Chapter preprint of the second edition of the Handbook of Satisfiability (2021)
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
Published by IOS Press. The print edition can be ordered at
https://www.iospress.nl/book/handbook-of-satisfiability-2/
The published electronic chapters (Version of Record) are available at
https://ebooks.iospress.nl/ISBN/978-1-64368-161-0/

Chapter 12

Automated Configuration and Selection of SAT Solvers

Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown

12.1. Introduction

Although SAT is widely believed to be hard in the worst case, many interesting types of SAT instances can be solved efficiently in practice, using heuristic algorithms. A wide variety of different heuristic mechanisms have been identified by the research community over the decades, partly by unambiguously improving on past ideas, but largely by finding different heuristics that work well on different types of SAT instances. Because of this sensitivity to instance characteristics, there is no single best SAT solver, and state-of-the-art solvers typically expose some design choices as parameters that can be set at runtime, thereby enabling users to optimize performance to a given class of SAT instances. This chapter focuses on automated approaches for performing such optimization, which is termed *algorithm configuration*. We also discuss the related problem of *algorithm selection*: i.e., methods for choosing one algorithm from a finite, typically modestly sized *portfolio* on a per-instance basis, with the goal of achieving better performance across instances than can be obtained from any single solver in the portfolio.

Automated algorithm configuration and per-instance algorithm selection are both examples of *meta-algorithmic design techniques*, i.e., generic methods for building algorithms for a given problem from a set or space of candidate algorithms (which can be small, as in the case of algorithm selection, or large, as in the case of algorithm configuration). Automated algorithm configuration and per-instance algorithm selection are of particular interest in the context of SAT, because of their major impact on solving SAT and related problems, such as MaxSAT, ASP, CSP and QBF. Conversely, SAT has played a major role in the development of algorithm configuration and selection techniques.

While the major part of this chapter is focused on automated algorithm configuration and per-instance algorithm selection, we also survey some conceptually more complex approaches that go beyond "pure" configuration and selection: combining configuration and selection together; building algorithm schedules rather than simply picking a single algorithm; extending methods to leverage

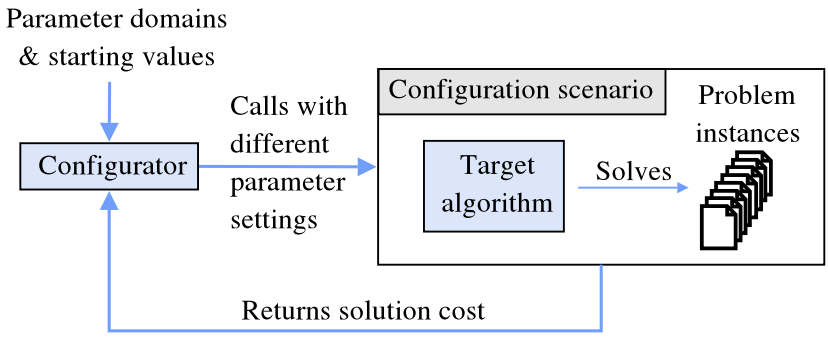

Figure 12.1. A configuration scenario includes an algorithm to be configured and a collection of problem instances. A configuration procedure executes the target algorithm with specified parameter settings on some or all of the instances, receives information about the performance of these runs, and uses this information to decide which subsequent parameter configurations to evaluate.

parallel computation; analyzing the importance of parameters; dynamically controlling parameters; and using configuration techniques for finding bugs in SAT solvers. Finally, we outline some open challenges related to the methods discussed in this chapter, the tools based on them, and their application.

12.2. Algorithm configuration

Algorithm configuration is the problem of automatically identifying parameter settings that optimize the performance of a given algorithm on a specific set or distribution of problem instances, as illustrated in Figure 12.1. Observe that an algorithm configuration procedure is a *meta-algorithm* in the sense that it takes another, arbitrary algorithm as part of its input.

More formally, the algorithm configuration problem can be stated as follows: given an algorithm A with possible parameter settings $\boldsymbol{\Theta} = \Theta_1 \times \cdots \times \Theta_n$, a distribution $\mathcal{D}$ over possible inputs with domain Π that A can be run on, and a cost metric $c : \boldsymbol{\Theta} \times \Pi \to \mathbb{R}$, find the configuration $\boldsymbol{\theta} = \langle \theta_1, \ldots, \theta_n \rangle \in \boldsymbol{\Theta}$ that minimizes c in expectation over $\mathcal{D}$. A common special case arises where $\mathcal{D}$ is a uniform distribution over $\Pi = \{\pi_1, \ldots, \pi_k\}$, in which case we can write the problem as finding the minimizer of the blackbox function $f(\boldsymbol{\theta}) := \frac{1}{k} \sum_{i=1}^{k} c(\boldsymbol{\theta}, \pi_i)$. When applying algorithm configuration to SAT solvers, the π_i are SAT instances deemed representative for the kind of instances for which we aim to optimize performance. We call f a blackbox function, because it depends on the cost $c(\boldsymbol{\theta}, \pi_i)$, which cannot be reasoned about analytically, but can only be evaluated by actually running A with configuration $\boldsymbol{\theta}$ on π_i.

Although the problem of "tuning" algorithm parameters might seem narrow, we note that many aspects of algorithm design can be expressed parametrically, from high-level design decisions to low-level aspects of specific heuristic mecha-

nisms. For example, parameters can be real-valued, integer-valued, or categorical. Parameters can also be conditional on an instantiation of other parameters—consider, e.g., the parameters of a heuristic mechanism h, which are completely ignored unless h is enabled via another, categorical parameter. Finally, combinations of otherwise valid parameter instantiations can be forbidden.

Indeed, the flexibility offered by expressing heuristic ideas parametrically and the effectiveness of algorithm configuration procedures can empower algorithm developers to adopt a new design philosophy that is both easier and produces better results. Instead of seeking to minimize the number of exposed parameters, a developer can instead aim to encode all plausible design ideas parametrically, leaving their eventual evaluation to the algorithm configuration procedure (see, e.g., [HBHH07, KXHLB09, KXHLB16]). This philosophy has been formalized in a design paradigm called Programming by Optimization (PbO), which facilitates the exploration and automatic optimization of design choices as an integral part of the software development process [Hoo12].

Over the past decade, evidence has accumulated that algorithm configuration is an effective way to design high-performance algorithms across a wide range of applications. This is because human experts are not only ill-suited for solving these noisy high-dimensional optimization problems manually, but often also unmotivated to carry out this tedious task (not to mention that experts rarely have time and are also expensive in practice). Our focus in this chapter is on SAT; nevertheless, it is important to point out that automated algorithm configuration has been demonstrated to produce dramatic improvements in a wide range of other combinatorial optimization problems beyond SAT, such as MIP [HHLB10], AI planning [VFG+13], TSP [CFH08], and answer set programming [SLS12]; these improvements routinely exceed those achieved manually by human experts. Likewise, very similar automated approaches have yielded better performance than humans for optimizing hyperparameters of deep convolutional neural networks [BBBK11, SLA12] and their architectures [DSH15], as well as generic image recognition pipelines [BYC14]; they have even led to effective push-button approaches for machine learning, by jointly selecting feature selection methods, algorithms and hyperparameter settings in prominent machine learning packages [THHLB13, FKE+15].

Automated algorithm configuration approaches also have benefits beyond optimizing algorithm performance. They can additionally be used to speed up the process of developing effective software (freeing developers from the burden of tedious experimentation); to apply highly parameterized algorithms to a new problem domain; and to improve protocols for empirical research in computer science (enabling fair comparisons of software based on the same fully formalized parameter optimization approach).

In the remainder of this section, we briefly survey some widely used algorithm configuration procedures—particularly those that have been widely applied to SAT. Throughout, we emphasize key conceptual ideas and methods.

