Sparkle: Toward Accessible Meta-Algorithmics for Improving the State of the Art in Solving Challenging Problems

Koen van der Blom, Holger H. Hoos, Chuan Luo, and Jeroen G. Rook

Abstract—**Many fields of computational science advance through improvements in the algorithms used for solving key problems. These advancements are often facilitated by benchmarks and competitions that enable performance comparisons and rankings of solvers. Simultaneously, meta-algorithmic techniques, such as automated algorithm selection and configuration, enable performance improvements by utilizing the complementary strengths of different algorithms or configurable algorithm components. In fact, meta-algorithms have become major drivers in advancing the state of the art in solving many prominent computational problems. However, meta-algorithmic techniques are complex and difficult to use correctly, while their incorrect use may reduce their efficiency, or in extreme cases, even lead to performance losses. Here, we introduce the Sparkle platform, which aims to make meta-algorithmic techniques more accessible to nonexpert users, and to make these techniques more broadly available in the context of competitions, to further enable the assessment and advancement of the true state of the art in solving challenging computational problems. To achieve this, Sparkle implements standard protocols for algorithm selection and configuration that support easy and correct use of these techniques. Following an experiment, Sparkle generates a report containing results, problem instances, algorithms, and other relevant information, for convenient use in scientific publications.**

Index Terms—**Algorithm configuration, algorithm selection, benchmarking, competitions, meta-algorithms, software tools.**

Manuscript received 15 September 2021; revised 27 January 2022, 20 May 2022, and 29 September 2022; accepted 4 October 2022. Date of publication 17 October 2022; date of current version 1 December 2022. This work was supported in part by TAILOR, a project funded by EU Horizon 2020 Research and Innovation Programme under Grant 952215, and in part by the National Natural Science Foundation of China under Grant 62202025. *(Corresponding author: Koen van der Blom.)*

Koen van der Blom was with the Faculty of Science, Leiden University, 2311 EZ Leiden, The Netherlands. He is now with LIP6, CNRS, Sorbonne Université, 75005 Paris, France (e-mail: koen.vdblom@lip6.fr).

Holger H. Hoos is with the Faculty of Science, Leiden University, 2311 EZ Leiden, The Netherlands, also with the Department of Computer Science, RWTH Aachen University, 52062 Aachen, Germany, and also with the Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, Canada (e-mail: hh@aim.rwth-aachen.de).

Chuan Luo was with the Faculty of Science, Leiden University, 2311 EZ Leiden, The Netherlands. He is now with the School of Software, Beihang University, Beijing 100190, China (e-mail: chuanluo@buaa.edu.cn).

Jeroen G. Rook was with the Faculty of Science, Leiden University, 2311 EZ Leiden, The Netherlands. He is now with the Data Management and Biometrics (DMB), University of Twente, 7522 NB Enschede, The Netherlands (e-mail: j.g.rook@utwente.nl).

This article has supplementary downloadable material available at https://doi.org/10.1109/TEVC.2022.3215013, provided by the authors.

Digital Object Identifier 10.1109/TEVC.2022.3215013

I. Introduction

Over the past decade, our ability to solve well-studied computationally challenging problems has increased substantially, as has the importance of high-performance solvers for these problems in the context of real-world applications. This can be seen, for example, in the case of the propositional satisfiability problem (SAT), one of the most prominent NP-complete combinatorial decision problems, which has important real-world applications in hard- and software verification (e.g., [1]).

Part of these advances have been incentivized by benchmarks and competitions, and the desire to outperform other solvers. Meanwhile, meta-algorithmic techniques, such as automated algorithm configuration (AAC) and automated algorithm selection (AAS), are also being used increasingly to advance the state of the art in solving a broad range of problems from AI and related areas. By carefully choosing parameter settings and algorithm components, AAC techniques are often able to achieve substantial performance gains. This has been shown, e.g., for SAT [2], [3], MaxSAT [4], the machine reassignment problem [5], mixed-integer programming (MIP) [2], [6], automated planning [7], and supervised machine learning [8], [9]. AAS techniques leverage the fact that different solvers perform best for different problem instances, and achieve better performance than standalone solvers by selecting from a given portfolio of solvers one (or more) solvers that are most suitable for a given instance [10]. AAS has seen successful application on a broad range of widely studied problems, including SAT [11], MaxSAT [12], MIP [13], [14], constraint programming (CP) [15], and AI planning [16].

Due to their substantial impact on performance, meta-algorithmic techniques, such as AAS and AAC, have become increasingly important for benchmarking and competitions. In both settings, traditional ranking and comparison schemes commonly represent the state of the art by relying on metrics that aggregate algorithm performance over a set of problem instances (e.g., average running time), and considering a single predefined parametrization for each algorithm (e.g., by using the default parameter settings). However, this typically fails to capture the true performance potential. AAC allows solvers to realize their performance potential by finding the best configuration for a specific set or distribution of problem instances. When all solvers are able to reach their full potential, this results in a more accurate representation of the state of the art. Similarly, as observed above in the context of AAS, the best solver is usually not the same for each instance. The true state of the art thus cannot be adequately represented by a single solver, as considered in traditional benchmarks and competitions, but requires the use of multiple solvers or solving techniques. Therefore, to more accurately assess the true state of the art, meta-algorithmic techniques, such as AAS and AAC have to be utilized. These observations have led to new types of competitions that integrate meta-algorithms, such as the configurable SAT solver challenge (CSSC) [3], and the per-instance selection-based Sparkle Challenges for SAT[1] and AI planning.[2]

However, automatically improving performance by means of meta-algorithmic techniques poses substantial additional challenges. Meta-algorithms are internally complex, and their implementations are usually not easy to use. In addition, there are several well-documented pitfalls that can easily lead to unsatisfactory results. For example, in AAC [17], letting each algorithm that is being configured measure their own running time can lead to unreliable results, because they may all use different, inconsistent, implementations to measure this. Furthermore, large-scale performance evaluations and the effective use of meta-algorithmic techniques all require substantial computational resources, often benefiting from parallel computation on high-performance compute (HPC) clusters, which introduces additional complexity. Consequently, mistakes do not just lead to poor results, but can also be expensive to rectify and cause additional environmental impact through CO_2 emissions.

Having defined the true state of the art in solving a given problem, and established the need for meta-algorithms (such as AAS and AAC) both to assess and advance it, we further observe that their use in practice (scientific as well as industry) is limited (e.g., in research, algorithms are primarily optimized manually or with simple methods, such as grid search [18]). Here, we introduce Sparkle,[3] a platform that has been designed to lower the threshold for using these meta-algorithmic techniques effectively and correctly, avoiding common pitfalls and following best practices. Naturally, this should benefit the users of meta-algorithmic techniques, who will have an easier time advancing and staying up to date with the state of the art in solving challenging computational problems. In addition, developers of meta-algorithms may benefit from increased adoption of their methods when they are made available through Sparkle. In turn, attention to improving meta-algorithmic techniques may also grow. In this work, we give a detailed overview of the steps required for applying AAS and AAC, how these are implemented in Sparkle, and which steps are made easier compared to using AAS and AAC in a stand-alone fashion, supported by a small user study. The initial version of Sparkle presented here incorporates one prominent AAS and AAC system each (AutoFolio [19] and SMAC [20], respectively). These were chosen carefully to ensure high-quality and broadly usable systems.

In the following, we first discuss how Sparkle is situated in relation to other work (Section II). Then, in Section III, we cover the core design principles behind Sparkle. Section IV introduces the first major use case, where AAS is employed with parameter-less solvers to take advantage of their complementary strengths, and to accurately assess the true state of the art. Next, Section V considers parameterized solvers, and how they can benefit from AAC in Sparkle in a relatively hassle-free way. In Section VI, we briefly discuss how Sparkle can be used to enable new types of competitions. Section VII covers additional use cases enabled by the Sparkle platform. In Section VIII, we discuss best practices and pitfalls covered by this implementation of Sparkle. Finally, in Section IX, we draw several general conclusions and briefly outline future work.

