsky, Sellmann, & Tierney, 2015) also uses an empirical performance model for guidance. For a more detailed description of these algorithms, we refer the interested reader to the original papers or to the report of the Configurable SAT Solver Challenge (Hutter et al., 2017).

If the cost metric c is runtime using PAR10 scores, several configurators use an adaptive capping strategy (Hutter et al., 2009) to terminate slow algorithm runs prematurely to save time.[3] For example, if the maximal cutoff time used at test time is $\kappa_{max} = 5000$ seconds and the best configuration known so far solves each instance in 10 seconds, we can save dramatically by cutting off slow algorithm runs after $\kappa > 10$ seconds instead of running all the way to κ_{max}. Since κ is adapted dynamically, each target algorithm run can be issued with a different one.

2.4 The Role of the Target Algorithm Wrapper

As depicted in Figure 1, configurators execute the target algorithm with configurations $\theta \in \mathbf{\Theta}$ on instances $\pi \in \Pi$ and measure the resulting cost $c(\theta, \pi)$. To be generally applicable, configurators specify an interface through which they evaluate the cost $c(\theta, \pi)$ of arbitrary algorithms to be optimized. For a new algorithm $\mathcal{A}$, users need to implement this interface to actually execute $\mathcal{A}$ with the desired configuration θ on the desired instance π and measure the desired cost metric $c(\theta, \pi)$ (e.g. runtime required to solve a SAT instance or validation error of a machine learning model).

In order to avoid having to change the algorithm to be optimized, this interface is usually implemented by a *wrapper*.[4] In the simplest case, the input to the wrapper is just a parameter configuration θ, but in general AC it also includes an instance π, and it can also include a random seed and computational resource limits, such as a runtime cutoff κ. Given these inputs, the wrapper executes the target algorithm with configuration θ on instance π, and measures and returns the desired cost metric $c(\theta, \pi)$.

3. Pitfalls and Best Practices Concerning Algorithm Execution

In this and the next section, we describe common pitfalls in algorithm configuration and illustrate their consequences on existing benchmarks from the algorithm configuration library *AClib* (Hutter et al., 2014a)[5]. Based on the insights we acquired in thousands of algorithm configuration experiments over the years, we propose best practices to avoid these pitfalls.

Throughout, we will use the state-of-the-art configurator *SMAC* (Hutter et al., 2011) as an example, typically optimizing PAR10. Where not specified otherwise, we ran all experiments on the University of Freiburg's META cluster, each of whose nodes shares 64 GB of RAM among two Intel Xeon E5-2650v2 8-core CPUs with 20 MB L3 cache and runs Ubuntu 14.04 LTS 64 bit.[6]

[3] As a side note, we remark that for model-based methods the internal model needs to handle dynamic timeouts arising from adaptive capping and PAR10 scores for guiding the search are based on predictions of that model. Furthermore, evaluations of incumbents for validation purposes are done purely with a fixed timeout κ_{max}, making PAR10 values comparable across configurators.

[4] An alternative to a general wrapper would be a programming language-specific reliable interface for the communication between configurator and target algorithm (Hoos, 2012), which would make it easier for users to apply algorithm configuration to new target algorithms. However, the design of such an interface would also need to consider the pitfalls identified in this paper.

[5] See `www.aclib.net`

[6] Data and scripts for the experiments in this paper are available at `http://www.automl.org/best-practices-in-algorithm-configuration/`.

3.1 *Pitfall 1: Trusting Your Target Algorithm*

Many state-of-the-art algorithms have been exhaustively benchmarked and tested with their default parameter configuration. However, since the configuration space of many algorithms is very large, we frequently observed hidden bugs triggered only by rarely-used combinations of parameter values. For example, Hutter et al. (2010a) reported finding bugs in mixed integer programming solvers and Manthey and Lindauer (2016) bugs in SAT solvers. Due to the size of the associated configuration spaces (e.g., 214 parameters and a discretized space of 10^{86} configurations in the state-of-the-art SAT solver *Riss*, Manthey, 2014b), exhaustive checks are infeasible in practice.

Over the years, the types of bugs we have experienced even in commercial solvers (that are the result of dozens of person-years of development time) include:

- Segmentation faults, Null pointer exceptions, and other unsuccessful algorithm terminations;
- Wrong results (e.g., claiming a satisfiable SAT instance to be unsatisfiable);
- Not respecting a specified runtime cutoff that is passed as an input;
- Not respecting a specified memory limit that is passed as an input;
- Rounding down runtime cutoffs to the next integer (even if that integer is zero); and
- Returning faulty runtime measurements (even negative ones!)

Effects The various issues above have a multitude of negative effects, from obvious to subtle. If the algorithm run does not respect its resource limits this can lead to congested compute nodes (see Pitfall 3) and to configurator runs that are stuck waiting for an endless algorithm run to finish. Wrongly reported runtimes (e.g., close to negative infinity in one example) can lead to endless configuration runs when trusted. Rounding down cutoff times can let configurators miss the best configuration (e.g., when they use adaptive capping to cap runtimes at the best observed runtime for an instance – if that runtime is below one second then each new configuration will fail on the instance due to using a cutoff of zero seconds).

Algorithm crashes can be fairly benign when they are noticed and counted with the highest possible cost, but they can be catastrophic when not recognized as crashes: e.g., when blindly minimizing an algorithm's runtime the configurator will typically simply find a configuration that crashes quickly. While this can be exploited to quickly find bugs (Hutter et al., 2010a; Manthey & Lindauer, 2016), obtaining faulty configurations is typically the worst possible result of using algorithm configuration in practice. Bugs that lead to wrong results tend to be discovered by configurators when optimizing for runtime, since (at least for $\mathcal{NP}$-hard problems) we found that such bugs often allow algorithms to find shortcuts and thus shorten runtimes. Therefore, blindly minimizing runtime without solution checking often yields faulty configurations.

Detailed Example In 2012, we used algorithm configuration to minimize the runtime of the state-of-the-art solver *glucose* (Audemard & Simon, 2009). We quickly found a parameter configuration that appeared to yield new state-of-the-art performance on the industrial instances of the SAT Challenge 2012[7]; however, checking this configuration with

[7] `http://baldur.iti.kit.edu/SAT-Challenge-2012/`

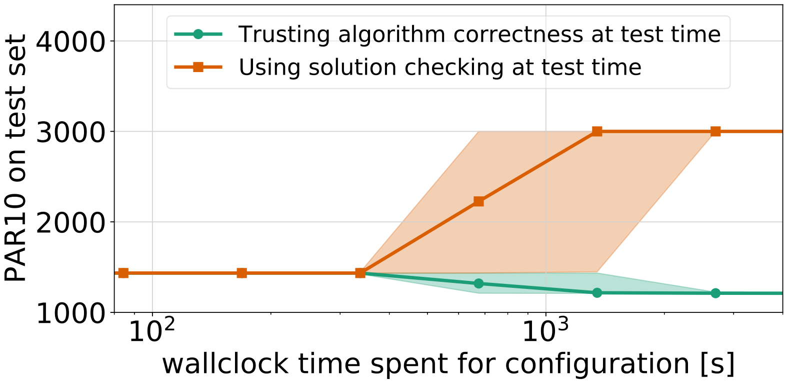


Figure 3: Difference in test set performance as judged when trusting the target algorithm (green) and using external solution checking (orange). We plot the penalized average runtime (PAR10) scores of *Glucose* v2.1 on the industrial instances from the SAT Challenge 2012, as a function of time spent for configuration, when the configuration process trusted *Glucose* v2.1 to be correct. We ran 12 *SMAC* runs and at each time step show the median and quartiles of their incumbents' scores. The green curve computes these scores trusting the solutions *Glucose* returns, while the orange curve penalizes faulty configurations with the worst value of 3000 (where faulty configurations are those that yield at least one wrong result on the test instances; such configurations would, e.g., be disqualified in the SAT competition). We emphasize that both curves are based on exactly the same set of 12 SMAC runs (which were broken in that they trusted *Glucose* rather than applying solution checking) and only differ in their validation.

the authors of *Glucose* revealed that it led to a bug which made *Glucose* falsely report some satisfiable instances as unsatisfiable.[8]

In Figure 3 we reconstruct this behaviour. We ran *SMAC* on *Glucose* v2.1 and evaluated configurations found over time when trusting *Glucose*'s correctness at configuration time: The green curve shows *Glucose*'s (buggy) outputs on the test instances, whereas the orange curve scored each configuration using solution checking, and returning the worst possible score for configurations that returned a wrong solution. After 300 to 3000 seconds, *SMAC* found configurations that seemed better when trusting *Glucose*'s outputs, but that actually sometimes returned wrong solutions, resulting in the true score (orange curve) going up (getting worse) to the worst possible PAR10 score.