12.2.1. ParamILS

The ParamILS framework [HHS07, HHLBS09] was the first to give rise to practical, general-purpose algorithm configuration procedures; it still yields excel-

lent performance on many problems today. This approach uses iterated local search [LMS02] as its underlying search strategy. It begins with a default parameter configuration, evaluates performance against a list of random configurations, and then performs local search in configuration space, starting from the best configuration so far. The local search process changes the setting of one parameter at a time and keeps those changes resulting in empirical performance improvements. After finding a local minimum, ParamILS randomly changes some parameters in order to escape; with a small probability, it also performs a complete restart to a random point in the configuration space. Throughout, ParamILS keeps track of the best configuration seen so far, the so-called *incumbent*.

ParamILS has two variants that differ in the approach they take to deciding how many runs to use for estimating the performance of each configuration. The *BasicILS* variant is most straightforward: it evaluates each configuration according to a fixed number of runs on a fixed set of instances. However, this variant has the drawback that it can spend far too long evaluating configurations that exhibit poor overall performance. The more practical *FocusedILS* variant starts by performing only a single run per configuration; if a new configuration does not outperform the incumbent on the same instance, it is rejected. If it beats the incumbent, FocusedILS doubles the number of instances to consider and tries again, until eventually it evaluates the new configuration on the same set of instances as the incumbent has been evaluated on. The number of runs allocated to incumbents also grows slowly over time. Overall, FocusedILS thus devotes most of its time to promising configurations, while also maintaining statistically robust performance estimates for incumbent configurations.

Both ParamILS variants can be sped up via an *adaptive capping* strategy for choosing the time after which to terminate unsuccessful target algorithm runs. Intuitively, when comparing the performance of two configurations $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ on an instance, and we already know that $\boldsymbol{\theta}_1$ solves the instance in time t_1, we do not need to run $\boldsymbol{\theta}_2$ for longer than t_1: we do not need to know precisely how bad $\boldsymbol{\theta}_2$ is, as long as we know that $\boldsymbol{\theta}_1$ is better. While this strategy exactly preserves the search trajectory of ParamILS and leads to substantial speedups, it does not perform effectively after a perturbation step. A more aggressive adaptive capping variant terminates all evaluations prematurely if they require more time than the corresponding evaluations of the *incumbent* configuration times a small bound multiplier.

ParamILS has led to very substantial improvements of SAT solvers for various applications. For example, it reduced the penalized average running time (with penalty factor 10, i.e., PAR-10)[1] of the SAT solver Spear [BH07] on a set of formal software verification instances by a factor of 500 [HBHH07]; this allowed Spear to win the QF_BV category of the SMT competition 2007. More recently, ParamILS was used to configure sat13, a high-performance SAT solver by Donald Knuth, achieving a 1.4-fold improvement in geometric average speedup over a default defined by the algorithm designer, even when tested on a diverse set of benchmark instances up to over three orders of magnitude harder than those in the given training set [Knu15]. This demonstrates that general-purpose automatic algorithm configuration can achieve very significant performance improvements

[1] PAR-10 counts timeouts at a maximal captime $\bar{\kappa}$ as having taken time $10 \cdot \bar{\kappa}$ [HHLBS09].

over manual optimization carried out by the very best algorithm designers.

12.2.2. GGA and GGA++

The Gender-based Genetic Algorithm (GGA) [AST09] is a genetic algorithm for algorithm configuration that evolves a population of parameter configurations (having pairs of configurations mate and produce offspring) and uses the concept of gender to balance the diversity and fitness of the population. GGA's intensification mechanism for increasing the number of runs N it performs for each configuration over time keeps N constant in each generation, starting with small N_{start} in the first generation and linearly increasing N up to a larger N_{target} in a user-defined target generation and thereafter. GGA implements an adaptive capping mechanism by evaluating several candidate configurations in parallel until the first ones succeed and then terminating the remaining evaluations.

In experiments by Ans'otegui et al. [AST09], GGA performed more robustly than ParamILS on three configuration scenarios with 4 to 7 parameters. In a later comparison by Hutter et al. [HHLB11], ParamILS and GGA performed comparably across 15 configuration scenarios with 4 to 26 parameters, and ParamILS performed substantially better for the 76-dimensional configuration problem of optimizing the mixed integer programming solver CPLEX. GGA was also successfully used to configure solvers for various variants of MaxSAT, allowing the automated construction of portfolios that won several categories in the 2013 and 2014 MaxSAT evaluations [AMS14].

Inspired by the model-based approach underlying SMAC (see Section 12.2.3 below), Ans'otegui et al. [AMST15] introduced GGA^{++}, a version of GGA that relies on random forest models for performance prediction. More precisely, GGA^{++} enhances the evolutionary approach underlying GGA by using a novel random forest variant in combination with a targeted sampling procedure to optimize configurations obtained by recombination. Ans'otegui et al. ansotegui-ijcai15a reported promising results compared to GGA and SMAC on two algorithm configuration scenarios based on prominent SAT solvers; unfortunately, however, this comparison was confounded by two significant bugs in the algorithm wrapper (see Footnote 9 of Eggensperger et al.[ELH19] for details).

12.2.3. SMAC

Sequential, Model-based Algorithm Configuration (SMAC) [HHLB11] is an algorithm configuration method that uses a *empirical performance model [LBNS09, HXHLB14]* (that maps parameter values to algorithm performance and is learned from data) to identify algorithm configurations worth evaluating. In essence, SMAC can be understood as using this model to replace the local search component of ParamILS. SMAC begins by carrying out some initial algorithm runs on both the algorithm default and, optionally, a small set of random configurations. It calls the best performing of these the *incumbent* and then iterates the following steps until a given time budget is exhausted:

1. fit a probabilistic regression model to the performance data gathered so far;

2. use the model to determine a list of promising configurations[2], interleaving configurations sampled uniformly at random;
3. compare configurations from the promising list to the current incumbent, saving the resulting performance data, and updating the incumbent if appropriate.[3]

As with ParamILS, the mechanism used for comparing configurations is critical; SMAC uses an approach very similar to that of FocusedILS, performing adaptive capping and dedicating more running time to better-performing configurations. Another key element of SMAC involves the means by which the model is used to identify promising configurations; it uses a so-called *acquisition function* to trade off exploration (choosing configurations in parts of the space where the model has low confidence) with exploitation (choosing configurations in parts of the space where the model predicts fast running times). This acquisition function is computed based on model predictions and then optimised across the configuration space to identify the most promising next configuration.

SMAC can be used with different types of empirical performance models and different acquisition functions. Work using it to configure solvers for SAT and similar combinatorial problems most commonly uses random forests [Bre01] as its underlying performance model and adopts the expected improvement acquisition function [JSW98], optimised via iterated local search. Random forests are particularly suitable for modelling the running time of solvers for hard combinatorial problems, as they can easily handle mixed continuous/categorical inputs, have a low computational complexity for constructing the model and making predictions, act as automated feature selectors to detect the few most important inputs, and can deal with large heteroscedastic noise. However, since they do not usually make probabilistic predictions, they had to be extended by using randomly-chosen split points and uncertainties computed based on the discrepancy of the predictions across the different trees [HXHLB14].

Hutter et al. [HHLB11] used 17 different configuration scenarios to compare the performance of SMAC to that of GGA and ParamILS. In this evaluation, SMAC performed statistically significantly better than ParamILS in 11 of 17 cases (achieving speedup factors ranging from 0.93× to 2.25×) and statistically significantly better than GGA in 13 of 17 scenarios (achieving speedup factors ranging from 1.01× to 2.76×).

SMAC has also been used in numerous practical applications. Notably, it was used by the US Federal Communication Commission to build a solver for determining the feasibility of radio spectrum repackings in its 2016–17 Incentive Auction; this solver made use of SAT encodings and various prominent SAT algorithms. The stakes of this auction were extremely high: 84 MHz of radio spectrum were purchased across the whole US and Canada at a cost of over $10 billion. As reported by Newman et al. [NFLB17], configuration with SMAC yielded large performance gains in this domain. Gnovelty+PCL [DP13] was the best single

[2]It is not necessary to identify more than one configuration in this step, but doing so can amortize the cost of fitting the model in Step 1.