II. Related Work

In the following, we consider tools that, like Sparkle, in some way support accessibility to, assessment of and/or advancement of the state of the art in solving computationally challenging problems. For each of these, we situate contributions to these goals against those made by Sparkle and highlight the gaps Sparkle can fill.

For single- and bi-objective evolutionary algorithms, the COCO framework [21] provides a well-established benchmarking environment, in which many standard tools required for experimentation with and benchmarking of these types of algorithms are readily available. In particular, COCO includes statistical analysis tools and standardized experimentation procedures, which help to avoid common mistakes (e.g., benchmark function variation, through rotations, etc., to avoid overfitting). Additionally, experimental results submitted by users are collected, which makes comparison with other algorithms easy. Support for meta-algorithmic procedures is currently not included.

With ParadisEO [22], evolutionary computation (EC) algorithms can be constructed from components. Through integrations with the algorithm configurator irace [23] and the experimentation platform IOHexperimenter [24], ParadisEO is able to automatically construct EC algorithms and measure their performance [25]. However, an integration like this only makes the configuration procedure available to the specific application and can therefore only be used in its specific context. In this case, ParadisEO is focused on EC algorithms and similar iterative optimization algorithms (as is the IOH framework). With Sparkle, on the other hand, we aim to support the use of AAC and other meta-algorithmic design procedures as generally as possible (and it is already broadly usable, e.g., for SAT solvers, AI planners, and MIP solvers).

SPOT [26] is an R package for hyperparameter optimization (HPO) and provides a number of related tools. Particularly, it focuses on surrogate-model-based parameter optimization to reduce computational cost. In addition to the core optimization tools, SPOT also includes functionality to visually compare parameters obtained as a result of the optimization process. Whereas SPOT focuses on a special case of AAC, Sparkle

[1]https://ada.liacs.nl/events/sparkle-sat-18/

[2]https://ada.liacs.nl/events/sparkle-planning-19/

[3]Sparkle is publicly available at: https://bitbucket.org/sparkle-ai/sparkle/

aims to make a broader range of meta-algorithmic techniques accessible to its users.

Existing algorithm configuration procedures, such as SMAC [20], irace [23], ParamILS [27], and GPS [28], are readily available for use by experts in algorithm configuration, but are often challenging to use for nonexperts. This usability issue stems from their focus on the core configuration process, which is sufficient for experts in algorithm configuration, while only limited support is provided for the overall configuration process that has to be considered in practice (this is elaborated further in Section V, Fig. 2). For algorithm selection, the situation is slightly better. Recognizing that different types of algorithm selectors and settings work best in different use cases, AutoFolio [19] automatically configures an algorithm selector for a given application scenario, using SMAC. Although this simplifies part of the process of constructing a high-performance algorithm selector, AutoFolio itself is not much more accessible to nonexperts than other algorithm selection systems.

Meta-algorithmic benchmarking libraries, such as AClib [29] and ASlib [30], provide scenarios for testing and benchmarking meta-algorithmic techniques. While ASlib limits itself purely to the scenarios to compare on, AClib also includes affordances for running a number of algorithm configurators on those scenarios. Finally, AClib provides tools for producing some basic statistics and plots for the configuration scenarios it has been used to run. Whereas ASlib and AClib are designed for AAS and AAC experts, respectively, Sparkle is designed to be accessible to allow nonexperts to benefit from techniques, such as AAS and AAC. In addition, Sparkle aims to better support users by generating reports that provide more details about the process that was used to obtain a given set of results.

For machine learning pipeline design, various AutoML frameworks exist, such as Auto-sklearn [31] and AutoGluon [32]. While these frameworks are limited to machine learning, this is not the case for Sparkle, which can in principle be applied to a much broader range of computational problems. In addition, Sparkle can also be used for simple performance evaluations and comparative performance analysis, which are usually not included in AutoML frameworks.

For machine learning problems, OpenML [33] collects datasets, experiments, algorithms, and results. This enables scientists to compare their approaches on the same datasets and under the same conditions. In addition, OpenML provides a wealth of data from many different runs of each algorithm on a range of datasets. While this is of great value for benchmarking and accurate performance comparisons, it markedly differs from Sparkle, which supports the broad use of meta-algorithmic techniques.

HAL [34] aimed to support similar functionality to Sparkle, with a particular focus on automated analysis and design of algorithms, including meta-algorithms. Unfortunately, HAL turned out to be over-engineered, in the sense that, while accommodating a wide range of functions at its release, it was difficult to install and use, resulting in limited adoption. Sparkle aims to avoid these issues, by ensuring that simple tasks are easy to carry out. To avoid an unnecessarily complex design, it is focused around simple, modular command scripts that are easy to use. Behind the scenes, these scripts may still call more complex classes and structures, but those too are all designed to have an easily understood interface that is called by the command scripts. To aid installation, Sparkle aims to automate the installation process to the largest extend possible, e.g., by automatically installing dependencies. Furthermore, unlike HAL, Sparkle automatically produces detailed reports.

III. Design

One of the main design principles underlying Sparkle is that it should be easy to get simple things done (e.g., adding a solver), and as easy as possible to achieve more complex goals (e.g., configuring an algorithm). Naturally, what is easy to do also depends on the level of expertise of different users with regard to the meta-algorithmic procedures made available through Sparkle. Here, we consider an expert user to be someone that is familiar with the specific meta-algorithm they want to use, and a nonexpert someone who is not. We note that, for this initial implementation of Sparkle, we expect nonexperts to be familiar with standard computer science concepts, such as programming and command line interfaces (CLIs). While Sparkle especially targets nonexperts, easier-to-use meta-algorithms should also be of benefit to expert users. These considerations resulted in a design that is primarily based on a relatively small set of commands that are largely self explanatory. By combining these commands, scripts can be written to specify and run experiments. For expert users, this should result in scripts that broadly follow the high-level processes they are already familiar with, although with a much reduced need to specify details, while nonexpert users will benefit from the scaffolding and abstraction afforded by this approach. For example, in algorithm selection, the instances, solvers, and feature extractor(s) are added to the system, after which the features and performance data are computed, and the portfolio selector and the report are constructed. This shows the usual process to construct a portfolio selector and makes clear what needs to be provided by the user, without requiring detailed instructions from the user on how to actually construct the selector, or on how to evaluate it.

To achieve a system that is as easy and safe to use as possible, we broadly consider the following three categories of usability throughout Sparkle.

1) Efficiency.
2) Correctness.
3) Understandability.

Efficiency primarily concerns how simple the process is for users to achieve their goals with Sparkle. This includes the commands they need to call (e.g., `add_solver`) and the input they need to provide (e.g., problem instances). Each of these interactions should be as easy and clear as possible. To this end, the number of necessary interactions is minimized, and also kept as simple as possible (e.g., the `run_solvers` command does not require any arguments because, by default, it simply runs all solver-instance combinations for which no performance data is available yet), while keeping options available for flexibility and more sophisticated functionality.

Correctness focuses on the correct execution of all processes initiated by the user. Here, the focus is on the internal operation of Sparkle. Since complex meta-algorithmic techniques are provided to potentially nonexpert users, support is needed in their correct use. To the largest extend possible, Sparkle aims to make sure that correct experimental procedures are followed (e.g., by correctly implementing the standard protocol for algorithm configuration from [35], also discussed later in Section V), and that potential pitfalls are avoided (see Section VIII). Since it is not always possible to guarantee correct use, Sparkle aims to provide adequate warnings and possible solutions when potential problems are detected that cannot be prevented by design (e.g., crashes of the algorithm that is being configured). As a result, at a minimum, the user should be made aware of potential problems (e.g., by writing warnings and errors to the command line) and should then be able to take action or seek help from experts.