Best Practice Most of the issues above can be avoided by wrapping target algorithm runs with a reliable piece of code that limits their resources and checks whether they yield correct results. Cast differently, the job of this wrapper is to actually measure the cost function $c(\theta, \pi)$ of interest, which should intuitively heavily penalize any sort of crashes or bugs that lead to wrong results.

If enough computational time is available, we recommend to first run systems such as *SpyBug* (Manthey & Lindauer, 2016) to find bugs in the configuration space, and to

[8] The bug in *Glucose* version 2.1 was fixed after we reported it to the developers, and we are not aware of any bugs in the newest *Glucose* version 4.1.

either fix them or to exclude the faulty part of the configuration space from consideration. Regardless of whether this is done or not, since it is infeasible to perfectly check the entire configuration space, we always recommend to check the returned solution of the target algorithms during the configuration process. For example, for SAT instances, our example wrapper exploits the standard SAT checker tool routinely used in the SAT competitions to verify the correctness of runs. For solvers that output unsatisfiability proofs, there are also effective tools for checking these proofs (Heule, Hunt, & Wetzler, 2014).

3.2 *Pitfall 2: Not Terminating Target Algorithm Runs Properly*

Given the undecidability of the halting problem, target algorithm runs need to be limited by some kind of runtime cutoff κ_{max} to prevent poor configurations from running forever. In many AI communities, it is a common practice to set a runtime cutoff as part of the cost metric and measure the number of timeouts with that cutoff (e.g., $\kappa_{max} = 5000$ seconds in the SAT race series). In algorithm configuration, the ability to prematurely cut off unsuccessful runs also enables adaptive capping (see Section 2). Therefore, it is essential that target algorithm runs respect their cutoff. This pitfall is related to Pitfall 1 as the user also needs to trust the target algorithm to work appropriately. While for Pitfall 1 we focus on the returned solution, here we draw attention to the resource limitations.

Effect Consequences of target algorithm runs not respecting their cutoffs can include:

1. If the target algorithm always uses the maximal cutoff κ_{max} and ignores an adapted cutoff $\kappa < \kappa_{max}$, the configuration process is slowed down since the benefits of adaptive capping are given up;

2. If the target algorithm completely ignores the cutoff, the configuration process may stall since the configurator waits for a slow target algorithm to terminate (which, in the worst case, may never happen);

3. If a wrapper is used that fails to terminate the actual algorithm run but nevertheless returns the control flow to the configurator after the cutoff time κ, then the slow runs executed by the configurator will continue to run in parallel and overload the machine, messing up the cost computation (e.g., wallclock time).

Example The latter (quite subtle) issue actually happened in a recent publication that compared *GGA++* and *SMAC*, in which a wrapper bug caused *SMAC* to perform poorly (Ansótegui et al., 2015). The authors wrote a wrapper for *SMAC* that tried to terminate its target algorithm runs (here: *Glucose* or *Lingeling*) after the specified cutoff time κ by sending a KILL signal, but since it ran the target algorithm through a shell (using `subprocess.Popen(cmd, shell=True)` in Python) the KILL signal only terminated the shell process but not the actual target algorithm (which continued uninterrupted until successful, sometimes for days). When attempting to reproduce the paper's experiments with the original wrapper kindly provided by the authors, over time more and more target algorithms were spawned without being terminated, causing our 16-core machine to slow

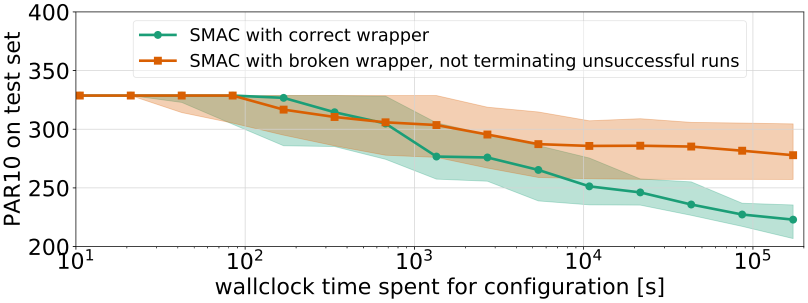


Figure 4: Effect of a broken wrapper that does not terminate target algorithm runs properly. We show PAR10 test set performance for optimizing *Cryptominisat* with *SMAC* on *Circuit Fuzz* instances, when using a correct and a broken wrapper during configuration, respectively. We show median test performance (measured using a correct wrapper) with quartiles across 80 runs of *SMAC*. Not terminating target algorithm runs properly eventually slowed down the machine affecting runtime measurements.

down and eventually become unreachable. This issue demonstrates that *SMAC* heavily relies on a robust wrapper that automatically terminates its target algorithm runs properly.[9]

To illustrate this issue in isolation, we compared *SMAC* using a working wrapper and a broken version of it that returns the control flow to the configurator when the runtime cutoff is reached, without terminating the target algorithm run process. Figure 4 shows the performance achieved when *SMAC* is run with either wrapper to configure *Cryptominisat* (Soos, 2014) for penalized average runtime (PAR10) to solve *Circuit Fuzz* instances (Brummayer, Lonsing, & Biere, 2012) as used in the CSSC 2014 (Hutter et al., 2017). We executed 80 *SMAC* runs for each wrapper, with 16 independent parallel runs each on five 16-core machines. Both *SMAC* versions performed equally well until too many target algorithm processes remained on the machines and prevented *SMAC* from progressing further. Only on one of the five machines that ran *SMAC* with the broken wrapper, the runs terminated after the specified wallclock-limit of 2 days; after an additional day, three of the remaining machines were still frozen caused by overload and the fourth could not be reached at all.

Best Practice To avoid this pitfall, we recommend to use some well-tested, external piece of code to reliably control and terminate target algorithm runs.

3.3 *Pitfall 3: Slow File System*

Related to Pitfall 2, another way to ruin runtime measurements by slowing down a machine is to overload the used file system. Each target algorithm run typically has to read the given

[9] In contrast to *SMAC*, *GGA++* does not require a wrapper; in the experiments by Ansótegui et al. (2015), *GGA++* directly sent its KILL signal to the target algorithm and therefore did not suffer from the same problem *SMAC* suffered from, which confounded the paper's comparison between *GGA++* and *SMAC*. Additionally, there was also a simple typo in the authors' wrapper for *SMAC* in parsing the target algorithm's output (here: *Glucose*) that caused it to count all successful runs on unsatisfiable instances as timeouts. Receiving wrong results for all unsatisfiable instances (about half the instance set) severely affected *SMAC*'s trajectory; this issue was only present in the wrapper for *SMAC* (and therefore did not affect *GGA++*), confounding the comparison between *GGA++* and *SMAC* further.

problem instance and writes some log files; thus, executing many algorithm configuration runs in parallel can stress the file system.

Effect Slowdowns caused by an overloaded file system can have a severe impact on runtime measurements; in particular this is problematic because most algorithm configurators measure their own configuration budget as wallclock time. Furthermore, these problems are often not immediately recognizable (because everything runs fine when tested at small scale) and sometimes only affect parts of a large set of experiments (as the overload might only happen for a short time).

Example 1 Over the years, we have experienced file system issues on a variety of clusters with shared file systems when target algorithm runs were allowed to write to the shared network file system. When executing hundreds (or on one cluster, even thousands) of algorithm configuration runs in parallel, this stressed the file system to the point where the system became very slow for all users and we measured 100-fold overheads in individual target algorithm evaluations. Writing target algorithm outputs to the local file system fixed these issues.

Example 2 Distributing configuration runs across multiple nodes in a compute cluster (e.g., in *GGA*, *irace*, or *dSMAC*) can be error-prone if the configurators communicate via the file system. In particular, we experienced issues with several shared network file systems with asynchronous I/O; e.g., on one compute node a file was written, but that file was not immediately accessible (or still empty) on other compute nodes. Often a second read access resolved the problem, but this solution can be brittle; a change of parallelization strategy may in that case yield more robust results.

Example 3 Even when writing target algorithm output to the local file system, we once experienced 200-fold overheads in target algorithm runs (invocations of sub-second target algorithm runs hanging for minutes) due to a subtle combination of issues when performing hundreds of algorithm configuration experiments in parallel. On the Orcinus cluster (part of Compute Canada's Westgrid cluster), which uses a Lustre file system, we had made our algorithm configuration benchmarks read-only to prevent accidental corruption. While that first seemed like a good idea, it disallowed our Python wrapper to create `.pyc` bytecode files and forced it to recompile at every invocation, which in turn triggered a stats call (similar to `ls` on the Linux command line) for each run. Stats calls are known to be slow on the Lustre file system, and executing them for each sub-second target algorithm run on hundreds of compute nodes in parallel led to extreme file system slowdowns. After testing many other possible reasons for the slowdowns, removing the read-only condition immediately fixed all issues.