[3]In order to ensure that the time for the construction and use of models does not outweigh the time for actually running algorithms, SMAC iteratively evaluates configurations from the promising list until the time for step 3 is larger than that required for steps 1 and 2 combined.

solver, able to solve 79.96% of representative problem instances within a minute; the parallel portfolio (see Section 12.4.3) of the best 20 algorithms could only solve 81.58% within the same amount of wall-clock time. After configuration, an eight-algorithm parallel portfolio based on SATenstein [KXHLB16] and clasp [GKNS07] identified using Hydra [XHLB10, see Section 12.4.1] was able to solve 96.03% of these problems in a minute (and 87.73% in only a second). Subsequent analysis showed that this improvement in solver performance is likely to have led directly to better economic outcomes, on the order of over $700 million in cost savings and over $300 million in additional economic value created by a more efficient auction outcome [NLBMS17].

12.2.4. The Configurable SAT Solver Challenges

The SAT community has a long history of regularly assessing the state of the art via competitions [JBRS12]. The first SAT competition dates back to the year 2002 [SBH05], and the event has been growing over time: in 2018, as many as 106 solver versions participated across four tracks [HJS18].

A drawback of traditional solver competitions is that they fail to reward solver developers for identifying versatile, powerful designs that incorporate a large number of interchangeable heuristics; instead, they reward developers primarily for configuring their solvers in a way that ultimately targets the distribution used for evaluation. The *Configurable SAT Solver Challenge (CSSC)*, held alongside the SAT competition in 2013 and in 2014 as part of the 2014 FLoC Olympic Games, addressed this issue [HLB+17]. It evaluated SAT solver performance *after* application-specific customization, taking into account the fact that effective algorithm configuration procedures can automatically customize solvers for a given distribution of benchmark instances. Specifically, for each distribution $\mathcal{D}$ of instances and each SAT solver S with configuration space $\boldsymbol{\Theta}_S$, an automated fixed-time offline configuration phase determined parameter settings of $\boldsymbol{\theta}_S \in \boldsymbol{\Theta}_S$ optimized for high performance on $\mathcal{D}$. Then, the performance of S on new instances from $\mathcal{D}$ (not seen during the configuration process) was evaluated with these settings $\boldsymbol{\theta}_S$, and the solver with the best performance was declared the winner. To avoid bias arising from the choice of algorithm configuration method, the CSSC independently used all three of the most widely used algorithm configuration methods: ParamILS, GGA, and SMAC.

Overall, the two CSSC events held to date have yielded four main insights. First, algorithm configuration often improved performance substantially, in some cases yielding orders-of-magnitude speedups (e.g., in the 2014 CSSC, the PAR-10 score of clasp [GKNS07] improved from 705 to 5 on N-Rooks instances [MS14], and the PAR-10 score of ProbSAT [BS12] on 5-SAT instances improved from 3000 to 2, i.e., from solving no instance at all in 300 seconds to solving all instances in an average of 2 seconds). Second, some solvers benefited more from configuration than others, meaning that algorithm rankings were substantially different after configuration than before. For example, in the Random SAT+UNSAT track of CSSC 2013, the solvers ranking first (clasp [GKNS07]), second (Lingeling [Bie13]) and third (Riss3g [Man13]) would have ranked 6th, 4th and 5th, respectively, based on performance with default parameter settings. Third, the configuration

budget used also affected the ranking; in particular, algorithms with larger configuration spaces needed longer configuration budgets (on the order of days) to achieve peak performance. Fourth, out of the three configurators used in the CSSC 2014, SMAC yielded the best performance in 51 configuration scenarios, ParamILS in 17, and GGA in 4. If only SMAC had been used for the configuration process, the largest slowdown of any solver on test instances would have been by a factor of 1.5; the same slowdown factor would have been 1.8 if ParamILS had been used exclusively, and 30.5 if GGA had been used exclusively.

12.2.5. Other Approaches

Finally, we survey some recent algorithm configuration approaches whose focus is not on the strategy for selecting the next configuration to evaluate but on the resources used for each evaluation.

12.2.5.1. iRace

The iterative racing procedure irace [LIDLSB11] uses an intensification mechanism based on racing a set of candidate configurations against each other, i.e., performing runs for each of them on one instance at a time and dropping candidates whose performance is statistically dominated. Based on the evaluations thus gathered, it constructs one-dimensional density estimates of which values perform well for each parameter; its search strategy is then an estimation of distribution approach that samples individual parameter values from these one-dimensional distributions. To increasingly focus the sampling process towards the most promising configurations, irace gradually decreases the standard deviations of these distributions (this process is dubbed *volume reduction*). Recently, Pérez Cáceres et al. [PCLHS17] have integrated a capping mechanism inspired by that used in ParamILS into irace and reported promising initial results on various configuration scenarios.

12.2.5.2. Structured Procrastination

The *Structured Procrastination (SP)* algorithm [KLBL17] is notable for offering the first nontrivial performance guarantees for general algorithm configuration with an average running time minimization objective. This work considered a worst-case setting in which an adversary causes every deterministic choice to play out as poorly as possible, but where observations of random variables are unbiased samples. In this setting, it is straightforward to argue that any fixed, deterministic heuristic for searching the space of configurations can be extremely unhelpful. The approach underlying SP therefore focuses on obtaining candidate configurations via random sampling.

Any method based on random sampling will eventually encounter the optimal configuration; the crucial question is the amount of time that this will take and whether the algorithm is guaranteed to eventually recognize the optimal configuration as such when it has been sampled often enough. Out of the methods surveyed so far, in discrete configuration spaces, ParamILS and SMAC provably converge to the optimal configuration, while GGA and GGA++ do not have this

property.[4] The key result of Kleinberg et al. [KLBL17] is that SP is guaranteed to find a near-optimal configuration with high probability, with worst-case running time that nearly matches a lower bound on what is possible and that asymptotically dominates that of existing alternatives, such as SMAC.

Unfortunately, SP is not useful in practice; it takes an extremely long time to run, even on configuration scenarios that are easy for existing methods, and hence has not been implemented or evaluated empirically. The problem is that SP treats every instance like the worst case, in which it is necessary to achieve a fine-grained understanding of every configuration's running time in order to distinguish between them. Extending SP to make it practical remains an active topic of research. Notably, a new variant called *Structured Procrastination with Confidence* [KLBLG19] maintains SP's theoretical guarantees while improving performance outside the worst case. While it is more promising than SP, this very recent algorithm has only been evaluated empirically on a single dataset to compare it to the two algorithms discussed next. (In this comparison, it achieved better performance, but a much more extensive study would be required to draw reliable conclusions.)

12.2.5.3. LeapsAndBounds, CapsAndRuns

A configuration procedure called LeapsAndBounds [WGS18b] is based on SP but improves upon it in various ways. In theoretical terms, LeapsAndBounds offers an improvement by more closely matching the lower bound in SP's analysis and refining that analysis to remove dependence on one of the parameters used. More importantly for our discussion here, LeapsAndBounds also improves performance outside the worst case. A second refinement of SP called CapsAndRuns [WGS18a] goes even further in this direction. Limited inital experiments (running Minisat [ES04] on SAT instances generated using CNFuzzDD (`http://fmv.jku.at/cnfuzzdd`) show considerable promise over the original version of SP. However, LeapsAndBounds has not been evaluated beyond this single setting, and has not been compared to ParamILS, SMAC, GGA/GGA++ or irace.

12.3. Per-instance algorithm selection

Per-instance algorithm selection entails automatically choosing from a "portfolio" of different algorithms for a given problem (such as SAT) every time a new problem instance is presented [Ric76]. This choice is made based on features of the given instance π, as illustrated in Figure 12.2. Like an algorithm configurator, a per-instance algorithm selector is a meta-algorithm that takes a set of algorithms as part of its input.