Understandability is concerned with explaining the output and operation. The main vehicle for this is the reporting functionality of Sparkle. For each meta-algorithmic process (e.g., algorithm configuration), a report can be generated. This report then describes the experimental setup (e.g., which problem instances were used), the process that was followed (e.g., how performance is assessed), and the results, along with pertinent references to the literature. To show how Sparkle helps compared to using AAS and AAC in a stand-alone fashion, example reports for AAS and AAC are included in the supplementary material, together with raw output of AutoFolio and SMAC (the state-of-the-art AAS and AAC systems currently integrated into Sparkle). Concretely, compared to AutoFolio, the most significant additions of the Sparkle report for AAS are: a detailed description of the algorithm selection procedure and settings, performance comparisons between individual solvers and the algorithm selector, insight into individual contributions of component solvers to the performance of the algorithm selector, and plots for visual comparison. Similarly, compared to SMAC, the main additions of the Sparkle report for AAC are: a detailed description of the algorithm configuration procedure and settings, plots for visual comparison, and a comparison of the number of instances on which the target algorithm timed out with default and configured parameters. Beyond the reports, other ways to support understanding include analysis tools, for instance to assess parameter importance (see Section VII-B).

IV. Sparkle for Parameter-Less Solvers

With the design principles from the previous section in mind, we first consider Sparkle for the simplified case in which the performance parameters (which only affect the performance of a given solver) for all solvers are fixed at a predetermined setting or no such parameters exist. Under these conditions, AAC is not applicable, but performance complementarity between solvers can be exploited using AAS. The per-instance algorithm selection problem arises in situations

```
initialise.py
add_instances.py PTN/
add_solver.py --deterministic 0
    PbO-CCSAT-Generic/
add_solver.py --deterministic 0 CSCCSat/
add_solver.py --deterministic 0 MiniSAT/
add_feature_extractor.py
    SAT-features-competition2012_sparkle/
compute_features.py
run_solvers.py
construct_sparkle_portfolio_selector.py
generate_report.py
```

Listing 1. Example of per-instance algorithm selection in Sparkle for SAT solving.

where no single algorithm is the best for every problem instance of interest, and thus performance can be improved by selecting the best performing algorithm on a per-instance basis [10]. AAS aims to tackle the per-instance algorithm selection problem in an automated fashion [36], [37], [38]. The key idea is to construct a per-instance algorithm selector that predicts the most suitable algorithm for each given problem instance, based on reasonably efficiently computable features of that instance. Fig. 1 illustrates a typical process followed in the construction of an algorithm selector and indicates where the version of Sparkle described in the following improves the ease of use compared to using a selector construction tool without aid beyond the affordances commonly made by selector construction tools.

At the conceptual level, the Sparkle platform for parameterless solvers then comprises the following core components.

1) A collection S of solvers.
2) A collection I of problem instances used for training.
3) A collection E of feature extractors to compute problem instance features.
4) Feature data F computed using the feature extractors in E for the instances in I.
5) Performance data (for now, only running times are implemented, but the main principles are the same for solution quality optimization) P for the solvers in S on the instances in I.
6) A procedure C for constructing a portfolio-based selector R based on S, optimized for performance on I, using feature data from F and performance data from P.
7) A procedure N for computing the contributions of any solver $s \in S$ to a given portfolio-based selector R on the instances in I.

In addition, we need to provide the following support components.

1) A mechanism for performing runs of solvers from S, of feature extractors from E, of the portfolio-based selector R, of the construction procedure C and of the contribution analysis procedure N.
2) A user interface (UI) that makes it easy to add solvers to S, to remove solvers from S, to submit an instance i to be solved, and to access performance and contribution data for solvers in S.

Above, in Listing 1, we show a concrete example of how these components are realized and used in the form of concrete

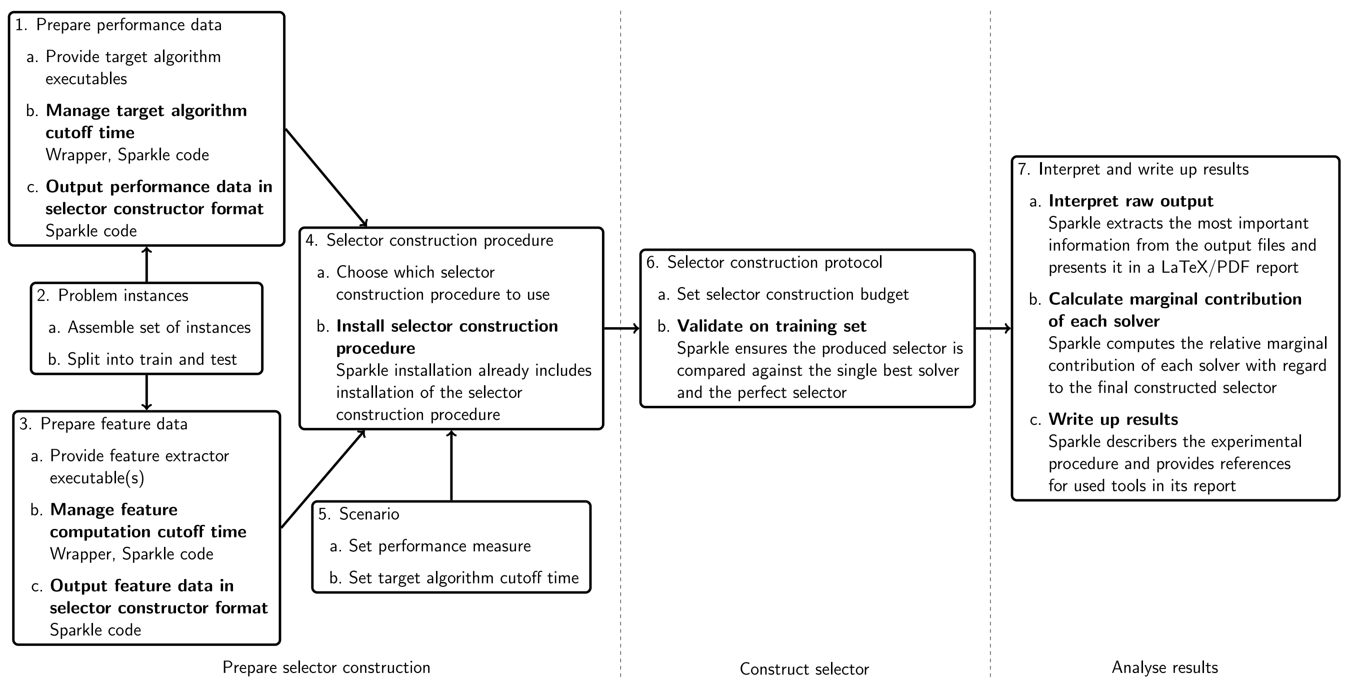
1. Prepare performance data
a. Provide target algorithm executables
b. Manage target algorithm cutoff time
Wrapper, Sparkle code
c. Output performance data in selector constructor format
Sparkle code
2. Problem instances
a. Assemble set of instances
b. Split into train and test
3. Prepare feature data
a. Provide feature extractor executable(s)
b. Manage feature computation cutoff time
Wrapper, Sparkle code
c. Output feature data in selector constructor format
Sparkle code
4. Selector construction procedure
a. Choose which selector construction procedure to use
b. Install selector construction procedure
Sparkle installation already includes installation of the selector construction procedure
5. Scenario
a. Set performance measure
b. Set target algorithm cutoff time
6. Selector construction protocol
a. Set selector construction budget
b. Validate on training set
Sparkle ensures the produced selector is compared against the single best solver and the perfect selector
7. Interpret and write up results
a. Interpret raw output
Sparkle extracts the most important information from the output files and presents it in a LaTeX/PDF report
b. Calculate marginal contribution of each solver
Sparkle computes the relative marginal contribution of each solver with regard to the final constructed selector
c. Write up results
Sparkle describes the experimental procedure and provides references for used tools in its report
Prepare selector construction
Construct selector
Analyse results

Fig. 1. Typical algorithm selector construction process, with steps where Sparkle improves the ease of use compared to using a stand-alone selector construction system shown in boldface. Solid lines connect related steps; dashed lines separate the phases: preparation, execution, and analysis.