Best Practice Issues with shared file systems on compute clusters can have subtle reasons and sometimes require close investigation (as in our Example 3). Nevertheless, most issues can be avoided by using the faster local file system (typically `/tmp/`, or even better, a temporary job-specific subdirectory thereof[10]) for all temporary files, and by measuring CPU time instead of wallclock time (at least for sequential algorithms).

[10] We note that on some modern Linux distributions, `/tmp/` can be a RAM disk and therefore may use resources allotted to the algorithm runs; in general, we recommend to make the choice about a fast temporary directory specific to the compute cluster used.

3.4 *Pitfall 4: Handling Target Algorithm Runs Differently*

The required functionalities of the target algorithm wrapper differ slightly for different configurators. For example, *SMAC* and *ParamILS* trust the wrapper to terminate target algorithms, but *GGA* sends a KILL signal on its own (see also Pitfall 2). Therefore, sometimes configurators are compared by using different target algorithm calls and measurements. However, if this is not done properly, it can lead to a biased comparison between configurators.

Effect Calling the target algorithm differently for different configurators can lead to different behaviors of the target algorithm and hence, to different returned performance values for the same input. If the configurators receive different performance measurements, they will optimize different objective functions and their runs become incomparable.

Example During the early development of *SMAC* (before any publication), we used the same wrappers for *ParamILS* and *SMAC* but an absolute path to the problem instance for one and a relative path for the other. Even this tiny difference lead to reproducible differences of runtime measurements of up to 20% when optimizing an algorithm implemented in UBCSAT 1.1.0 (Tompkins & Hoos, 2005). The reason was that that version of UBCSAT stored its callstring in its heap space such that the number of characters in the instance name affected data locality and therefore the number of cache misses and the runtime (whereas the number of search steps stayed the same).[11] This subtle issue demonstrates the importance of using the same wrapper for all configurators being compared such that exactly the same target algorithm calls are used.

Best Practice We recommend to use a single wrapper when comparing configurators against each other, in order to guarantee that all configurators optimize the same objective. For studies comparing configurators, it is also paramount to use tried-and-tested publicly available benchmark scenarios (lowering the risk of typos, etc; see also Footnote 9); our algorithm configuration benchmark library AClib (Hutter et al., 2014a) provides a very broad collection of such benchmarks.

4. Pitfalls and Best Practices Concerning Over-Tuning

A common issue in applying algorithm configuration is the over-tuning effect (Birattari, 2004; Hutter, Hoos, & Stützle, 2007b; Birattari & Kacprzyk, 2009; Hutter et al., 2009) Over-tuning is very related to the concept of over-fitting in machine learning and denotes the phenomenon of finding parameter configurations that yield strong performance for the training task but do not generalize to test tasks. We emphasize that over-tuning effects are not necessarily only related to the set of training instances used, but can also include the characteristics of the experimental setup such as the resource limitations and bugs in the solver (see Pitfall 1). To safeguard against over-tuning effects, it is crucial to evaluate generalization performance (typically, using a set of benchmark instances disjoint from the benchmarks used for training). In the following, we discuss three pitfalls related to over-tuning.

[11] This issue is fixed in later versions of UBCSAT.

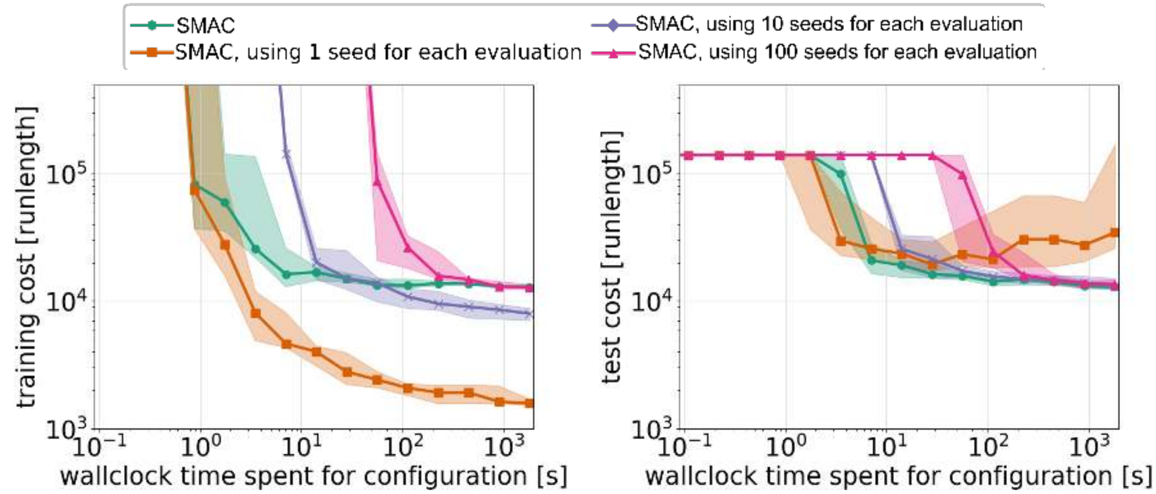


Figure 5: Optimizing *Saps* with *SMAC* on one "quasigroup with holes" instance (QWH) and different numbers of random seeds. Each line in the left plot shows the median and quartiles of the estimated training cost and each line in the right plot shows the median and quartiles of the test cost of *Saps* over time across 10 runs of *SMAC*.

4.1 ***Pitfall 5: Over-tuning to Random Seeds***

Many algorithms are randomized (e.g., SAT solvers or AI planners). However, in many communities, the random seeds of these algorithms are fixed to simulate a deterministic behavior and to ensure reproducibility of benchmark results.

Effect Ignoring the stochasticity of an algorithm in algorithm configuration by fixing the random seed can lead to over-tuning effects to this seed, i.e., finding a configuration that yields good performance with this fixed random seed (or set of seeds) but poor performance when used with other random seeds. The extreme case is not to only fix the random seed, but to tune the random seed, which can lead to an even stronger over-tuning effect.[12]

Example To illustrate over-tuning to a random seed in its purest form, independent of a difference between training and test instances, we optimized the parameters of the local-search SAT solver *Saps* (Hutter, Tompkins, & Hoos, 2002) on a single instance, the only difference between training and test being the set of random seeds used. We used different settings of *SMAC* to handle random seeds: We compared *SMAC* using a fixed set of 1, 10 or 100 random seeds for each target algorithm run and standard *SMAC*, which handled the random seed itself (using a larger number of seeds to evaluate the best configurations).

As a cost metric, we minimized the average number of local search steps (the solver's so-called *runlength*) since this is perfectly reproducible. For the parameter configurations recommended at each step of each *SMAC* run, we measured *SMAC*'s training cost (as the mean across the respective sets of seeds discussed above) as well as its test cost (the mean

[12] We note that, in principle, one could construct situations where fixing or even optimizing the seed could lead to good performance if that seed is used in all future experiments and a large number of instances is available to obtain generalization to other instances. However, we believe that the potential misuse of tuning seeds outweighs any potential benefits.

runlength across 1000 fixed random seeds that were disjoint from the sets of seeds used for configuration) [13].

Figure 5 shows median costs across 10 *SMAC* runs, contrasting training cost (left) and test cost (right). On training, *SMAC*, using 1 seed per evaluation quickly improved and achieved the best training cost on its one random seed, but its performance does not generalize to the test seeds. *SMAC*, using 10 or 100 seeds per evaluation were slower but generalized better, and standard *SMAC* was both fast and generalized best by adaptively handling the number of seeds to run for each configuration.

Best Practice For randomized algorithms, we recommend to tune parameter configurations across different random seeds—most configurators will take care of the required number of random seeds if the corresponding options are used. If a configuration's performance does not even generalize well to new random seeds, we expect it to also not generalize well to new instances. Furthermore, the number of available instances is often restricted, but there are infinitely many random seeds which can be easily sampled. Likewise, when there are only few test instances, at validation time we recommend to perform multiple runs with different random seeds for each test instance.

4.2 *Pitfall 6: Over-tuning to Training Instances*

The most common over-tuning effect is over-tuning to the set of training instances, i.e., finding configurations that perform well on training instances but not on new unseen instances. This can happen if the training instances are not representative for the test instances; in particular this is often an issue if the training instance set is too small or the instances are not homogeneous (Hutter, Hoos, & Leyton-Brown, 2010b; Schneider & Hoos, 2012), i.e., if there exists no single configuration with strong performance for all instances.

Effect In practice, over-tuned configurations that only perform well on a small finite set of instances are of little value, because users are typically interested in configurations that also perform well on new instances. Phrasing this more generally, research insights should also generalize to experiments with similar characteristics.