More formally, the *per-instance algorithm selection problem* is defined as follows. We start with a set $\mathcal{P}$ of algorithms, which we call a *portfolio*, a distribution $\mathcal{D}$ over possible inputs with domain Π, and a cost metric $c : \mathcal{P} \times \Pi \to \mathbb{R}$. We will

[4] GGA and GGA^{++} use a finite maximum number N_{end} to evaluate each configuration, but indefinite growth of this number is required for reliably selecting the optimal configuration in the presence of noise.

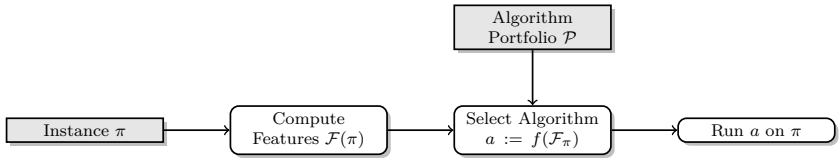

Figure 12.2. The computational core of algorithm selection at test time: for each new instance π, features $\mathcal{F}(\pi)$ are computed, and based on these, function f selects an algorithm a from the set of candidate algorithms $\mathcal{P}$ to run on π.

summarize problem instances as vectors of ϕ real-valued *features*; we represent this reduction as $\mathcal{F} : \mathcal{D} \to \mathbb{R}^\phi$. Our goal is to construct a function $f : \mathbb{R}^\phi \to \mathcal{P}$ that maps any given input (here: a SAT instance) to an algorithm from the portfolio; this function is called a *per-instance algorithm selector*. Typically, the aim is to minimize c in expectation over $\mathcal{D}$, but more complex goals are possible, such as aiming for diversity across subcategories of problem instances; indeed, such evaluation functions have been used in past SAT competitions.

Per-instance algorithm selection has two distinct phases: an offline training phase, in which a selector is constructed, and an online use phase, where that selector is used to determine the algorithm to be run on new problem instances π. As we will see in in the following, selector construction is often approached as a supervised machine learning task, using (cost-based) classification or regression techniques, although there also exist approaches based on feature-based clustering of instances.

In the following, we consider per-instance algorithm selection approaches along with two extensions, which turn out to be important for achieving state-of-the-art performance in SAT and other problems: pre-solving schedules and backup solvers. The former are sequences of algorithms run prior to the algorithm selection process, mostly in order to avoid potentially costly feature extraction on instances that are solved easily; the latter are used as a fall-back when feature computation fails. We describe in the greatest detail approaches that have had broad impact in the SAT community and that have been demonstrated to achieve both strong and robust performance.

12.3.1. SATzilla

The first version of SATzilla was built for the 2003 SAT Competition and was indeed the first algorithm selection system to be entered into such a competition [NLBA+03, NLBH+04]. It used independent ridge regression models to predict the performance of component SAT solvers based on a rich set of up to 84 instance features, and selected the solver with the best predicted running time. These models were trained in an offline phase on a broad and diverse set of training instances. In the 2004 submission, a local-search-based pre-solver was added to quickly solve easily satisfiable instances without incurring the cost of computing features. Overall, SATzilla won two silver and one bronze medal across the nine tracks of the 2003 SAT Competition and two bronze medals in the 2004

SAT Competition, providing an early indication for the potential of per-instance algorithm selection for SAT.

For the 2007 SAT Competition, this basic design was elaborated in various ways [XHHLB07]. Two key updates were made to the empirical performance models. First, SATzilla 2007 was based on so-called hierarchical hardness models, using a sparse multinomial logistic regression (SMLR) classifier to predict the probability of instance satisfiability and then using this result to weight the predictions of two quadratic regression models, trained only on satisfiable and unsatisfiable instances respectively [XHLB07]. Second, a method from survival analysis [SH79] was used to address bias stemming from the fact that true algorithm running times are not fully observed when long runs are capped. Several other innovations tuned performance by optimizing on a separate validation set of instances. A pre-solver selection mechanism allowed up to two solvers to be run sequentially before feature computation; an automatic solver subset selection mechanism was used to drop solvers whose inclusion worsened overall performance; and a backup solver was identified for use when feature computation failed. Finally, the set of features was somewhat reduced (to 48) to exclude those whose time complexity exceeded their marginal value. This version of SATzilla was very successful in the 2007 SAT Competition, winning three gold, one silver, and one bronze medal. Following the 2007 competition, SATzilla was improved further by making the selector construction more scalable and completely automated, by integrating candidate solvers based on local search, by predicting competition score instead of running time, and by using hierarchical hardness models that take into account different types of SAT instances [XHHLB08]. The resulting version won three gold and two silver medals in the 2009 competition [XHHLB09].

In 2011, a substantially improved version of SATzilla was introduced, which showed outstanding performance in the evaluation track of the 2011 SAT Competition and in the 2012 SAT Challenge [XHHLB12]. This version of SATzilla differed from the previous design by using cost-sensitive decision forests to predict for every pair of component solvers the one that would perform better on a given SAT instance, then employing majority voting over these predictions to select the solver to run. Specifically, the pairwise classification mechanism at the heart of this approach penalized misclassifications in direct proportion to their impact on overall selector performance, without predicting running time or competition score. As in earlier versions of SATzilla, pre- and backup-solvers were used in addition to the main per-instance algorithm selection mechanism. However, the decision about whether to initiate feature computation (rather than immediately running the statically determined backup solver) was made by a random forest classifier, using a subset of cheaply computable features. In the 2012 SAT Competition, this version of SATzilla used a total of 31 SAT solvers and 138 instance features; it ended up winning first prizes in two of the three main tracks and second prize in the remaining track, while also placing first in the sequential portfolio track. In the application track of this competition, it solved 6.4% more instances than the single best "interacting multi-engine" solver (which combined multiple SAT solving engines in a design that allows for richer interaction than found in a per-instance algorithm selector like SATzilla), 11.8% more instances than the

best conventional, "single-engine" solver, and only 6.5% fewer instances than the virtual best solver (a hypothetical, perfect selector).

12.3.2. ISAC, 3S and CSHC

ISAC (Instance-Specific Algorithm Configuration) uses a clustering approach to perform algorithm selection [KMST10]. Specifically, ISAC clusters the instances in the given training set based on their features, using the g-means algorithm in combination with a post-processing scheme that reassigns instances from clusters whose size falls below a user-defined threshold. For each cluster, the cluster centre and the best-performing solver (from the given set of component solvers) are stored; given a new problem instance, ISAC determines the nearest cluster centre (using a 1-nearest neighbour approach) and runs the solver associated with it. ISAC was originally designed to combine algorithm configuration and per-instance selection, by automatically configuring a single, parametric solver for each cluster (see Section 12.4.1). Initial results for SAT and two other problems (set covering and mixed integer programming) were promising, but limited in scope; in the case of SAT, ISAC was applied to a single local-search-based SAT solver, SAPS [HTH02], on which it showed substantial speedups over the default configuration on several benchmarks [KMST10]. Later, used as a pure per-instance algorithm selector in combination with a large number of high-performance SAT solvers, ISAC was demonstrated to achieve substantial performance improvements compared to the single best solver on a broad set of random SAT instances from several SAT competitions, and more modest improvements on hand-crafted and industrial SAT instances; however, in all cases, performance in terms of PAR-10 scores was more than a factor of 9.9 worse than that of the virtual best solver [CMMO13].

The clustering approach used by ISAC also underlies 3S (semi-static solver schedules), which combines per-instance algorithm selection with sequential algorithm scheduling [KMS+11]. Like ISAC, 3S uses the g-means algorithm for clustering problem instances. For a given instance i to be solved, it then uses a k-nearest neighbour approach to determine a set of similar training instances and selects the algorithm with the best performance on this instance set. This mechanism is enhanced by weighting algorithm performance on a reference instance i' by the distance of i' to i, and by adapting the neighbourhood size k based on the features of i. 3S incorporates the per-instance algorithm selector thus obtained into a static algorithm schedule, determined using mixed integer programming. While this approach is conceptually similar to the pre-solver schedules used in SATzilla from 2007 onwards, it offers considerably more flexibility in the design of sequential schedules. On random SAT instances, 3S was reported to reduce the gap between the 2009 version of SATzilla (which then constituted the state of the art for these types of instances) and the virtual best solver (VBS) (a hypothetical, perfect per-instance algorithm selector) by 57%. Unlike SATzilla, 3S participated in the 2011 SAT Competition, where it showed impressive performance, winning 2 gold, 2 silver and 3 bronze medals. In a comparison based on scenarios from the 2012 SAT Challenge, 3S was reported to obtain PAR-10 scores between 49% and 81% worse than those reached by the 2012 version of SATzilla [MSSS13].