Sparkle commands for constructing a per-instance algorithm selector based on three SAT solvers.[4]

As seen in this example, in practice, constructing a per-instance algorithm selector in Sparkle works as follows.

1) After a general initialization in line 1, a user seeds I with a collection of problem instances[5] suitable for training selectors in line 2 (e.g., when dealing with SAT solvers, these could be taken from past SAT competitions[6] [39], [40], [41], [42], [43], [44], [45], [46]), adds one or more solvers to S (lines 3–5) and adds one or more feature extractors to the collection E (line 6). The user then triggers the computation (optionally with `–parallel`) of the feature data F (line 7) and the performance data P (line 8), by using a Sparkle command to run S and E on all instances in I with a fixed cutoff time $t_{\max}$ (specified through a settings file). Finally, the user runs the construction procedure C to obtain an initial portfolio-based selector R (line 9). At any point following this, the user can generate a report for the current portfolio-based selector (line 10).
2) When a new instance i (or a set of problem instances) is to be solved, it can be passed to the `run_sparkle_portfolio_selector.py` command. Following this, its features are computed, using the feature extractors in E; the resulting feature vector is then passed to the portfolio-based selector R, which runs one or more solvers from S on the new instance i. When multiple solvers are chosen by the selector, these are executed according to a fixed schedule, based on the output of the selector.
3) When a new solver s is added by the user, s is added to S and run on all instances in I (this can be streamlined by using the `–run-solver-now` option with the `add_instances.py` command) with cutoff time $t_{\max}$; the resulting performance data are added to P. We note that s is run once for each instance, even for nondeterministic solvers. This approach has been taken, in order to avoid the overhead of performing multiple runs per instance, and to match common practices in current competitions. After this, C is run to obtain a new portfolio-based selector R that may utilize s.
4) Similarly, when a user removes a solver s, s is removed from S and the performance data for s is removed from P. Then, C is run to obtain a new portfolio-based selector R that no longer uses s.
5) A report generated for a portfolio-based selector,[7] contains detailed information (including the collection of instances I, solvers S, and feature extractors E, the procedures of constructing portfolio-based selectors, the detailed experimental setup, etc.). Further, the generated report contains the comparison of the constructed portfolio-based selector to both the single best solver and virtual best solver (VBS).[8]

[4]Some paths are shortened for legibility, e.g., `initialise.py` would be `Commands/initialise.py` and `PbO-CCSAT-Generic/` would be `path/to/PbO-CCSAT-Generic/`. A complete runnable example is available at https://bitbucket.org/sparkle-ai/sparkle/src/main/Examples/selection.sh.

[5]Sparkle aims to be general in the types of problems it supports. To accommodate situations where problem instances are specified by multiple files (as is usually the case in AI planning), an additional file can be included in the instance directories added to Sparkle to indicate which combinations of files jointly form a problem instance.

[6]http://www.satcompetition.org/

[7]An example report is included in the supplemental material.

[8]The VBS is the perfect selector and always reports the best solution out of those reported by all candidate solvers [47]. Hence, the VBS can be considered as an oracle, and its performance represents the upper bound for the performance of a portfolio-based selector.

In this initial implementation of Sparkle, per-instance algorithm selectors are constructed using the freely available, state-of-the-art AutoFolio system [19], which uses an algorithm configuration procedure to automatically construct high-performance algorithm selectors. Evidently, other per-instance algorithm selectors (e.g., [2], [5], [11], and [47]) and selector construction methods (e.g., [48] and [49]), as well as algorithm scheduling and parallel portfolio construction methods (e.g., [50]) can in principle be used instead of AutoFolio to exploit performance complementarity between parameter-less solvers. The advantages of AutoFolio are that: 1) it is a general-purpose method applicable to algorithms for arbitrary problems; 2) it incorporates a number of per-instance algorithm selection techniques known from the literature; and 3) it automatically configures an algorithm selector based on the resulting collection of algorithm selection procedures and their hyperparameters, to maximize performance in a given situation, as characterized by a given set of solvers, feature extractors, and training instances. The current implementation using AutoFolio clearly demonstrates the potential of the Sparkle system.

Performance can be assessed and optimized using various metrics, such as the number of solved instances [51], as well as PAR10, the penalized average running time with a penalty factor 10, which averages running time over a given set of instances, counting each timed-out run as ten times the given cutoff time [27]. Currently, for algorithm selection, Sparkle is restricted to the widely used PARk metric, where the penalty factor k can be set by the user through a settings file.

In the current implementation of Sparkle, solver contributions (to the PARk score of the per-instance algorithm selector) are assessed based on their marginal contribution (i.e., cost of omission) [52]. The marginal contribution defines the contribution of a given solver as the drop in performance caused by removing that solver from a given selector [52]. Note that the adoption of the marginal contribution can reward solvers that exhibit poor overall performance, but a strong performance on some nontrivial instance classes [51]. As a result, the use of the marginal contribution can identify a collection of solvers that have strong complementarity. However, the adoption of this metric can also penalize correlated candidate solvers. For example, if there are two solvers that show similar performance on the same instances, their marginal contribution would be small. This issue could be addressed by using another evaluation metric called the Shapley value [51], which is based on a prominent concept from cooperative game theory. However, compared to calculating the marginal contribution, computing Shapley values for all solvers would be significantly more time consuming. As a result, in order to efficiently compute each solver's contribution, we decided to use marginal contribution as our evaluation metric in this initial version of Sparkle.

To accurately compute the marginal contribution of a given solver s, we have to construct a per-instance algorithm selector that does not use s and assess its performance. Fortunately, the performance of a perfect selector, i.e., one that always selects for a given problem instance the solver from S that performs best on it, can easily be estimated in the form of the marginal contribution based on perfect selectors with and without s, and this can be calculated very efficiently from the performance data P. Furthermore, the performance of selectors produced by state-of-the-art selector construction methods is known to often be close to that of an idealized, perfect selector over the given set of solvers [52], [53]. To assess the performance of the actual selector R created by Sparkle, we also compute the marginal contribution of the solvers in S to this actual selector using the same mechanism. By comparing the performance of the perfect selector and the actual selector we can give insight on the optimality gap of the actual selector.

Using the marginal contribution to evaluate each given solver, there is a clear incentive for solver developers to focus their efforts on improving the state of the art, as represented by a high quality portfolio-based algorithm selector, such as AutoFolio [19], *Zilla [49], or ASAP [54]. The Sparkle platform operationalizes this incentive by constructing the per-instance algorithm selector, tracking solver contributions, and making detailed information on the performance and contribution of their solvers available to solver developers; it also provides a fair and well-defined way to assess solver contributions. We note that, although the marginal contribution is useful, there are other metrics one may want to consider. Selectors can also make suboptimal choices, and although AutoFolio (and thus Sparkle) specifically aims to mitigate this issue by suggesting algorithm schedules (to hopefully avoid the worst case where a single poorly chosen solver does not solve the problem), other and better solutions could be developed to address this general issue in AAS.