Example To illustrate this problem, we studied training and test performance of various configurations for three exemplary benchmarks (see Figure 6):

***Clasp* on N-Rooks** We studied the runtime of the solver *Clasp* (Gebser et al., 2012) on N-Rooks instances (Manthey & Steinke, 2014a), a benchmark from the Configurable SAT Solver Challenge (CSSC 2014; Hutter et al., 2017). In this case, the runtimes on the training and test set were almost perfectly linearly correlated, with a Spearman correlation coefficient of 0.99, i.e., the ranking of the configurations on both sets is nearly identical; this is also visualized in Figure 6a. This is a very good case for applying algorithm configuration, and, correspondingly, in the CSSC 2014 algorithm configuration yielded large improvements for this benchmark.

***Lingeling* on mixed SAT** We reconstructed a benchmark from Ansótegui et al. (2015) in which they optimized *Lingeling* (Biere, 2014) on a mixed set of industrial SAT

[13] Note that Hutter et al. (2007b) used the *median* to aggregate across the 1000 seeds, resulting in slightly lower training and test runlengths.

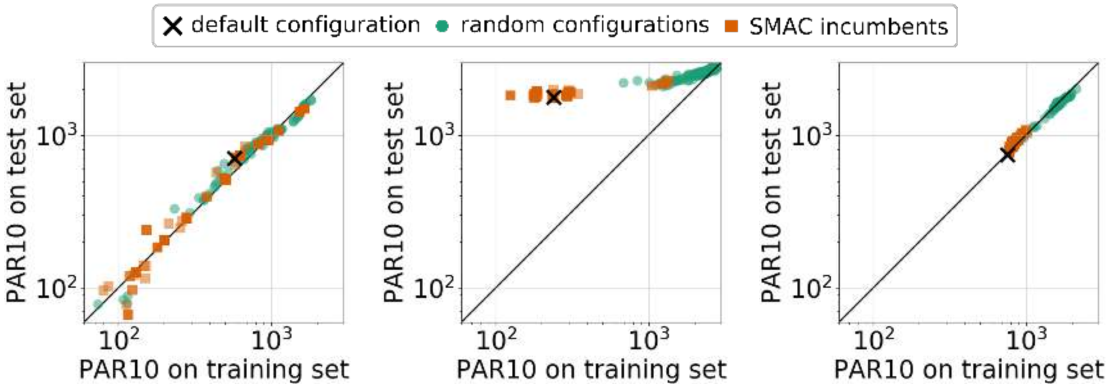
instances. Instead of randomly splitting the data into train and test instances, they first created a training set by removing hard instances (i.e., not solved within the cutoff time by reference solvers) and used these remaining hard instances as test instances. Figure 6b shows that *SMAC* improved the runtime of *Lingeling* on the training set but that these improvements did not generalize to the test instances. In fact, the training and test scores of the optimized configurations (orange squares) are only weakly correlated (Spearman correlation coefficient of 0.15). The benchmark's heterogeneity and the mismatch between training and test set make this benchmark poorly suited for algorithm configuration.

***Clasp* on LABS** Figure 6c shows another benchmark from the CSSC: configuration of *Clasp* on SAT-encoded low autocorrelation binary sequence (LABS) benchmarks (Mugrauer & Balint, 2013). This illustrates a rare worst case for algorithm configuration, in which performance even degrades on the training set, which is possible due to *SMAC*'s (and any other configurator's) racing approach: the configurator already changes the incumbent before all training instances have been evaluated, and if a subset is not representative of the full set this may lead to performance degradation on the full set.

While we have occasionally observed such strong heterogeneity on instances with very heterogeneous sources, it was very surprising to observe this in a case where all instances stemmed from the same instance family. We therefore analyzed this benchmark further (Hutter et al., 2017), showing that twice as many *SMAC* runs with a fivefold larger configuration budget managed to improve training performance slightly. However, that improvement on the training set still did not generalize to the test set due to the benchmark's heterogeneity. (Although visually not apparent from Figure 6c, for this benchmark, the correlation between scores on training and test instances was quite low (0.42) for the 20% best-performing randomly sampled configurations). Again, for such heterogeneous benchmarks we recommend the usage of portfolio approaches.

Best Practice Over-tuning is often not easy to fully rule out by design, since the effect can only be measured by assessing test performance after the configuration process completed (for example by scatter plots, such as in Figure 6). Nevertheless, the following strategies minimize the risk of over-tuning (see also Section 5):

1. The training instances should be representative of the test instances;
2. The training set should be relatively large (typically hundreds to thousands of instances) to increase the chance of being representative;
3. The instance sets should stem from a similar application, use context, etc., increasing the likelihood that they have similar structures which can be exploited with similar solution strategies;
4. If the instance set is heterogeneous, portfolio approaches (Xu et al., 2008; Kadioglu et al., 2011; Malitsky et al., 2012; Lindauer et al., 2015) or instance-specific algorithm configuration (Xu et al., 2010; Kadioglu et al., 2010) should be used.



(a) *Clasp* on N-Rooks (b) *Lingeling* on mixed SAT (c) *Clasp* on LABS

Figure 6: Comparing training and test performance of different configurations to study whether these performances on both sets are correlated. Green dots indicate randomly sampled configurations, the black cross marks the performance of the default configuration of the solver, and orange squares correspond to incumbent configurations of 16 *SMAC* runs.

4.3 *Pitfall 7: Over-tuning to a Particular Machine Type*

In the age of cloud computing and large compute clusters, an obvious idea is to use these remotely-accessible compute resources to benchmark algorithms and configure them. However, in the end, these remote machines are not always the production systems the algorithms are used on in the end. Geschwender, Hutter, Kotthoff, Malitsky, Hoos, and Leyton-Brown (2014) indicated in a preliminary study that it is possible in principle to configure algorithms in the cloud, and that the found configurations perform well on another machine. Unfortunately, recent other experiments showed that this does not hold for all kinds of algorithms – for example, the performance of solvers for SAT (Aigner, Biere, Kirsch, Niemetz, & Preiner, 2013) and mixed integer programming (Lodi & Tramontani, 2014; Koch et al., 2011) can depend strongly on the used machine type (including hardware, operating system and installed software libraries).

Effect Some algorithms are machine-dependent and obtain different results depending on the hardware they run on. Being unaware of this can ruin both, a successful application and a comparison of configuration methods, in two ways: Firstly, when configuring on one system the best found configuration might perform poorly on another system. Secondly, the ranking of the best found configurations of target algorithms on one system might change when rerunning the experiments on a different system.

Example An example for such machine-dependent algorithms are SAT solvers that are often highly optimized against cache misses (Aigner et al., 2013). To study the effect of different machines, we optimized three SAT solvers from the configurable SAT solver challenge (Hutter et al., 2017), namely *Minisat-HACK-999ED* (Oh, 2014), *Clasp* (Gebser et al., 2012) and *Lingeling* (Biere, 2014) on *Circuit Fuzz* instances (Brummayer et al., 2012). As different machine types, we used AWS m4.4xlarge instances with 2.4-GHz Intel Xeon E5-2676 v3 CPUs with 30MB level-3 cache and the META-cluster at the University

	AWS		META-Cluster	
solver	Rank	PAR10	Rank	PAR10
Minisat-HACK-999ED	1	187	2	205
Clasp	2	215	1	193
Lingeling	3	231	3	208

Table 1: Three SAT solvers from the configurable SAT solver challenge on *Circuit Fuzz* instances on two different hardware systems.

of Freiburg with 2.6GHz Intel Xeon E5-2650v2 8-core CPUs with 20 MB L3 cache. On both systems, we ran Ubuntu 14.04 64bit and allowed for a memory limit of 3GB for each solver run. The binaries were statically compiled such that they are not linked against different libraries on the different systems. For each solver we ran 12 independent *SMAC* runs and validated the cost of the best found configuration for each solver on test instances on the same system.

Table 1 lists the ranking and the PAR10 scores of the solvers on each machine (showing the test cost of the configuration performing best on training); we note that the PAR10 scores are only comparable on the same system. In both environments, *Lingeling* ended up on rank 3, but the ranks of *Clasp* and *Minisat-HACK-999ED* differed between the two environments: if the AWS cloud would be our environment for running AC experiments, we would decide for *Minisat-HACK-999ED*, but this would not be the best choice on the META-cluster. We note that, since we picked the best of 12 *SMAC* runs, due to the high variance of extremal statistics, the exact numbers of this experiments might vary in a rerun. Since we did not have enough compute resources on AWS for carrying out multiple runs, to gain additional confidence in our conclusions, we carried out an additional experiment: we validated the configurations found on AWS on the META-cluster and found that in that setting the configured *Minisat-HACK-999ED* performed even worse than *Lingeling* and *Clasp*. Therefore, we conclude that the ranking of configured algorithms depends on the hardware.

Best Practice We note that this pitfall exists only for machine-sensitive algorithms. Therefore, we recommend to investigate whether an algorithm at hand has machine-dependent performance, for example, by validating the performance of various configurations on both the system used for configuration and the production system.