Like ISAC and 3S, the CSHC (cost-sensitive hierarchical clustering) algorithm selection system is based on the idea of feature-based partitioning of problem instance space into clusters [MSSS13]. However, instead of unsupervised clustering, as used in ISAC and 3S, CSHC creates these clusters using a supervised learning algorithm. Effectively, it thus constitutes a cost-sensitive approach for constructing decision trees (we note that all decision tree algorithms determine hierarchical rules for "clustering" the training examples stored in each leaf node). The reader may recall that more recent versions of SATzilla also relied on (random forests of) cost-sensitive decision trees; however, the approaches are different. SATzilla uses decision trees to predict which of two algorithms will be faster, followed by voting across quadratically many such classifiers to choose a single algorithm; CHSC's decision trees directly predict the best algorithm to run. More specifically, starting with all instances in the given training set, CSHC recursively partitions a given "cluster" of instances into two "sub-clusters", based on a single feature value, such that the performance of the best-performing algorithm in each "sub-cluster" is optimized. When asked to solve a new instance i, CSHC first runs the 3S static algorithm schedule for a fixed fraction (10%) of the overall time budget. If this does not solve the instance, it computes features and uses the learned decision tree to identify a leaf node and thereby the algorithm that performed best on that "cluster" of instances. CSHC was reported to achieve slightly better performance than the 2012 version of SATzilla on a number of scenarios based on the 2012 SAT Challenge, although the observed performance differences were reported not to be statistically significant [MSSS13].

12.3.3. SNNAP

A combination of supervised and unsupervised learning forms the basis for SNNAP (solver-based nearest neigbour for algorithm portfolios) [CMMO13]. Specifically, SNNAP uses random forests to predict the running times of individual solvers; these predictions then provide the basis for identifying instances from the given training set that are similar to the instance to be solved. Instance similarity is assessed based on a (small) fixed number of best solvers known or predicted to perform best on a given instance, using Jaccard distance. The solver to be run is then selected as the one predicted to perform best on the k nearest instances in the given training set to the instance to be solved, for a user-defined value of k. In the experiments reported by Collautti et al., $k = 60$ and similarity calculations are based on the three best solvers for any given instance.

SNNAP has been demonstrated to show substantially stronger performance than ISAC on large collections of instances collected from SAT competitions between 2006 and 2012 [CMMO13]. A comparison against a 2015 internal development version of SATzilla, which is a minor variant of the 2011 version described previously, revealed that on 5 of 7 scenarios based on the 2011 SAT Competition and the 2012 SAT Challenge, SNNAP performed significantly worse than SATzilla, with statistically tied and slightly better performance, respectively, on the two remaining scenarios [LHHS15].

12.3.4. AutoFolio

As is evident from the previous sections, when designing an algorithm selection system, many design choices have to be made. These choices can have substantial impact on performance, and what works best generally depends on the given use case. Therefore, following the previously outlined programming by optimization approach, AutoFolio combines a highly parametric framework for algorithm selection with a general-purpose algorithm configurator, which is used to instantiate this framework in such a way that performance is optimized for a given use case (such as a specific set or distribution of SAT instances [LHHS15]). AutoFolio is not restricted to SAT, but can be applied to arbitrary algorithm selection problems.

The underlying algorithm selection framework, called claspfolio 2, was originally developed in the context of algorithm selection for answer set programming (ASP) and incorporates techniques and ideas from a broad range of high-performance per-instance algorithm selection approaches [HLS14]. This framework encompasses strategies from a broad range of algorithm selection systems, including 3S [KMS+11], ISAC [KMST10], SATzilla [XHHLB08, XHHLB12], and SNNAP [CMMO13]; for each selection approach, AutoFolio considers subsidiary design choices and parameter settings. Furthermore, several techniques for preprocessing the training data from a given algorithm selection scenario and for constructing pre-solver schedules can be configured independently from the selection approach. Automated configuration of this flexible framework is performed using SMAC [HHLB11], in combination with cross-validation on the given training set to robustly evaluate the performance of candidate configurations.

AutoFolio has been empirically evaluated against other algorithm selection techniques across a broad range of algorithm selection scenarios, including several benchmark sets derived from SAT competitions [LHHS15]. In those experiments, AutoFolio was found to be the only approach that achieved state-of-the-art performance for all benchmarks; notably, for each benchmark scenario, it was statistically tied with or exceeded the performance of the best of SNNAP, ISAC and SATzilla. However, there remained a considerable gap to the performance bounds obtained from the virtual best solvers for these algorithm selection scenarios (e.g., a factor of 8.8 in PAR-10 score between the AutoFolio selector and the VBS, compared to a factor of 1.8 between the single best solver and the AutoFolio selector on the SAT12-INDU benchmark), indicating room for further improvements in automated algorithm selection.

12.3.5. The Sparkle SAT Challenge

Solver competitions, such as the quasi-annual SAT solver competitions and races, serve several purposes. Aside from providing an objective, neutral performance assessment of the participating algorithms, they also provide a powerful incentive for improving the state of the art in solving the problem in question. Unfortunately, as discussed in Section 12.2.4, traditional competition designs assess – and incentivize improvement of – broad-spectrum performance of fairly monolithic solvers. This becomes an issue for problems such as SAT, where the overall

state of the art is represented by a set of algorithms, and more precisely for algorithm selectors, since they leverage the performance complementarity in such a solver set, rather than simply benefiting from the performance of the fastest single algorithm.

Therefore, just as the Configurable SAT Solver Challenges built the existence of powerful, general-purpose algorithm configurators into the design of a competition, the Sparkle SAT Challenge is predicated on the ready availability of per-instance algorithm selection systems for SAT. The key idea underlying this competition, which was held as part of the 2018 FLoC Olympic Games, was to let competitors submit SAT solvers that would be integrated into a per-instance algorithm selector constructed using AutoFolio [LHHS15]. Participating solvers would then be evaluated based on the degree to which they contribute to the overall performance of the selection system thus obtained [LH18]. To make it easier for participants to optimize the contributions of their solvers, and hence not only their standing in the competition, but also the overall performance of the final algorithm selector, daily evaluations (on a set of training instances) were conducted and published during a leader-board phase.

The Sparkle SAT Challenge was conducted using a newly designed software platform called Sparkle, which aims to make the construction and evaluation of algorithm selectors more easily accessible to solver developers and practitioners. It used the well-known feature extractor by Xu et al. [XHHLB12] in combination with the state-of-the-art AutoFolio selector construction system discussed in Section 12.3.4. Solver contributions to overall selector performance were assessed using relative marginal contribution (RMC), a normalised version of marginal contribution [XHHLB12]. More sophisticated approaches exist, but would have been computationally too expensive to use in the competition [see, e.g.,, FKM+16, KFM+18]. While official results were based on RMC to an actual selector obtained from AutoFolio, RMC to the virtual best solver over the same set of component algorithms was also reported. It is worth noting that the Sparkle system used to run the challenge not only fully automates the construction of the selector, but also the comprehensive evaluation of component solver and selector performance, which is compiled into a detailed technical report generated by Sparkle.

The results from the Sparkle SAT Challenge, as presented during the 2018 FLoC Olympic Games [LH18], gave rise to three main findings.