Considering our implementation in Sparkle, and the step-by-step process outlined in Fig. 1, we observe several improvements to the ease of use. In step 1, Sparkle takes care of most work the user would otherwise have to do to collect the performance data. Instead of writing their own scripts to run all combinations of instances and algorithms, ensuring correct and consistent cutoff time measurements between all algorithms (step 1b), and formatting the performance data (step 1c), with Sparkle, the user only has to adapt a small part of a wrapper template (for each algorithm) to call the algorithm executable and print the performance. Sparkle provides similar savings in time and effort for feature data preparation (step 3). In step 4b, Sparkle provides a minor convenience by automating the installation of AutoFolio, but this also saves some time and effort. Importantly, Sparkle ensures the process is performed correctly, by always running validation on the training set in step 6b. With AutoFolio, this is not guaranteed, since it runs validation only when the selector is not saved to a file. In step 7, Sparkle helps the user beyond the basic performance indicators returned by AutoFolio, by generating a report. This report includes (in addition to the basic performance indicators AutoFolio also returns for the produced selector) the individual performance per solver and a visual comparison of the selector to the single best solver (step 7a), the contribution to the selector of each solver in the form of the marginal contribution (step 7b), and the followed experimental procedure and references (step 7c). The automatic generation of the report not only saves time compared to using custom-built scripts or partially manual processes, but also helps to avoid errors.

```
 1  initialise.py
 2  add_instances.py PTN/
 3  add_solver.py --deterministic 0
        PbO-CCSAT-Generic/
 4  add_solver.py --deterministic 0 CSCCSat/
 5  add_solver.py --deterministic 0 MiniSAT/
 6  add_feature_extractor.py
        SAT-features-competition2012_sparkle/
 7  compute_features.py
 8  run_solvers.py
 9  construct_sparkle_portfolio_selector.py
10  generate_report.py
```

Listing 2. Example of algorithm configuration in Sparkle for the capacitated vehicle routing problem.

To confirm the benefits of Sparkle for AAS, a small study was performed with two users. They were asked how much time and code they needed for each step from Fig. 1. One user (using AutoFolio directly and through Sparkle) reduced the time spent by 68 % when using Sparkle compared to using AutoFolio without Sparkle, and reported a similar reduction (69 %) in terms of the lines of code they needed to write. For the second user (having only tried AutoFolio[9]) the results were strongly influenced by steps 1a and 2a which together took up more than 95 % of the time, and also a large portion of the code. This is the result of the user having to spend significant time and coding effort to get algorithms from external sources installed and running on a specific HPC environment (step 1a), and also spent significant time collecting problem instances (step 2a). Taking this into account, the time spent could have been reduced by 1 % when using Sparkle and the code written by 5 %. When we exclude these two steps (we note that these two steps are equal between Sparkle and AutoFolio without Sparkle), and look only at the remaining steps, however, the savings were quite a bit more significant, with the user being able to save 22 % of their time and 16 % of the coding effort. It is also worth mentioning that this second user did not compute the marginal contributions of the component algorithms to the produced algorithm selector, something Sparkle would have done for them. For both users, Sparkle helped to reduce the time and effort spent for making effective use of AAS. This suggest that Sparkle can indeed make the use of AAS easier. Full results and an explanation of the data processing are included in the supplemental material.

V. Sparkle for Parameterized Solvers

The version of Sparkle considered in the previous section exploited the fact that there are effective algorithm selection methods that can be used to leverage complementary strengths of nondominated solvers. In practice, many of these solvers are (or easily can be) parameterized, such that they can be configured for optimized performance on different types of instances. For example, in the case of SAT solving, the degree to which the performance of state-of-the-art solvers can be optimized for specific types of instances has been demonstrated in the CSSCs [3]. This motivates the use of automated configuration procedures, such as SMAC [20], irace [23], ParamILS [27], or GPS [28], to obtain state-of-the-art solvers for specific types of instances. Fig. 2 describes a typical process followed in algorithm configuration and indicates where the version of Sparkle described in the following improves the ease of use compared to using a configurator without aid beyond the affordances commonly made by algorithm configurators. To effectively exploit the configurability of solvers, we now describe an extension of the basic version of Sparkle discussed in Section IV.

In addition to the components introduced in the previous section, this extended version of Sparkle also comprises the following.

1) Configuration spaces for some (or all) of the solvers in S, where a configuration space Θ consists of a list of parameters, a domain of possible values for each such parameter, a (possibly empty) set of conditional parameter dependencies and a (possibly empty) set of constraints on combinations of parameter values.
2) An AAC procedure O for optimizing the performance of a given solver $s \in S$ on a subset of I.

In a practical setting, we also need to provide.

1) A mechanism for performing runs of the automatic configuration procedure O;
2) UI affordances to specify configuration spaces for solvers in S; to specify which subset of instances from I is to be used for a run of the configuration procedure O on a specific solver $s \in S$, to launch such a run and to validate the performance of the configurations of s thus obtained on another subset of instances from I.

The implementation in Sparkle is illustrated in Listing 2.[10] Following initialization (line 1), two instance sets from the same distribution are added to I to accommodate the training and testing phases (lines 2 and 3). Similarly, a solver s is added to S (line 4). Besides the solver executable,[11] only a wrapper and a pcs-file [55] (to specify the configuration space for a parameterized solver s) to be used for configuration have to be included in the solver directory to be added to Sparkle. To start the configuration procedure O, a solver and instance set that are available in Sparkle are specified (line 5). We note that here, the performance metric is changed from the default (running time), to absolute solution quality. Alternatively, this can also be specified in the settings file, such that the option does not have to be given with the commands. Likewise, other settings (e.g., cutoff times, the value k for the PARk score, etc.) can also be specified in the settings file, with common settings also changeable via command-line options. Configuration is then performed with multiple independent runs of the configuration procedure, following the widely used standard protocol for algorithm configuration (e.g., [35]). The use of this standard protocol is important, because it exploits the fact that algorithm configurators are typically randomized,

[9]Numbers for Sparkle were estimated based on their numbers for AutoFolio, the process for this is explained in the supplemental material.

[10]Some paths are shortened for legibility reasons, e.g., `initialise.py` would be `Commands/initialise.py` and `VRP_SISRs/` would be `path/to/VRP_SISRs/`. A complete runnable example is available at https://bitbucket.org/sparkle-ai/sparkle/src/main/Examples/configuration_quality.sh.

[11]The user is assumed to know how to install solvers in this Sparkle version.

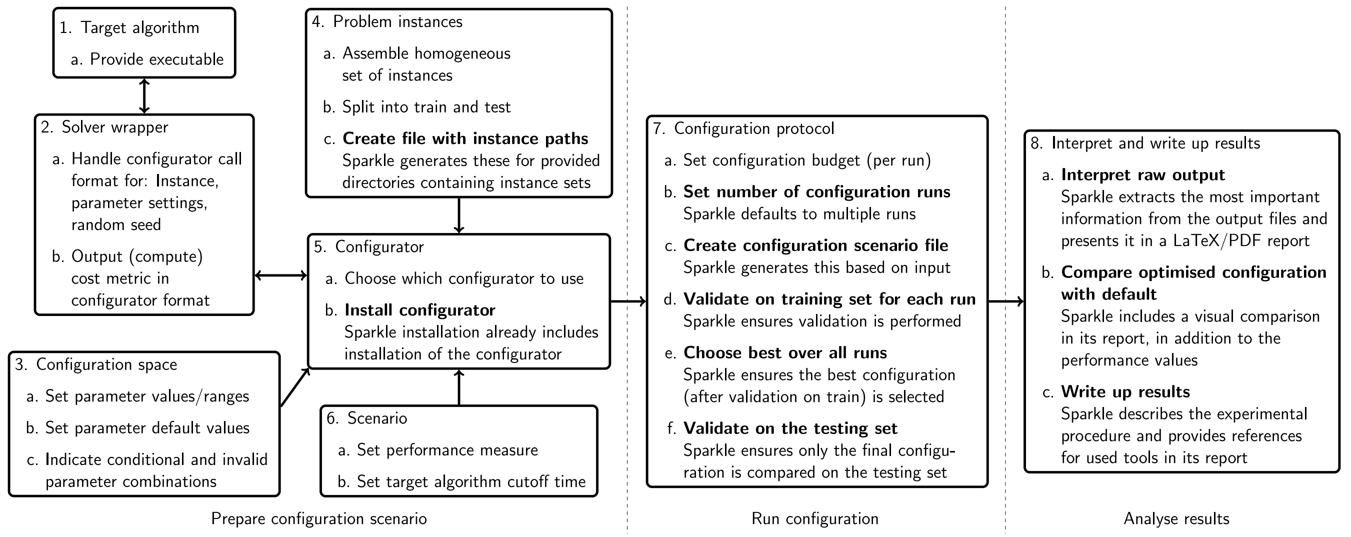
1. Target algorithm
a. Provide executable
2. Solver wrapper
a. Handle configurator call format for: Instance, parameter settings, random seed
b. Output (compute) cost metric in configurator format
3. Configuration space
a. Set parameter values/ranges
b. Set parameter default values
c. Indicate conditional and invalid parameter combinations
4. Problem instances
a. Assemble homogeneous set of instances
b. Split into train and test
c. **Create file with instance paths** Sparkle generates these for provided directories containing instance sets
5. Configurator
a. Choose which configurator to use
b. **Install configurator** Sparkle installation already includes installation of the configurator
6. Scenario
a. Set performance measure
b. Set target algorithm cutoff time
7. Configuration protocol
a. Set configuration budget (per run)
b. **Set number of configuration runs** Sparkle defaults to multiple runs
c. **Create configuration scenario file** Sparkle generates this based on input
d. **Validate on training set for each run** Sparkle ensures validation is performed
e. **Choose best over all runs** Sparkle ensures the best configuration (after validation on train) is selected
f. **Validate on the testing set** Sparkle ensures only the final configuration is compared on the testing set
8. Interpret and write up results
a. **Interpret raw output** Sparkle extracts the most important information from the output files and presents it in a LaTeX/PDF report
b. **Compare optimised configuration with default** Sparkle includes a visual comparison in its report, in addition to the performance values
c. **Write up results** Sparkle describes the experimental procedure and provides references for used tools in its report
Prepare configuration scenario
Run configuration
Analyse results