5. Further Recommendations for Effective Configuration

In the following, we describe recommendations for users of algorithm configuration systems to obtain parameter configurations that will perform better in production. Some of these recommendations are rules of thumb, since the involved factors for a successful configuration can be very complex and can change across configuration scenarios. For general empirical algorithmics, McGeoch (2012) recommends further best practices, including design, reports and analysis of computational experiments.

5.1 Training and Test Sets

As discussed before, following standard practice, we strongly recommend to split the available instances into a training and a test set to obtain an unbiased estimate of generalization performance from the test set (Birattari & Kacprzyk, 2009). To obtain trivial parallelization of randomized configuration procedures, we recommend to run n independent configuration runs and use the training set to select the best of the n resulting configurations (Hutter et al., 2012). Only that single chosen configuration should be evaluated on the test set; we explicitly note that we cannot select the configuration that performs best on the test set, because that would amount to peeking at our test data and render performance estimates on the test set biased.

5.2 Representative Instances and Runtime Cutoff

Intuitively, instances for which every parameter configuration times out do not help the configurator to make progress. One strategy can be to remove these from the training set. However, this comes with the risk to bias the training set towards easy instances and should be used with caution. Generally, we therefore recommend to use training instances for the configuration process that are representative of the ones to be solved later. Using training instances from a range of hardness can also often help yield configurations that generalize (Hoos, Kaufmann, Schaub, & Schneider, 2013). If feasible, we recommend to select instances and runtime cutoffs such that roughly 75% or more of the training instances used during configuration can be solved by the initial parameter configuration within the cutoff. We emphasize that – while the configuration protocol may in principle choose to subsample the training instances in arbitrary ways – the test set should never be touched and not pre-evaluated to ensure an unbiased cost estimate of the optimized configurations in the end (see Pitfall 6). To select a good training instance set, Bayless, Tompkins, and Hoos (2014) proposed a way to quantify whether an instance set is a good proxy for another instance set. Furthermore, Styles and Hoos (2015) proposed a splitting strategy of the instances for better scaling to hard instances: They split the instances into a training, validation and test set to use easy instances during configuration for fast progress and select a configuration on the harder validation set such that the configuration will perform well on the hard test set.

5.3 Homogeneous vs Heterogenous Instance Sets

Sometimes configurators are used to obtain well-performing and robust configurations on a heterogeneous instance set. However, we know from algorithm selection (Rice, 1976; Kotthoff, 2014) that often no single configuration exists that performs well for all instances in a heterogeneous set, but a portfolio of configurations is required to obtain good performance (Xu, Hutter, Hoos, & Leyton-Brown, 2011; Kadioglu et al., 2010). Furthermore, the task of algorithm configuration becomes a lot harder if all instances can be solved best with very different configurations. Therefore, we recommend to use algorithm configuration mainly on homogeneous instance sets. Furthermore, the size of the used instance set should be adjusted accordingly to the homogeneity of the instance set: on homogeneous instance sets, 50 instances might suffice for good generalization performance to new instances, but on fairly heterogeneous instance sets, we recommend to use at least 300 or, if possible, more than 1000 instances to obtain a robust parameter configuration.

5.4 Appropriate Configuration Settings

To use configurators, the user has to set the budget available for the configurator. If the configuration budget is too small, the configurator might make little or no progress within it. In contrast, if the configuration budget is too large, we waste a lot of time and computational resources because the configurator might converge long before the budget is used up. A good rule of thumb in our experience is to use a budget that equals at least the expected runtime of the default configuration on 200 to 1000 instances. In practice, an effective configuration budget strongly depends on several factors, including heterogeneity of the instance set (more heterogeneous instance sets require a larger configuration budget) or size of the configuration space (larger configuration spaces require more time to search effectively, Hutter et al., 2017). Finally, if the configurator finds better performing configurations quickly, then the estimate of the total runtime based on the runtime of the default configuration might be too conservative.

5.5 Efficient Use of Parallel Resources

Some configurators (such as *GGA*, *irace* and *dSMAC*) can make use of parallel resources, while others (such as *ParamILS* and *SMAC*) benefit from executing several independent parallel runs[14] (and using the result from the one with the best training set performance; see, e.g., Hutter et al., 2012). In the special case of *GGA*, using more parallel resources can actually improve the adaptive capping mechanism. Given k cores, we therefore recommend to execute one *GGA* run with k cores, but k independent *ParamILS* or *SMAC* runs with one core each. While this protocol was not used in early works[15], it has been used in more recent evaluations (Ansótegui et al., 2015; Hutter et al., 2017).

5.6 Reasonable Configuration Space

Another challenge in using algorithm configuration systems is to find the best configuration space. The user has to decide which parameters to optimize and which ranges to allow. The optimal set of parameters to configure is often not clear and in case of doubt, we recommend to add more parameters to the configuration space and to use generous value ranges.

However, we note that unreasonably large configuration spaces are hard to configure and require substantially larger configuration budgets. For example, the state-of-the-art SAT solver *Lingeling* (Biere, 2013) has more than 300 parameters and most of them have a value range between 0 and 32bit maxint, but most of these parameters are either not really relevant for optimizing *Lingeling*'s runtime or the relevant value ranges are much smaller. Even though *Lingeling* can already substantially benefit from configuration we expect that with a more carefully designed configuration space even better results could be

[14] In order to perform k independent runs with *ParamILS* or *SMAC*, one should use a different seed (equivalent to the numRun parameter) for each run.

[15] Hutter et al. (2011) only used a single core per run of *GGA*, but still followed the protocol by Ansótegui et al. (2009) to race groups of 8 runs in parallel per core; therefore, *GGA*'s adaptive capping mechanism was the same in that work as in Ansótegui et al. (2009).

obtained. Therefore, we recommend to avoid including such parameters and to use smaller value ranges if corresponding expert knowledge is available.

Nevertheless, configurators have already been successfully applied to such large configuration spaces: *GGA++* has been used to optimize over 100 parameters of *Lingeling* (Ansótegui et al., 2015), *irace* has been used to optimize over 200 parameters of the mixed integer programming solver SCIP (López-Ibáñez & Stützle, 2014; Achterberg, 2009) and with *SMAC*, we have optimized configuration spaces with over 900 parameters (Lindauer, Hoos, Leyton-Brown, & Schaub, 2017a).

5.7 Which Parameters to Tune

Parameters should never be part of the configuration space if they change the semantics of the problem to be solved; e.g., do not tune the allowed memory or parameters that control whether a run is counted as successful (such as the allowed optimality gap in an optimization setting). Furthermore, to obtain an unbiased estimate of a configuration's performance across seeds one should not include the seed (or parameters with a similar effect) as a tunable parameter.

5.8 Runtime Metrics

A common cost metric in algorithm configuration is runtime. Obtaining clean runtime measurements is a problem that is by no means limited to algorithm configuration and also appears in general empirical algorithmics (McGeoch, 2012). However, in algorithm configuration, this problem can be even more tricky, because benchmark machines can be influenced by heavy I/O load on a shared file system created by multiple configuration runs (see Pitfall 3). Furthermore, other running processes on the same machine can influence the measurements. The latter issue can be fixed by using processor affinity to bind processes to a certain CPU. Therefore, we recommend to measure CPU time instead of wallclock time. However, binding processes does not grant exclusive usage of the assigned cores; thus other interfering factors such as operation system load and shared caches remain. Also, CPU time can sometimes be brittle; e.g., its resolution can be insufficient for very short target algorithm runs, such as milliseconds. We note that algorithm configuration *can* be used to optimize runtime at such very small scales, but extreme care needs to be taken to avoid any pitfalls associated to measuring runtimes. When possible, a better solution for this case is to measure and optimize elementary operations, such as search steps of a local search algorithm or MEMS (number of memory accesses, Knuth, 2011); however, it has to be ensured that such proxy metrics correlate well with runtime. Additionally, expensive one-time operations, such as downloading files or setting up should not be part of the measured runtime and need to be ignored, e.g. via the wrapper. Finally, it remains an open question how robust are different ways to measure runtime and related metrics and how do they influence algorithm configuration.

5.9 Monitoring Experiments

Even a well designed experiment can go wrong because of software and hardware issues. This makes conducting a flawless experiment challenging. However, the risk for falling for a pitfall can be minimized when carefully monitoring ongoing experiments.

Investigating at the first bad sign can save a lot of time and resources. An unexpectedly high load on a machine or swapping memory can be signs of misconfigured scripts. More subtle effects that should also raise one's attention include the following: (1) the target algorithm uses much more wallclock time than the CPU time reported to the configurator; (2) many configurations crash; or (3) there is a large variation between the performances of independent configuration runs that only differ in their seeds.

We recommend to analyze ongoing experiments with respect to these signs and make use of automated tools, e.g. CAVE (Biedenkapp et al., 2018), to analyze and visualize experimental results in a common and unified way independently of the underlying configurator and problem.