First, as observed consistently in the literature and in other competitions, using per-instance algorithm selection, performance complementarity between SAT solvers was leveraged, leading to substantial performance improvements over the single best solver (SBS). These improvements tended to be much larger than those between the SBS and the runners-up in terms of stand-alone solver performance. The difference in stand-alone performance between the SBS and the runner-up was less than 1% in terms of PAR2 score, while the automatically constructed selector was more than 7.5% better than the SBS, despite the fact that the test instances differed substantially from the training instances used as a basis for constructing the selector. This highlights the continued importance of per-instance algorithm selection for achieving state-of-the-art performance in solving SAT.

Second (as previously observed by Xu et al. [XHHLB12]) the stand-alone per-

formance of SAT solvers tended to be uncorrelated with their contributions to a state-of-the-art per-instance selector built from them, and likewise uncorrelated with their contributions to the VBS. This suggests that conventional competitions may not be the most effective way of improving the state of the art in SAT solving, since they motivate solver developers to optimize stand-alone performance. In the Sparkle SAT Challenge, the solver ranked third in terms of stand-alone performance had a relative marginal contribution to selector performance that was over five times lower than that of the solver ranked last.

Third, when given the opportunity to repeatedly evaluate the contributions of their solvers to overall selector performance, solver developers appear to have succeeded in maximizing those contributions and, as a result, also overall selector performance. We attribute this partly to the easy and frictionless access to cutting-edge selector construction methods offered by the Sparkle platform. In particular, selector performance on training data accessible to participants in the challenge improved by a factor of more than 2 in terms of PAR2 score.

12.4. Related approaches

Automated algorithm configuration and per-instance algorithm selection, although very widely studied in the literature, are not the only meta-algorithmic design techniques. In this section, we briefly outline additional techniques, many of which have been applied to SAT.

12.4.1. Combining algorithm configuration and algorithm selection

Algorithm configuration and algorithm selection have different strengths. Algorithm configuration is good at searching enormous parameter spaces to find settings that yield peak average performance, at a cost of hours or days of offline compute time. Algorithm selection is good at exploiting variability in the running times of different solvers (or different configurations) *within* instance distributions of interest, and can select the best of a small, fixed set of solvers quickly enough to be used online. In many practical applications, the algorithm configuration setting is closer to the needs of a typical practitioner than the algorithm selection setting: notably, it can be much easier to maintain a single algorithm codebase with parameterized components than different codebases for multiple solvers. However, sometimes different codebases arise separately (as in the SAT competition setting), and sometimes practitioners face heterogeneous sets of problem instances, on which no single configuration yields sufficiently strong average performance. In order to get the best of both worlds, we can use algorithm configuration to obtain a set of complementary configurations that can be fruitfully combined using algorithm selection.

One simple approach to combining algorithm configuration and algorithm selection is to cluster a heterogeneous set of problem instances into a set of homogeneous subsets, to run configuration on each of these subsets to obtain specialized configurations, and then to treat these specialized configurations as the solvers to be combined with a per-instance algorithm selector. This is the approach followed by *instance-specific algorithm configuration (ISAC [KMST10])*.

A more sophisticated way to combine configuration and selection is to use configuration to iteratively identify configurations that are maximally complementary to an existing set. *Hydra* [XHLB10] follows this idea, building on an approach first proposed by Leyton-Brown et al. [LBNA+03]. Hydra starts from an empty portfolio and then iteratively adds a new configuration that maximally improves the oracle performance of the configurations in the portfolio. Because the oracle performance of an algorithm portfolio is a submodular function, this approach is guaranteed to yield portfolios whose oracle performance is within a factor of $1 - 1/e$ from optimal.

12.4.2. Algorithm schedules

While per-instance algorithm selection often works very well, it depends on the availability of cheaply computable instance features that jointly correlate well with algorithm running time. An alternative strategy that does not depend on the existence of such features is building an *algorithm schedule* that runs several algorithms in a fixed or adaptive sequence. Algorithm schedules work when many solvers have high variance in their running time distributions across instances (i.e., solve some instances very quickly and others very slowly) and when different instances are easy for different solvers.

Indeed, algorithm schedules can also be useful even when informative instance features do exist, because they offer a way to solve easy instances without computing features. This matters particularly when performance is measured by making a multiplicative rather than additive comparison to optimal, because feature computation can dominate running time on easy instances. It is for this reason that several per-instance algorithm selection systems use pre-solver schedules, as described in Section 12.3.

Algorithm schedules can be constructed using various techniques. They can be defined by hand, built by local search in the joint space of algorithms and running times (as done for determining the pre-solving schedules used by SATzilla [XHHLB08]), or by means of mixed integer programming [KMS+11] or answer set programming [HKLS15]. It is also possible to start with an empty schedule and use a Hydra-like approach to greedily append one configuration at a time to the current schedule, along with a time budget for which it should be allowed to run [SSHH15]. (In contrast to Hydra, the objective function optimized by algorithm configuration in this case is performance gain of the schedule *per additional time spent*, but otherwise the approach is similar.)

12.4.3. Parallel portfolios

So far, this chapter has focused on approaches that require only a single processor core. Of course, modern computing is increasingly parallel, particularly as cloud resources become available. Parallel portfolios are based on the idea of running multiple sequential solvers in parallel on a given problem instance to be solved; if these component solvers have complementary strength, this can lead to an improvement in performance in terms of wall-clock time and, in some cases, even in terms of overall CPU time, when solving decision problems such as SAT.

Parallel portfolios can thus be used to construct solvers that effectively exploit parallel computing resources without the well-known difficulties associated with developing intrinsically parallel solvers.

Even simple parallel portfolios can be very powerful. To give one notable example, in the 2011 SAT Competition, the solver ppfolio [Rou11] won 11 medals in the parallel track and 5 medals in the sequential track (where its parallelism was realised by task switching) by following an incredibly straightforward strategy: running the best solvers from several categories of the 2009 SAT Competition in parallel. This approach was effective, because the solvers forming the parallel portfolio performed well individually and complemented each other well. Roughly speaking, automated methods for constructing parallel portfolios also aim to identify an optimal compromise between complementarity and individual strength.

Algorithm configuration and per-instance selection can both be extended to the parallel case. Algorithm configuration can be used to create strong parallel solvers based only on the parametric code of a sequential solver [LHLBS17]. Once more, Hydra's greedy approach is effective: starting with an empty portfolio (on zero processor cores), greedily add the configuration to run on the next processor core that most improves performance when run alongside the existing portfolio. Algorithm selection can also be generalized to produce a parallel portfolio of solvers rather than a single solver [LHH15]. In particular, a rather broadly applicable approach for parallelizing an existing algorithm selection approach is to adapt its underlying machine learning model to rank the solvers, and then simply to select the top-ranked K solvers to concurrently run on K processor cores.

12.4.4. Parameter control

Just as different solvers and solver configurations exhibit strong performance on different problem instances, so it is the case that different strategies work best during different phases of solving a single instance. For example, local search algorithms need to effectively switch between exploration and exploitation phases, and tree search algorithms may benefit from using different heuristics at different depths of the search tree or as a function of the number of clauses already learned. This observation gives rise to the meta-problem of adapting the parameters of a given algorithm online as a function of the current search state. While algorithm configuration can be formulated as a black-box optimization problem and algorithm selection as a supervised learning problem, *parameter control* is a more complex problem akin to reinforcement learning [LL00, LL01, EFW+02, BC12, AN16, BBHL19].

12.4.5. Parameter importance analysis

It is important to complement the automated methods for constructing strong solvers discussed up to this point with analysis techniques that can yield insights about what components were responsible for observed performance improvements. For algorithm configuration, it is possible to use model-based approaches to detect which parameters have an important influence on performance. Techniques

can be based on forward selection [LBNS09, HHLB13]; based on a decomposition of the variance of predicted performance into components attributable to each single parameter and combination of (two or more) parameters [HHLB14]; or based on local effects around the best found configuration [BMLH18]. Alternatively, ablation analysis can be used to assess the relative contributions of individual parameters to observed performance differences between algorithm configurations [FH16]. It is also possible to perform a model-based variant of such an ablation analysis, which means that the analysis can be performed without executing additional runs of the algorithm under study [BLE+17].