Fig. 2. Typical algorithm configuration process, with steps where Sparkle improves the ease of use compared to using a stand-alone configurator shown in boldface. Further, Sparkle also provides useful support for steps that are already commonly supported by AAC tools, e.g., by providing a broadly usable and carefully constructed wrapper template for target algorithms. Solid lines connect related steps; dashed lines separate the phases: preparation, execution, and analysis.

and that AAC is a time-consuming process that benefits from using parallel computing resources. Based on the same protocol, the configuration found by each run is then validated on the training set before selecting the best-performing one, which is then validated on the instances in the test set (line 6). Finally, a report[12] is generated which describes the experimental setup and compares the default and optimized configuration (line 7; as for AAS, it also details the components I, S, and C). To utilize this optimized configuration on a new instance i (or a set of problem instances), the `run_configured_solver.py` command can be used. It takes the most recently configured solver s and runs it on the given instance(s) with the optimized configuration found by the configuration procedure O.

Currently, SMAC [20], which is a widely used state of the art configurator that is also freely available for academic use, is the configurator O used in Sparkle. However, in principle this could also be any other state-of-the-art general-purpose algorithm configuration procedure, such as ParamILS [27], irace [23], GPS [28], or GGA++ [56].

The current performance metrics for running time (PARk) and absolute solution quality were chosen because they are most widely used. Beyond this, a user can define their own metric through a combination of the absolute solution quality metric and the wrapper for the target algorithm, although this requires some effort. The same applies to metrics that depend on the performance of other solvers, although using these requires knowledge about where Sparkle saves these results.

In this version of Sparkle, the configuration procedure O can be used to optimize solver performance for specific types of instances characterized by subsets of instances available in Sparkle (as part of instance set I). This makes it possible to configure solvers for overall performance across broad sets of instances, as well as, more importantly, for specific types of instances – the latter makes it even easier for solver developers to achieve contributions to the state-of-the-art in solving hard problems (e.g., SAT) by determining and then submitting configurations of their solvers that boost the performance of the portfolio-based selector constructed by Sparkle as described in Section IV.

When comparing our implementation in Sparkle to the step-by-step process for AAC shown in Fig. 2 (and specifically, SMAC), we observe several usability improvements. In step 4b, Sparkle automates the creation of files with paths to each individual training and testing instance. While users can do this fairly easily with their own scripts, it still saves some time. Installation of the AAC procedure is automated by Sparkle (step 5b). In step 7, Sparkle ensures a number of important aspects of AAC happen correctly (i.e., following the standard protocol for AAC [35]), and also saves time by automating these steps. Specifically, step 7b ensures multiple configuration runs are done, step 7c provides a minor aid by automating the creation of a configuration scenario file for SMAC, and steps 7d–7f automate the correct handling of validation for configuration results and selecting the final configuration. Finally, in step 8, Sparkle provides major support by providing much more detailed result analysis than SMAC does by default. Beyond the performance of the configured algorithm and the parameter string associated with it given by SMAC, Sparkle additionally provides the performance of the default configuration, plots comparing the performance of the configured and default algorithm parameters per instance, the number of timeouts of those two configurations, and a description of the followed experimental procedure. The automated collection of all results saves time, and gives the user substantially more insight. It also helps prevent mistakes.

For AAC, we also performed a small user study with four users, who were asked to estimate how much time and code they needed for each step in Fig. 2. One user (having tried

[12]An example report is included in the supplemental material.

AAC with SMAC[13]) would save 35 % of their time by using SMAC through Sparkle, while their code use would be almost equal (only 1 % less). Interestingly, the user did not follow many of the substeps of step 7 (see Fig. 2), which relate to the standard protocol for AAC [35], and thus little or no time was spent on these substeps. Had they followed the standard protocol, time and code savings with Sparkle likely would have been substantially larger. This also supports the need for tools like Sparkle, that, by design, ensure the correct and efficient use of AAC. The second user (also using AAC with SMAC[13]), could have reduced time spent by 51 % and lines of code written by 43 % by using Sparkle. For the third user (using SMAC through Sparkle) the data they entered shows that they spent almost no time and had to write no code at all for steps 7 and 8, suggesting that the major support Sparkle provides for these steps is helpful. Since we cannot estimate how much time and code they would have needed for those steps when using SMAC without Sparkle, it is not clear how much time and coding effort they saved by using Sparkle, but it is reasonable to assume the savings were substantial. The fourth user (using SMAC[13] without Sparkle) could have reduced the time they spent by 40 % with Sparkle, and the lines of code written by 29 %. The interested reader can find the instructions for users, complete results, and analysis procedure in the supplemental material.

VI. Sparkle as a Competition Platform

By utilizing the Sparkle platform, we can organize Sparkle challenges, a novel type of competitive event, which aims to advance the state of the art in solving various challenging computational problems, including Boolean satisfiability (SAT) [46], AI planning [57], and other problems, by leveraging automatically constructed algorithm selectors and by quantifying contributions of individual solvers.

As mentioned previously, it is well established that the state of the art for solving challenging computational problems (e.g., SAT [58], planning [54], minimum vertex cover [59], answer set programming [48], satisfiability modulo theories [60], etc.) is not defined by a single solver, but rather by a collection of nondominated solvers with complementary strengths. To exploit this performance complementarity, machine learning techniques can be leveraged to build effective automatic algorithm selectors that utilize state-of-the-art solvers. Sparkle challenges automatically combine all participating solvers into a state-of-the-art algorithm selector, and assess the contribution of each participating solver to the performance of that algorithm selector, using the functionality introduced in Section IV. Participants are incentivized to advance the state of the art as measured by this selector, by maximizing the contribution of their solver to the overall selector performance.

At the moment, traditional solver competitions (such as the international SAT competitions[6] [44], [45], [46], international planning competitions[14] [57], [61], [62], etc.) measure the performance of each individual solver across a large set of benchmark instances, and identify the winning solver(s) based on their overall performance across this instance set.