5.10 Comparing Configurators on Existing, Open-Source Benchmarks

Although algorithm configuration has been well established for over a decade, nearly every new paper on this topic uses a new set of benchmarks to compare different configurators. This makes it harder to assess progress in the field, and every new benchmark could again suffer from one of the pitfalls described above. Therefore, we recommend to use existing and open-source algorithm configuration benchmarks that are already well tested and can be freely used by the community. The only existing library of such benchmarks we are aware of is the algorithm configuration library AClib (Hutter et al., 2014a), which comprises 326 benchmarks (in version 1.2) based on open-source scripts and allows users to pick benchmarks from different domains (e.g., mixed integer programming, AI Planning, SAT, and machine learning) and with different characteristics (e.g., small or large configuration spaces).

6. A Generic Wrapper: Towards a Reliable and Unified AC Interface

Learning from the pitfalls above, our conclusion is that most of these pitfalls can be either completely prevented or their risk of occurrence can be substantially reduced by using a generic wrapper which wraps the executions of all target algorithm runs and has the following features:

1. Parsing the input arguments provided by the configurator in a uniform way such that a user only needs to implement a function to translate them into a call of the target algorithm;
2. Reliably limiting the run's computational resources (runtime and memory consumption);
3. Measuring the cost metric in a standardized way (for which a user only needs to implement a function to parse the output of the target algorithm); and
4. Returning the output in a standardized way.

We note that some pitfalls cannot be tested easily. E.g., the user is still responsible for domain-dependent solution checking and checking whether the configurator is used as intended. However, if using a wrapper with the features above most pitfalls can be avoided. To demonstrate the usefulness of such a generic wrapper, and to provide a practical proposal for avoiding many of the described pitfalls, we implemented such a wrapper and are already using it in the algorithm configuration library AClib (Hutter et al., 2014a), to wrap 20 different target algorithms.[16] To address the pitfalls mentioned above, our generic wrapper implements the following best practices:

Resource Limitation The tool `runsolver` (Roussel, 2011) has been used for several years by the SAT community, in SAT competitions and by many SAT developers, to limit the runtime and memory consumption of an algorithm run.[17] We also use this tool in the generic wrapper to reliably limit such resources and to measure algorithm runtimes. This addresses both Pifall 1 ("Trusting Your Target Algorithm") and Pitfall 2 ("Not Terminating Target Algorithm Runs Properly").

Solution Checking for SAT One of the exemplary instantiations of the generic wrapper we provide for SAT solvers implements solution checking to avoid issues of algorithm correctness (Pitfall 1: "Trusting Your Target Algorithm").

Writing to $TMPDIR On most high-performance clusters these days, the environment variable $TMPDIR specifies a temporary directory on a local file system (not on a shared file system) of a compute node that allows for fast write and read access without affecting the remaining cluster. If this environment variable is set, the generic wrapper writes all temporary files (e.g., log files of the `runsolver`) to this folder. It only copies these files to a permanent file system in case of a crash of the target algorithm to allow debugging of these crashes. This addresses Pitfall 3 ("Slow File System").

Furthermore, the use of the generic wrapper has the following advantages compared to implementing the same features directly in an algorithm configurator (which is nevertheless a feasible approach for some use cases):

Fair Comparisons As discussed in Pitfall 4 ("Handling Target Algorithm Runs Differently"), to compare different configurators, using a uniform wrapper will ensure that all configurators optimize the same objective function. Even if a wrapper turns out to have a bug, at least all configurators would be affected in the same way.

Easy Use of Different Configurators So far, most configurators implement different interfaces to call target algorithms. Therefore, users often implement only one of the interfaces and have not explored which of the available configurator is in fact the best one for their configuration problem. Using a generic wrapper (implementing

[16] Our package called `GenericWrapper4AC` is available at `https://github.com/automl/GenericWrapper4AC`.

[17] The runsolver uses process group IDs to keep track of running processes For example, if the memory or time limit is exceeded, it traverses the process tree bottom-up to terminate all processes that run. However, we note that it is possible to bypass this procedure if a process forks itself or starts a process on a different machine, which can neither be detected nor monitored by the `runsolver`.

either a unified interface or several configurator-specific interfaces) will also help users to easily use several configurators for their target algorithms.

Easier Implementation of New Configurators The implementation of new configurators is not an easy task, mainly because the handling of target algorithm runs may require many lines of code and is often still brittle. To reduce the burden on configurator developers, the generic wrapper can take over some of the functions required in this setting (e.g., resource limitations). Also, when translating a configurator to a new programming language, one can ensure that functionalities regarding handling that target algorithm remain exactly the same.

Open Source and Community Since the generic wrapper is an open-source implementation, we believe that the community will improve the code base and thus improve its quality and robustness over time.

Appendix A provides additional details about our generic wrapper, and an example wrapper for a SAT solver.

7. Conclusion

Empirically comparing algorithms correctly is hard. This is well known and true for almost every empirical study that involves running third-party code, stochastic algorithms and computationally expensive computations and therefore also applies to algorithm configuration. Subtle mistakes, such as measuring the wrong metric or running parallel experiments without meticulous resource management, can heavily bias the outcome. In this work, we pointed out several pitfalls that can occur in running algorithm configuration experiments and provide concrete examples of how these can impact results. We found that many of these pitfalls result from treating the objective function differently in different configurators, from issues in allocating and monitoring resource consumption, and from various issues concerning over-tuning. To prevent most of these pitfalls we share recommendations and best practices for conducting algorithm configuration experiments, which we hope to be useful for both novices and experts. We also provide an open-source implementation of a generic wrapper that provides a unified interface for the communication between target algorithms and configurators and for limiting resource consumption.

Acknowledgements

We thank Manuel López-Ibáñez and Kevin Tierney for adapting the interfaces of *irace* and *GGA* to work together with `GenericWrapper4AC`, Yuri Malitsky and Horst Samulowitz for providing the wrappers and benchmarks of Ansótegui et al. (2015), and Kevin Tierney, Manuel López-Ibáñez and Lars Kotthoff for very helpful feedback on the first draft of the paper that led to the inclusion of some further possible issues. Some of the recommendations in Section 5 were inspired by a discussion at a Dagstuhl seminar (see Lindauer & Hutter, 2017b, for more details), and we are thankful for the valuable contributions of the attendees of that discussion: Aymeric Blot, Wanru Gao, Holger Hoos, Laetitia Jourdan, Lars Kotthoff, Manuel López-Ibáñez, Nysret Musliu, Günter Rudolph, Marc Schoenauer, Thomas Stützle and Joaquin Vanschoren. We also thank the anonymous reviewers for their valuable feedback. The authors acknowledge funding by the DFG (German Research Foundation) under Emmy Noether grant HU 1900/2-1. K. Eggensperger additionally acknowledges funding by the State Graduate Funding Program of Baden-Württemberg.

Appendix A. Details on `GenericWrapper4AC`

Listing 1 shows an example for how to extend the `GenericWrapper4AC` to wrap the well-known SAT Solver *MiniSAT* (Eén & Sörensson, 2004). Since the output format is standardized in the SAT community, we already provide a domain-specific generic wrapper, called `SatWrapper`, which can parse and verify the SAT solver's output using standard tools from the annual SAT competitions. Therefore, SAT solver users only need to implement one method, which constructs a command line call string for their SAT solver from the provided input arguments (parameter settings, instance, cutoff time, seed).

```
1 class MiniSATWrapper(SatWrapper):
2 
3     def get_command_line_args(self, runargs, config):
4         cmd = "minisat -rnd-seed=%d" %(runargs["seed"])
5         for name, value in config.items():
6             cmd += " %s=%s" %(name, value)
7         cmd += " %s" %(runargs["instance"])
8         return cmd
```

Listing 1: Example GenericWrapper for SAT Solver *MiniSAT*, building on our domain-specific `SatWrapper`

In the example shown, the command line call of *MiniSAT* consists of passing the random *seed* (Line 4), adding all parameters in the format `parameter=value` (Lines 5 and 6), and appending the CNF instance name at the end (Line 7). Importantly, it takes care of all aspects of handling cutoff times, measuring runtimes, etc, to avoid the pitfalls discussed in Section 3.

```
class SimpleWrapper(AbstractWrapper):

    def get_command_line_args(self, runargs, config):
        [...]

    def process_results(self, fp, exit_code):
        try:
            resultMap = {'status': 'SUCCESS', 'cost': float(fp.read()) }
        except ValueError:
            resultMap = {'status': 'CRASHED'}

        return resultMap
```

Listing 2: Example GenericWrapper from scratch

For users of algorithm configuration outside SAT solving, Listing 2 shows an example for how to write a function `process_results` to parse algorithm outputs. Let us assume that the target algorithm only prints the target cost to be minimized (similar to the format of *irace*, López-Ibáñez et al., 2016). Reading the output of the provided file pointer `fp`, the function builds and returns a dictionary which includes the cost value and a status, which is either `SUCCESS` if the target algorithm printed only a single number or `CRASHED` otherwise. Other states can be `TIMEOUT` for using more than the cutoff time κ or `ABORT` to signal the configurator to abort the AC experiment because of major issues. Furthermore, the exit code of the target algorithm run is also provided (but not used in our example). Another possible functionality that is not shown here is to implement a (domain-specific) method to verify the target algorithm's returned solution.