12.4.6. Using algorithm configuration to find bugs in solvers

As mentioned at the beginning of this chapter, algorithm configuration has uses beyond performance optimization; one of these is the identification of parameter configurations that lead to errors, and especially those that do so quickly. Algorithm configuration optimizes a user-defined performance metric; in SAT solving, common choices are minimizing the number of timeouts for a given cutoff time and minimizing (penalized) average running time. A subtlety is how one should count runs that terminate quickly but either with no result (e.g., due to a segmentation fault) or with an incorrect result (e.g., UNSAT for a satisfiable instance). When seeking to produce good average performance, such runs are typically given a terrible score (e.g., the timeout multiplied by a large constant) in order to drive the configurator towards areas of the configuration space that do not cause erroneous behaviour. In contrast, blindly minimizing average measured running time actually causes the configurator to find configurations that terminate as quickly as possible; often this yields buggy configurations that *break* quickly [ELH19]. This can be viewed as a tool for automatically finding whitebox tests that expose problems quickly [HHLB10, ML16].

12.5. Conclusions and open challenges

The meta-algorithmic techniques described here have already improved the state of the art in SAT solving for various types of applications [see, e.g.,, HBHH07, NFLB17], but there still remain several open challenges in terms of methods, tools and applications.

Regarding methods, one of the biggest challenges is to develop an approach that works robustly, even for SAT instances that do not resemble the instances seen at training time. It would be highly desirable to have available an effective configuration method for highly heterogeneous instances, which could be used to identify better default configurations. However, in many cases, there simply does not exist a single configuration that performs well across instances. Instead, we believe that it might be more promising to develop a robust approach for selecting the best out of a small number of automatically-determined default configurations. An interesting research challenge consists of identifying such robust defaults based on a limited number of training instances. Ways to effectively integrate domain knowledge from solver developers would be extremely helpful towards this goal.

In terms of tools, it would be helpful to make it easier for developers and users of SAT solvers to take advantage of meta-algorithmic tools and the performance improvements they afford. Currently, considerable expertise is required to determine, e.g., which parameters should be configured for which kinds of instance sets, and to automatically build robust solvers for application to new instances, using approaches such as Hydra. This high barrier to entry could be lowered, at least in part, by designing a highly parametric CDCL solver, with a rich configuration space that covers most of the techniques used in modern CDCL-based SAT solvers, similar to the way that SATenstein [KXHLB09, KXHLB16] covers a substantial part of the space of stochastic local search solvers. Furthermore, there remains a need for making automatic algorithm configuration and per-instance selection methods easier to use for practitioners.

We see significant room for further innovation in solver competitions, which have historically been a significant driver of new work in the field. We see substantial value in competitions that deeply integrate both automated algorithm configuration (as done in the Configurable SAT Solver Challenges) and per-instance algorithm selection (as done in the Sparkle SAT Challenge). We are also excited about recent work that has shown how ideas from coalitional game theory can be used to derive new evaluation metrics for such competitions [FKM+16, KFM+18]. We hope that these metrics can used in future competitions (perhaps alongside existing alternatives) to further improve the way in which innovative ideas in solver design are recognized.

Finally, as is evident both from the literature and from competitions, algorithm configuration and algorithm selection have the potential to produce performance gains in solving virtually any application problem. Specifically, we see significant opportunities in applying meta-algorithmic technologies to improve SMT solvers, which play a key role in some of the most important applications of SAT-related techniques in the areas of hardware and software verification.

References

[AMS14] C. Ansótegui, Y. Malitsky, and M. Sellmann. Maxsat by improved instance-specific algorithm configuration. In *Proceedings of the Twenty-eighth National Conference on Artificial Intelligence (AAAI'14)*, pages 2594–2600. AAAI Press, 2014.

[AMST15] C. Ansótegui, Y. Malitsky, M. Sellmann, and K. Tierney. Model-based genetic algorithms for algorithm configuration. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 733–739, 2015.

[AN16] S. Adriaensen and A. Nowé. Towards a white box approach to automated algorithm design. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 554–560. AAAI Press, 2016.

[AST09] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09)*, volume 5732 of

Lecture Notes in Computer Science, pages 142–157. Springer Verlag, 2009.

[BBBK11] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 25th International Conference on Advances in Neural Information Processing Systems (NIPS'11)*, pages 2546–2554, 2011.

[BBHL19] A. Biedenkapp, H. F. Bozkurt, F. Hutter, and M. Lindauer. Towards white-box benchmarks for algorithm control. In *IJCAI 2019 DSO Workshop*, August 2019.

[BC12] R. Battiti and P. Campigotto. An investigation of reinforcement learning for reactive search optimization. In Youssef Hamadi, Eric Monfroy, and Frédéric Saubion, editors, *Autonomous Search*, pages 131–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[BH07] D. Babić and F. Hutter. Spear theorem prover. Solver description, SAT competition 2007, 2007.

[Bie13] A. Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 51–52. University of Helsinki, 2013.

[BLE+17] A. Biedenkapp, M. Lindauer, K. Eggensperger, C. Fawcett, H. Hoos, and F. Hutter. Efficient parameter importance analysis via ablation with surrogates. In *Proceedings of the Conference on Artificial Intelligence (AAAI'17)*, pages 773–779. AAAI Press, 2017.

[BMLH18] A. Biedenkapp, J. Marben, M. Lindauer, and F. Hutter. Cave: Configuration assessment, visualization and evaluation. In *Proceedings of the International Conference on Learning and Intelligent Optimization (LION'18)*, 2018.

[Bre01] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[BS12] A. Balint and U. Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 16–29. Springer Verlag, 2012.

[BYC14] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*, pages 115–123. Omnipress, 2014.

[CFH08] M. Chiarandini, Chris Fawcett, and H. Hoos. A modular multiphase heuristic solverfor post enrolment course timetabling. In *Proceedings of the Seventh International Conference on the Practice and Theory of Automated Timetabling*, 2008.

[CMMO13] M. Collautti, Y. Malitsky, D. Mehta, and B. O'Sullivan. SNNAP: Solver-based nearest neighbor for algorithm portfolios. In *Machine Learning and Knowledge Discovery in Databases (ECML/PKDD'13)*, volume 8190 of *Lecture Notes in Computer Sci-*

ence, pages 435–450. Springer Verlag, 2013.

[DP13] T.-T. Duong and D.-N. Pham. gNovelty+GC: Weight-Enhanced Diversification on Stochastic Local Search for SAT. In *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 49–50. University of Helsinki, 2013.

[DSH15] T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 3460–3468, 2015.

[EFW+02] S.L. Epstein, E.C. Freuder, R. Wallace, A. Morozov, and B. Samuels. The adaptive constraint engine. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 525–542, London, UK, UK, 2002. Springer-Verlag.

[ELH19] K. Eggensperger, M. Lindauer, and F. Hutter. Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research*, 64:861–893, 2019.

[ES04] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Verlag, 2004.

[FH16] C. Fawcett and H.H. Hoos. Analysing differences between algorithm configurations through ablation. *Journal of Heuristics*, 22(4):431–458, 2016.

[FKE+15] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems (NIPS'15)*, 2015.

[FKM+16] A. Fréchette, L. Kotthoff, T.P. Michalak, T. Rahwan, H. Hoos, and K. Leyton-Brown. Using the shapley value to analyze algorithm portfolios. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16)*, pages 3397–3403. AAAI Press, 2016.

[GKNS07] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning*, pages 260–265. 2007.

[HBHH07] F. Hutter, D. Babić, H. Hoos, and A. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 27–34, 2007.

[HHLB10] F. Hutter, H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proceedings of the Seventh International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR'10)*, volume 6140 of *Lecture Notes in Computer Science*, pages 186–202. Springer Verlag, 2010.

[HHLB11] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of*

the Fifth International Conference on Learning and Intelligent Optimization (LION'11), volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer Verlag, 2011.