Rather than the gold, silver, and bronze medals awarded in traditional solver competitions, participants in Sparkle challenges are awarded slices of a single gold medal; the size of each slice is proportional to the magnitude of the marginal contribution made by the respective solver to the performance of the automatically constructed selector built from all participating solvers on all benchmarking instances. That is to say, Sparkle challenges identify the best solver per instance, and award solvers based on the number of instances for which they contribute the best performance.

In recent years, we have already organized two Sparkle challenges with earlier unpublished versions of the Sparkle platform described here. The Sparkle SAT Challenge 2018[1] was an official competition affiliated with the 21st International Conference on Theory and Applications of Satisfiability Testing,[15] while the Sparkle Planning Challenge 2019[2] was a satellite competitive event affiliated with the 29th International Conference on Automated Planning and Scheduling.[16]

Both events attracted significant participation and produced a series of interesting results, thus demonstrating the viability of using Sparkle for new competition formats. (A presentation and discussion of the results of the Sparkle challenges are beyond the scope of this article, but the interested reader can find further information on the web pages referenced earlier.)

VII. Other Uses and Extensions

In Sections IV and V, we have outlined the two primary use cases supported in the current version of Sparkle, algorithm selection, and algorithm configuration. Beyond those, a number of other use cases and extensions should be highlighted.

A. Benchmarking

Many tools in Sparkle can also be used for traditional benchmarking tasks, in addition to their use as part of the selection or configuration processes. In particular, the parallel processing functionalities available in Sparkle turn the parallel execution of a collection of algorithms or computation of the features for a set of instances into simple tasks, especially compared to writing scripts for each new case.

One specific benchmarking use-case enabled by Sparkle is the comparison of multiple solvers by using the AAS functionalities. This facilitates analysis for a specific experiment or application, as opposed to competitions, which often consider larger sets of solvers and problem instances. The report resulting from AAS provides, for each solver, the respective PAR score and marginal contribution to the portfolio selector. This gives an indication of the value of each solver. We emphasize again that analyzing solvers based on their marginal contributions better represents the state of the art than traditional ranking schemes aimed at pinpointing a single best solver. More details on which solver worked well on which specific

[13]Numbers for Sparkle were estimated based on their numbers for SMAC, the process for this is explained in the supplemental material.

[14]https://ipc2018.bitbucket.io/

[15]http://sat2018.azurewebsites.net/competitions/

[16]https://www.icaps-conference.org/competitions/

instances are stored in a file containing the results per problem instance, but currently have to be compared manually.

B. Parameter Importance Analysis

Following algorithm configuration, solver developers can gain valuable insights by analyzing the resulting configuration in more detail. Parameter importance analysis gives insights into how much individual parameters contribute to the performance difference between two parameter configurations of a solver s on a subset of the instances in I. This gives insight into the impact of parameter changes on solver performance. In Sparkle, we analyze the parameter importance between the default expert-chosen parameter configuration θ_d and the best-found configuration θ_c after automated configuration, both originating from the configuration space Θ for solver s.

Currently, Sparkle uses ablation analysis [63] to assess parameter importance. Ablation analysis constructs a path from the default configuration θ_d to the best-found configuration θ_c by iteratively changing the parameter that yields the largest performance gain, starting from θ_d. Eventually, all parameters are changed, and thus, the target configuration is reached. The resulting ablation path shows for each parameter how much the solver performance is affected and as such provides a local perspective of the parameter importance.

Ablation analysis in Sparkle requires a solver s, a subset of instances of I, and a finished configuration run on the former two. The command `run_ablation.py` verifies these prerequisites are met and then generates an ablation scenario. The default configuration θ_d is retrieved from the pcs-file of the solver, and the best parameter settings found by automatic configuration act as the target configuration θ_c. A second subset of instances of I enables the validation of the found ablation path. Ablation analysis can be started immediately after the algorithm configuration has finished when the optional flag `–ablation` is added to the `configure_solver.py` command. The results of the ablation analysis are automatically included in the configuration report as a table, where each row describes a parameter modification along with the resulting performance score. Optionally, the table can be excluded with the `–no-ablation` flag for `generate_report.py`.

C. Parallel Algorithm Portfolios

Similarly to AAS, parallel algorithm portfolios (PAPs) [50], [64], [65] can help to obtain greater performance out of a given (set of) algorithm(s). Where AAS leverages the performance variation between different algorithms, PAPs can also utilize the variation between multiple runs of a single, randomized algorithm. This is done by executing a portfolio of algorithms in parallel. When considering running time optimization, the performance variation between different algorithms will result in each finding solutions faster for different instances.

Here, we consider basic PAPs that do not take advantage of additional information. As such, it is not necessary to predict which algorithm is fastest, but all algorithms in the portfolio are always included, and in theory, they can always take advantage of the fastest running time, trading off wall-clock time against parallelism. In practice, the parallelization creates some overhead, and more sophisticated PAPs can run only the solvers predicted to be fast, in order to reduce overhead.

This first version of PAPs has been implemented in Sparkle, and broadly follows the same command flow as AAS and AAC. A selection of algorithms can be combined into a PAP, and for each nondeterministic algorithm, the desired number of copies can be indicated. Once a portfolio has been created, it can be used on a subset of instances in I, of which the performance results are presented in a report. These PAPs are especially useful to effectively utilize parallel resources, increase robustness, and to still gain performance when no good features are yet available to use AAS on a given problem.

VIII. Best Practices and Pitfall Avoidance

It is well known that pitfalls commonly arise when using AAC techniques, and that certain practices are helpful in avoiding these and ensuring successful applications [17]; similar considerations apply to the effective use of AAS and other meta-algorithmic techniques (e.g., in the context of CP [66]). Sparkle aims to incorporate best practices to make correct and effective use of these techniques.

We do not exhaustively discuss all pitfalls here, since some of them do not apply (e.g., each configurator handling the algorithms under configuration differently) and others are beyond the scope of the current implementation of Sparkle (e.g., assuring generalisability across machines with different hardware). Naturally, as Sparkle is further extended, additional best practices will be incorporated.

Correct interaction between the solvers included by the user, the configurator, and Sparkle itself, is supported by providing wrapper templates to the user for both AAS and AAC.

As mentioned previously in the discussion of Sparkle for parameterized solvers (Section V), the widely used standard protocol for algorithm configuration [35] is integrated into Sparkle, to facilitate its correct and efficient use. Particularly, Sparkle performs multiple runs of the configurator, validates the best configuration from each run on the training instances, and selects the best performing from those.

To ensure the correct handling of the running time cutoffs for target algorithms, Sparkle uses runsolver, a tool that is widely used in benchmarking studies and competitions [67], [68]. The main goal in this context is to measure running times in a trusted and consistent way, rather than relying on target algorithms that may each measure and report their running time differently and possibly incorrectly. For some cases, runsolver is called in the wrapper the user provides for their solver, and while templates for these wrappers are provided that include this call, it still, in part, relies on the user. Like the running time cutoffs, the actual measurements of the running time of target algorithms are also done through runsolver. The same goes for other situations where time is measured, such as feature extraction.

With regard to the termination of target algorithm runs, Sparkle also relies on runsolver: To ensure termination happens at the right time, runsolver measures the time, and terminates when the cutoff time is reached. As with time measurement, the termination procedure also partially relies on a wrapper that is adapted by the user for their target algorithm, but otherwise Sparkle handles everything.

Correct use is also supported by some smaller adjustments. When configuring nondeterministic algorithms, Sparkle uses multiple random seeds per problem instance, to avoid overtuning to specific seeds. To deal with unexpected output, some checks are included, e.g., to try to detect target algorithm crashes. To deal with incorrect results returned by target algorithms, solution checkers should be used whenever possible, and such a checker is included for SAT. Memory limits for algorithm runs are handled through the Slurm workload manager,[17] by including them in run and batch calls to Slurm.

IX. Conclusion

To conclude, we summarize the main takeaways of our work, before briefly discussing directions for future work.