Except these two target algorithm-specific functions, the `GenericWrapper4AC` handles everything else, including

- Parsing the input format; native interfaces to *ParamILS*, *ROAR* and *SMAC* are supported right now, and an additional layer to run *GGA*(++) and *irace* is available as well. (see AClib2[18] for examples).
- Calling the target algorithm and limiting its resource limits using the `runsolver` tool (Roussel, 2011)
- Measuring the CPU time of the target algorithm run (using `runsolver`)
- Returning the cost of the target algorithm run to the configurator

The `GenericWrapper4AC` is available at GitHub and can be easily installed via `python setup.py install` (including the `runsolver`) and runs on UNIX systems.

References

Achterberg, T. (2009). SCIP: solving constraint integer programs. *Mathematical Programming Computation*, *1*, 1–41.

Aigner, M., Biere, A., Kirsch, C., Niemetz, A., & Preiner, M. (2013). Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In *Proceeding of the Fourth International Workshop on Pragmatics of SAT (POS'13)*.

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U., & Amarasinghe, S. (2014). Opentuner: an extensible framework for program autotuning. In Amaral, J., & Torrellas, J. (Eds.), *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pp. 303–316. ACM.

Ansótegui, C., Gabàs, J., Malitsky, Y., & Sellmann, M. (2016). Maxsat by improved instance-specific algorithm configuration. *Artificial Intelligence*, *235*, 26–39.

Ansótegui, C., Malitsky, Y., Sellmann, M., & Tierney, K. (2015). Model-based genetic algorithms for algorithm configuration. In Yang, Q., & Wooldridge, M. (Eds.), *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pp. 733–739.

Ansótegui, C., Sellmann, M., & Tierney, K. (2009). A gender-based genetic algorithm for the automatic configuration of algorithms. In Gent, I. (Ed.), *Proceedings of the*

[18] `https://bitbucket.org/mlindauer/aclib2`

Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09), Vol. 5732 of *Lecture Notes in Computer Science*, pp. 142–157. Springer-Verlag.

Audemard, G., & Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In Boutilier, C. (Ed.), *Proceedings of the 22th International Joint Conference on Artificial Intelligence (IJCAI'09)*, pp. 399–404.

Bartz-Beielstein, T., Lasarczyk, C., & Preuss, M. (2010). The sequential parameter optimization toolbox. In Bartz-Beielstein, T., Chiarandini, M., Paquete, L., & Preus, M. (Eds.), *Experimental Methods for the Analysis of Optimization Algorithms*, pp. 337–362. Springer-Verlag.

Bayless, S., Tompkins, D., & Hoos, H. (2014). Evaluating instance generators by configuration. In Pardalos, P., & Resende, M. (Eds.), *Proceedings of the Eighth International Conference on Learning and Intelligent Optimization (LION'14)*, Lecture Notes in Computer Science. Springer-Verlag.

Bezerra, L., López-Ibáñez, M., & Stützle, T. (2016). Automatic component-wise design of multiobjective evolutionary algorithms. *IEEE Trans. Evolutionary Computation*, *20*(3), 403–417.

Biedenkapp, A., Marben, J., Lindauer, M., & Hutter, F. (2018). Cave: Configuration assessment, visualization and evaluation. In *Proceedings of the Tenth International Conference on Learning and Intelligent Optimization (LION'18)*, Lecture Notes in Computer Science. Springer-Verlag. To appear.

Biere, A. (2013). Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In Balint, A., Belov, A., Heule, M., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, Vol. B-2013-1 of *Department of Computer Science Series of Publications B*, pp. 51–52. University of Helsinki.

Biere, A. (2014). Yet another local search solver and Lingeling and friends entering the SAT competition 2014. In Belov, A., Diepold, D., Heule, M., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. B-2014-2 of *Department of Computer Science Series of Publications B*, pp. 39–40. University of Helsinki.

Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.). (2009). *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Birattari, M., & Kacprzyk, J. (2009). *Tuning metaheuristics: a machine learning perspective*, Vol. 197. Springer-Verlag.

Birattari, M., Stützle, T., Paquete, L., & Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. In Langdon, W., Cantu-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M., Schultz, A., Miller, J., Burke, E., & Jonoska, N. (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, pp. 11–18. Morgan Kaufmann Publishers.

Birattari, M. (2004). *The problem of tuning metaheuristics as seen from a machine learning perspective*. Ph.D. thesis, Université Libre de Bruxelles.

Brochu, E., Cora, V., & de Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning.. *arXiv:1012.2599*.

Brummayer, R., Lonsing, F., & Biere, A. (2012). Automated testing and debugging of SAT and QBF solvers. In Cimatti, A., & Sebastiani, R. (Eds.), *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, Vol. 7317 of *Lecture Notes in Computer Science*, pp. 44–57. Springer-Verlag.

Cáceres, L. P., & Stützle, T. (2017). Exploring variable neighborhood search for automatic algorithm configuration. *Electronic Notes in Discrete Mathematics*, *58*, 167–174.

Chiarandini, M., Fawcett, C., & Hoos, H. (2008). A modular multiphase heuristic solver for post enrolment course timetabling. In Gendreau, M., & Burke, E. (Eds.), *Proceedings of the Seventh International Conference on the Practice and Theory of Automated Timetabling*.

Dorigo, M. (2016). Swarm intelligence: A few things you need to know if you want to publish in this journal..

Eén, N., & Sörensson, N. (2004). An extensible SAT-solver. In Giunchiglia, E., & Tacchella, A. (Eds.), *Proceedings of the conference on Theory and Applications of Satisfiability Testing (SAT)*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer-Verlag.

Falkner, S., Lindauer, M., & Hutter, F. (2015). SpySMAC: Automated configuration and performance analysis of SAT solvers. In Heule, M., & Weaver, S. (Eds.), *Proceedings of the Eighteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'15)*, Lecture Notes in Computer Science, pp. 1–8. Springer-Verlag.

Feurer, M., Springenberg, T., & Hutter, F. (2015). Initializing Bayesian hyperparameter optimization via meta-learning. In Bonet, B., & Koenig, S. (Eds.), *Proceedings of the Twenty-nineth National Conference on Artificial Intelligence (AAAI'15)*, pp. 1128–1135. AAAI Press.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M., & Ziller, S. (2011). A portfolio solver for answer set programming: Preliminary report. In Delgrande, J., & Faber, W. (Eds.), *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, Vol. 6645 of *Lecture Notes in Computer Science*, pp. 352–357. Springer-Verlag.

Gebser, M., Kaufmann, B., & Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, *187-188*, 52–89.

Gent, I., Grant, S., MacIntyre, E., Prosser, P., Shaw, P., Smith, B., & Walsh, T. (1997). How not to do it. Tech. rep. 97.92, University of Leeds.

Geschwender, D., Hutter, F., Kotthoff, L., Malitsky, Y., Hoos, H., & Leyton-Brown, K. (2014). Algorithm configuration in the cloud: A feasibility study. In Pardalos, P., & Resende, M. (Eds.), *Proceedings of the Eighth International Conference on Learning and Intelligent Optimization (LION'14)*, Lecture Notes in Computer Science, pp. 41–46. Springer-Verlag.

Hansen, N. (2006). The CMA evolution strategy: a comparing review. In Lozano, J., Larranaga, P., Inza, I., & Bengoetxea, E. (Eds.), *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, pp. 75–102. Springer.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep reinforcement learning that matters.. *arXiv:1709.06560.*

Heule, M., Hunt, W., & Wetzler, N. (2014). Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Software Testing Verification and Reliability*, *24*(8), 593–607.

Hooker, J. (1995). Testing heuristics: We have it all wrong. *Journal of Heuristics*, *1*, 33–42.

Hoos, H. (2012). Programming by optimization. *Communications of the ACM*, *55*(2), 70–80.

Hoos, H., Kaufmann, B., Schaub, T., & Schneider, M. (2013). Robust benchmark set selection for boolean constraint solvers. In Pardalos, P., & Nicosia, G. (Eds.), *Proceedings of the Seventh International Conference on Learning and Intelligent Optimization (LION'13)*, Vol. 7997 of *Lecture Notes in Computer Science*, pp. 138–152. Springer-Verlag.

Howe, A., & Dahlman, E. (2002). A critical assessment of benchmark comparison in planning. *Journal of Artificial Intelligence Research*, *17*, 1–33.