[HHLB13] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Identifying key algorithm parameters and instance features using forward selection. In *Proceedings of the 7th International Conference on Learning and Optimization (LION-7)*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, January 2013.

[HHLB14] F. Hutter, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proceedings of International Conference on Machine Learning 2014 (ICML'14)*, pages 754–762, June 2014.

[HHLBS09] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[HHS07] F. Hutter, H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*, pages 1152–1157. AAAI Press, 2007.

[HJS18] M. Heule, M. Järvisalo, and M. Suda, editors. *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2018.

[HKLS15] H. Hoos, R. Kaminski, M. Lindauer, and T. Schaub. aspeed: Solver scheduling via answer set programming. *Theory and Practice of Logic Programming*, 15:117–142, 2015.

[HLB+17] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. Hoos, and K. Leyton-Brown. The configurable SAT solver challenge (CSSC). *Artificial Intelligence Journal (AIJ)*, 243:1–25, February 2017.

[HLS14] H. Hoos, M. Lindauer, and T. Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, 14:569–585, 2014.

[Hoo12] H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[HTH02] F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: efficient dynamic local search for SAT. In *Proceedings of the 8th International conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer Verlag, 2002.

[HXHLB14] F. Hutter, L. Xu, H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods and evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[JBRS12] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–94, 2012.

[JSW98] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492, 1998.

[KFM+18] L. Kotthoff, A. Fréchette, T.P. Michalak, T. Rahwan, H. Hoos, and K. Leyton-Brown. Quantifying algorithmic improvements over time. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, pages 5165–5171, 2018.

[KLBL17] R. Kleinberg, K. Leyton-Brown, and B. Lucier. Efficiency through procrastination: Approximately optimal algorithm configuration with runtime guarantees. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 2017.

[KLBLG19] R. Kleinberg, K. Leyton-Brown, B. Lucier, and D. Graham. Procrastinating with confidence: Near-optimal, anytime, adaptive algorithm configuration. *arXiv:1902.05454 [cs.AI]*, 2019.

[KMS+11] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm selection and scheduling. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.

[KMST10] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC - instance-specific algorithm configuration. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*, pages 751–756. IOS Press, 2010.

[Knu15] D.E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, Reading, MA, USA, 2015.

[KXHLB09] A. KhudaBukhsh, L. Xu, H.H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of the 22th International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 517–524, 2009.

[KXHLB16] A. KhudaBukhsh, L. Xu, H.H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence Journal (AIJ)*, 232:20–42, March 2016.

[LBNA+03] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2470 of *Lecture Notes in Computer Science*, pages 899–903. Springer Verlag, 2003.

[LBNS09] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009.

[LH18] C. Luo and H.H. Hoos. Sparkle SAT Challenge 2018, 2018. http://ada.liacs.nl/events/sparkle-sat-18. Last accessed on 15 June 2019.

[LHH15] M. Lindauer, H. Hoos, and F. Hutter. From sequential algorithm selection to parallel portfolio selection. In *Proceedings of the Ninth International Conference on Learning and Intelligent Optimization (LION'15)*, volume 8994 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 2015.

[LHHS15] M. Lindauer, H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial*

Intelligence Research, 53:745–778, August 2015.

[LHLBS17] M. Lindauer, H. Hoos, K. Leyton-Brown, and T. Schaub. Automatic construction of parallel portfolios via algorithm configuration. *Artificial Intelligence Journal (AIJ)*, 244:272–290, March 2017.

[LIDLSB11] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package, iterated race for automatic algorithm configuration. Technical report, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[LL00] M.G. Lagoudakis and M.L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML'00)*, pages 511–518, 2000.

[LL01] M.G. Lagoudakis and M.L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *Electronic Notes in Discrete Mathematics*, volume 9, pages 344–359, 2001.

[LMS02] H. R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In *Handbook of Metaheuristics*, pages 321–353. 2002.

[Man13] N. Manthey. The SAT solver RISS3G at SC 2013. In *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 72–73. University of Helsinki, 2013.

[ML16] N. Manthey and M. Lindauer. Spybug: Automated bug detection in the configuration space of SAT solvers. In *Proceedings of the International Conference on Satisfiability Solving (SAT'16)*, pages 554–561, 2016.

[MS14] N. Manthey and P. Steinke. Too many rooks. In *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 97–98. University of Helsinki, 2014.

[MSSS13] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI'13)*, pages 608–614. AAAI Press, 2013.

[NFLB17] N. Newman, A. Fréchette, and K. Leyton-Brown. Deep optimization for spectrum repacking. *Communications of the ACM*, 61(1):97–104, 2017.

[NLBA+03] E. Nudelman, K. Leyton-Brown, G. Andrew, C. Gomes, J. McFadden, B. Selman, and Y. Shoham. Satzilla 0.9. Solver description, International SAT Competition, 2003.

[NLBH+04] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: beyond the clauses-to-variables ratio. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 438–452. Springer Verlag, 2004.

[NLBMS17] N. Newman, K. Leyton-Brown, P. Milgrom, and I. Segal. Assessing economic outcomes in simulated reverse clock auctions for radio spectrum. *arXiv:1706.04324 [cs.GT]*, 2017.

[PCLHS17] L. Pérez Cáceres, M. López-Ibánez, H. Hoos, and T. Stützle. An experimental study of adaptive capping in irace. In *Proceedings of the 11th International Conference on Learning and Intelligent Optimization (LION 11)*, volume 10556 of *Lecture Notes in Computer Science*, pages 235–250. Springer, 2017.

[Ric76] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[Rou11] O. Roussel. Description of ppfolio. Solver description, SAT competition 2011, 2011.

[SBH05] L. Simon, D. Le Berre, and E. Hirsch. The SAT2002 competition report. *Annals of Mathematics and Artificial Intelligence*, 43:307–342, 2005.

[SH79] J. Schmee and G. J. Hahn. A simple method for regression analysis with censored data. *Technometrics*, 21(4):417–432, 1979.

[SLA12] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Proceedings of the 26th International Conference on Advances in Neural Information Processing Systems (NIPS'12)*, pages 2960–2968, 2012.

[SLS12] B. Silverthorn, Y. Lierler, and M. Schneider. Surviving solver sensitivity: An ASP practitioner's guide. In *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, volume 17, pages 164–175. Leibniz International Proceedings in Informatics (LIPIcs), 2012.

[SSHH15] J. Seipp, S. Sievers, M. Helmert, and F. Hutter. Automatic configuration of sequential planning portfolios. In *Proceedings of the Twenty-nineth National Conference on Artificial Intelligence (AAAI'15)*, pages 3364–3370. AAAI Press, 2015.

[THHLB13] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*, pages 847–855. ACM press, New York, NY, USA, 2013.

[VFG+13] M. Vallati, C. Fawcett, A. Gerevini, H. Hoos, and A. Saetti. Automatic generation of efficient domain-optimized planners from generic parametrized planners. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SOCS'14)*. AAAI Press, 2013.

[WGS18a] G. Weisz, A. Györgi, and C. Szepesvári. CapsAndRuns: An improved method for approximately optimal algorithm configuration. *ICML 2018 AutoML Workshop*, 2018.

[WGS18b] G. Weisz, A. Györgi, and C. Szepesvári. LeapsAndBounds: A method for approximately optimal algorithm configuration. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, pages 5254–5262, 2018.

[XHHLB07] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Satzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *Lec-*

ture Notes in Computer Science, pages 712–727. Springer Verlag, 2007.

[XHHLB08] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, June 2008.

[XHHLB09] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla2009: an automatic algorithm portfolio for sat. Solver description, SAT competition 2009, 2009.

[XHHLB12] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Evaluating component solver contributions in portfolio-based algorithm selectors. In *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 228–241. Springer Verlag, 2012.

[XHLB07] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hierarchical hardness models for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *Lecture Notes in Computer Science*, pages 696–711. Springer Verlag, 2007.

[XHLB10] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, pages 210–216. AAAI Press, 2010.