Hutter, F., Babić, D., Hoos, H., & Hu, A. (2007a). Boosting verification by automatic tuning of decision procedures. In O'Conner, L. (Ed.), *Formal Methods in Computer Aided Design (FMCAD'07)*, pp. 27–34. IEEE Computer Society Press.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2010a). Automated configuration of mixed integer programming solvers. In Lodi, A., Milano, M., & Toth, P. (Eds.), *Proceedings of the Seventh International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR'10)*, Vol. 6140 of *Lecture Notes in Computer Science*, pp. 186–202. Springer-Verlag.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2010b). Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligenc (AMAI), Special Issue on Learning and Intelligent Optimization*, *60*(1), 65–89.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In Coello, C. (Ed.), *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION'11)*, Vol. 6683 of *Lecture Notes in Computer Science*, pp. 507–523. Springer-Verlag.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2012). Parallel algorithm configuration. In Hamadi, Y., & Schoenauer, M. (Eds.), *Proceedings of the Sixth International Conference on Learning and Intelligent Optimization (LION'12)*, Vol. 7219 of *Lecture Notes in Computer Science*, pp. 55–70. Springer-Verlag.

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

Hutter, F., Hoos, H., & Stützle, T. (2007b). Automatic algorithm configuration based on local search. In Holte, R., & Howe, A. (Eds.), *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*, pp. 1152–1157. AAAI Press.

Hutter, F., Lindauer, M., Balint, A., Bayless, S., Hoos, H., & Leyton-Brown, K. (2017). The configurable SAT solver challenge (CSSC). *Artificial Intelligence*, *243*, 1–25.

Hutter, F., López-Ibánez, M., Fawcett, C., Lindauer, M., Hoos, H., Leyton-Brown, K., & Stützle, T. (2014a). AClib: a benchmark library for algorithm configuration. In Pardalos, P., & Resende, M. (Eds.), *Proceedings of the Eighth International Conference on Learning and Intelligent Optimization (LION'14)*, Lecture Notes in Computer Science. Springer-Verlag.

Hutter, F., Tompkins, D., & Hoos, H. (2002). Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In Hentenryck, P. V. (Ed.), *Proceedings of the international conference on Principles and Practice of Constraint Programming*, Vol. 2470 of *Lecture Notes in Computer Science*, pp. 233–248. Springer-Verlag.

Hutter, F., Xu, L., Hoos, H., & Leyton-Brown, K. (2014b). Algorithm runtime prediction: Methods and evaluation. *Artificial Intelligence*, *206*, 79–111.

Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2011). Algorithm selection and scheduling. In Lee, J. (Ed.), *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11)*, Vol. 6876 of *Lecture Notes in Computer Science*, pp. 454–469. Springer-Verlag.

Kadioglu, S., Malitsky, Y., Sellmann, M., & Tierney, K. (2010). ISAC - instance-specific algorithm configuration. In Coelho, H., Studer, R., & Wooldridge, M. (Eds.), *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*, pp. 751–756. IOS Press.

Knuth, D. (2011). *The Art of Computer Programming, Volume IV*. Addison-Wesley.

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R., Danna, E., Gamrath, G., Gleixner, A., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D., & Wolter, K. (2011). MIPLIB 2010. *Mathematical Programming Computation*, *3*, 103–163.

Kotthoff, L. (2014). Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, *35*(3), 48–60.

Laguna, M. (2017). Journal of heuristic policies on heuristic search research..

Lindauer, M., Hoos, H., Hutter, F., & Schaub, T. (2015). Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, *53*, 745–778.

Lindauer, M., Hoos, H., Leyton-Brown, K., & Schaub, T. (2017a). Automatic construction of parallel portfolios via algorithm configuration. *Artificial Intelligence*, *244*, 272–290.

Lindauer, M., & Hutter, F. (2017b). Pitfalls and best practices for algorithm configuration (breakout session report). *Dagstuhl Reports*, *6*, 70–72.

Lodi, A., & Tramontani, A. (2014). Performance variability in mixed-integer programming. In Topaloglu, H., Smith, J., & Greenberg, H. (Eds.), *Theory Driven by Influential Applications*, chap. 1, pp. 1–12. INFORMS.

López-Ibáñez, M., Dubois-Lacoste, J., Caceres, L. P., Birattari, M., & Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, *3*, 43–58.

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

Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2012). Parallel SAT solver selection and scheduling. In Milano, M. (Ed.), *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming (CP'12)*, Vol. 7514 of *Lecture Notes in Computer Science*, pp. 512–526. Springer-Verlag.

Manthey, N. (2014b). Riss 4.27. In Belov, A., Diepold, D., Heule, M., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. B-2014-2 of *Department of Computer Science Series of Publications B*, pp. 65–67. University of Helsinki.

Manthey, N., & Lindauer, M. (2016). Spybug: Automated bug detection in the configuration space of sat solvers. In Creignou, N., & Berre, D. L. (Eds.), *Proceedings of the Nineteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'16)*, Lecture Notes in Computer Science, pp. 554–561. Springer-Verlag.

Manthey, N., & Steinke, P. (2014a). Too many rooks. In Belov, A., Diepold, D., Heule, M., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. B-2014-2 of *Department of Computer Science Series of Publications B*, pp. 97–98. University of Helsinki.

McGeoch, C. (1987). *Experimental Analysis of Algorithms*. Ph.D. thesis, Carnegie-Mellon University, Computer Science.

McGeoch, C. (2002). Experimental analysis of algorithms. In Pardalos, P., & Romeijn, E. (Eds.), *Handbook of Global Optimization*, pp. 489–513. Springer-Verlag.

McGeoch, C. C. (2012). *A Guide to Experimental Algorithmics*. Cambridge University Press.

Mockus, J., Tiesis, V., & Zilinskas, A. (1978). The application of Bayesian methods for seeking the extremum. *Towards Global Optimization*, *2*(117-129).

Mugrauer, F., & Balint, A. (2013). SAT encoded low autocorrelation binary sequence (LABS) benchmark description. In Balint, A., Belov, A., Heule, M., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, Vol. B-2013-1 of *Department of Computer Science Series of Publications B*. University of Helsinki.

Oh, C. (2014). MiniSat HACK 999ED, MiniSat HACK 1430ED and SWDiA5BY. In Belov, A., Diepold, D., Heule, M., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. B-2014-2 of *Department of Computer Science Series of Publications B*, p. 46. University of Helsinki.

Rice, J. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Roussel, O. (2011). Controlling a solver execution with the runsolver tool. *Journal on Satisfiability, Boolean Modeling and Computation*, *7*(4), 139–144.

Schneider, M., & Hoos, H. (2012). Quantifying homogeneity of instance sets for algorithm configuration. In Hamadi, Y., & Schoenauer, M. (Eds.), *Proceedings of the Sixth International Conference on Learning and Intelligent Optimization (LION'12)*, Vol. 7219 of *Lecture Notes in Computer Science*, pp. 190–204. Springer-Verlag.

Shahriari, B., Swersky, K., Wang, Z., Adams, R., & de Freitas, N. (2016). Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, *104*(1), 148–175.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In Bartlett, P., Pereira, F., Burges, C., Bottou, L., & Weinberger, K. (Eds.), *Proceedings of the 26th International Conference on Advances in Neural Information Processing Systems (NIPS'12)*, pp. 2960–2968.

Soos, M. (2014). CryptoMiniSat v4. In Belov, A., Diepold, D., Heule, M., & Järvisalo, M. (Eds.), *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Vol. B-2014-2 of *Department of Computer Science Series of Publications B*, p. 23. University of Helsinki.

Styles, J., & Hoos, H. (2015). Ordered racing protocols for automatically configuring algorithms for scaling performance. In Blum, C., & Alba, E. (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'13)*, pp. 551–558. ACM.

Thornton, C., Hutter, F., Hoos, H., & Leyton-Brown, K. (2013). Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In Dhillon, I., Koren, Y., Ghani, R., Senator, T., Bradley, P., Parekh, R., He, J., Grossman, R., & Uthurusamy, R. (Eds.), *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*, pp. 847–855. ACM Press.

Tompkins, D., & Hoos, H. (2005). UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, Lecture Notes in Computer Science, pp. 306–320. Springer-Verlag.

Vallati, M., Fawcett, C., Gerevini, A., Hoos, H., & Saetti, A. (2013). Automatic generation of efficient domain-optimized planners from generic parametrized planners. In Helmert, M., & Röger, G. (Eds.), *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SOCS'14)*. AAAI Press.

Weise, T., Chiong, R., & Tang, K. (2012). Evolutionary optimization: Pitfalls and booby traps. *Journal of Computer Science and Technology*, *27*, 907–936.

Xu, L., Hoos, H., & Leyton-Brown, K. (2010). Hydra: Automatically configuring algorithms for portfolio-based selection. In Fox, M., & Poole, D. (Eds.), *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, pp. 210–216. AAAI Press.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, *32*, 565–606.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2011). Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *Proc. of RCRA workshop at IJCAI*.