A. Summary

High-performance solvers have been key to improving our ability to solve computationally challenging problems, such as SAT, MIP, and AI planning, including in academic and real-world settings. Improvements to these solvers are in part incentivized by benchmarks and competitions, where participants drive each other to continuously advance the state of the art. At the same time, by utilizing meta-algorithmic techniques, such as AAC and AAS, performance can be improved by making maximal use of the potential available from existing algorithms and algorithm components.

Assessing the performance of such improvements is still commonly done by measuring which algorithm is the best overall. However, this is not an accurate representation of the state of the art. The true state of the art is represented by a collection of complementary solvers that perform well on different subsets of problem instances (as also leveraged by AAS). This is complemented by techniques, such as AAC and PAPs, which get us closer to the maximal performance potential of algorithms or portfolios of algorithms, and as such also support a more accurate picture of the state of the art. The assessment of the true state of the art requires complex specialized techniques that are not easy to use correctly, and the same applies to using meta-algorithmics to benefit from their ability to maximize performance.

To this end, we introduced the Sparkle platform. With this platform, meta-algorithmic tools are made accessible to solver developers and users that may not have much expertise in meta-algorithmics. The key principles behind making these tools accessible are that they can be used effectively and correctly, even by nonexperts. To achieve this, standard protocols are implemented to automatically ensure correct use and pitfall avoidance where possible. Otherwise, checking mechanisms are used to alert the user to potential problems, and support messages are provided to guide them in resolving potential issues. All of this drives the increased adoption and application of meta-algorithmic techniques, which in turn improves performance and results in a more accurate view of the state of the art.

Specifically, in this work we have shown how AAS and AAC are implemented in and accessible through Sparkle. By implementing standard protocols, the effort and expertise required from users to adopt meta-algorithms are decreased compared to stand-alone AAS and AAC tools. In addition, we have outlined the use of Sparkle as a competition platform. By employing the meta-algorithms available in Sparkle for competitions, the outcome of the competitions more accurately represents the true state of the art. With AAS solvers can be credited based on their contribution, which recognizes that different solvers are the best for different instances, and there is no one best solver. Finally, extensions to parameter importance analysis and the use of PAPs have been discussed to showcase how Sparkle can further support the user in analyzing results and gain performance from other meta-algorithmic techniques.

B. Future Work

To reach the overarching goal of the Sparkle platform to cover the full range of meta-algorithmic techniques that are of broad interest, several extensions are desirable.

The Sparkle team aims to ensure support for an up-to-date collection of AAS and AAC techniques, and mechanisms for others to add such techniques. This will keep Sparkle aligned with the advancements in AAS and AAC, and also facilitate extensions to benchmarking and comparison of these techniques. By including, new, improved, techniques in Sparkle, users can easily adopt them, since the interface stays the same.

To further maximize the impact of solvers, integrations of selection and configuration procedures will be provided in Sparkle. One way is to configure a solver that optimizes its contribution to a portfolio-based selector, similar to Hydra [13], [69]. Sparkle could be extended to obtain new configurations for a set of separate and possibly very different solvers, rather than for a single parameterized solver, as in Hydra. With this extension in place, interesting combinations of AAC and AAS could be used in Sparkle challenges.

While multiobjective meta-algorithms are still scarce, some do exist, such as the multiobjective algorithm configurator, MO-ParamILS [70]. Sparkle should be extended to support this for a broader audience, to also facilitate the many problem domains with conflicting objectives.

For AAS as well as AAC, several useful extensions to Sparkle are possible. For AAS, this includes: 1) metrics other than PAR*k*, starting with absolute solution quality and 2) alternatives to the marginal contribution to assess algorithm contributions to a selector. Specific to AAC, ideas include: 1) additional performance metrics beyond PAR*k* and absolute solution quality, including mechanisms for users to add their own metrics and 2) global parameter importance analysis tools (e.g., forward selection [71] or functional ANOVA [72]).

Once the functionality currently provided by Sparkle is in broad use, the state of the art in solving a diverse range of computational problems should, and can easily, be assessed

[17]https://slurm.schedmd.com

more accurately, by means of solver configurability and complementarity. Improvements to the state of the art can then also be better incentivised by new types of competitions, such as the CSSCs [3] and Sparkle challenges.[1],[2] Beyond providing the necessary tools, future versions of Sparkle should therefore simplify setting up such competitions, e.g., by reducing the manual work for organizers.

In parallel to the possible extensions mentioned above, Sparkle should also be further improved in how it supports users. For example, a testing harness could be supplied that audits the compliance of a solver with the requirements of Sparkle's configuration mechanism, and installation of both Sparkle itself and target algorithms could be simplified with support for container technologies (e.g., Docker[18] or Kubernetes[19]). On the practical side, the current version of Sparkle only runs with the widely used Slurm workload manager.[17] However, everything could in principle run on a single machine, a cluster, or in the cloud, and we aim to accommodate all of these scenarios in the future. Finally, by further reducing the prerequisite knowledge required to use Sparkle, meta-algorithmic techniques should become even more accessible to a progressively wider audience.

Once completed, these extensions bring Sparkle close to the vision of making the full range of meta-algortihmic techniques that are of broad interest accessible and usable for a wide audience. The resulting increase in adoption of these techniques should help researchers and practitioners to realize the full potential of their algorithms, and to benefit from each others' work, thus maximally advancing the true state of the art in solving challenging computational problems.

Acknowledgment

The authors would like to thank Richard Middelkoop for work on the implementation of PAPs, and all users we have interviewed for taking the time to provide input on the amount of code and time they needed per step for AAS or AAC. Some of the ideas discussed in this document have their roots in joint work and discussions with Frank Hutter, Chris Fawcett, and Kevin Leyton-Brown.

References

[1] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *Int. J. Softw. Tools Technol. Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[2] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, "ISAC—Instance-specific algorithm configuration," in *Proc. 19th Eur. Conf. Artif. Intell. (ECAI)*, 2010, pp. 751–756.

[3] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. H. Hoos, and K. Leyton-Brown, "The configurable SAT solver challenge (CSSC)," *Artif. Intell.*, vol. 243, pp. 1–25, Feb. 2017.

[4] C. Ansótegui, J. Gabàs, Y. Malitsky, and M. Sellmann, "MaxSAT by improved instance-specific algorithm configuration," *Artif. Intell.*, vol. 235, pp. 26–39, Jun. 2016.

[5] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm portfolios based on cost-sensitive hierarchical clustering," in *Proc. 23rd Int. Joint Conf. Artif. Intell. (IJCAI)*, 2013, pp. 608–614.

[6] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Automated configuration of mixed integer programming solvers," in *Proc. 7th Int. Conf. Integr. AI OR Techn. Constraint Program. Comb. Optim. Problems (CPAIOR)*, 2010, pp. 186–202.

[7] C. Fawcett, M. Helmert, H. Hoos, E. Karpas, G. Röger, and J. Seipp, "FD-Autotune: Domain-specific configuration using fast downward," in *Proc. ICAPS Workshop Planning Learn. (PAL)*, 2011, pp. 13–20.

[8] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA," *J. Mach. Learn. Res.*, vol. 18, no. 25, pp. 1–5, 2017.

[9] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. (KDD)*, Chicago, IL, USA, Aug. 2013, pp. 847–855.

[10] J. R. Rice, "The algorithm selection problem," in *Advances in Computers*, vol. 15. Amsterdam, The Netherlands: Elsevier, 1976, pp. 65–118.

[11] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res.*, vol. 32, pp. 565–606, Jun. 2008.

[12] C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney, "Reactive dialectic search portfolios for MaxSAT," in *Proc. 31st AAAI Conf. Artif. Intell. (AAAI)*, 2017, pp. 765–772.

[13] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming," in *Proc. 18th RCRA Workshop Exp. Eval. Algorithms Solving Problems Comb. Explosion (RCRA)*, 2011, pp. 16–30.

[14] G. Di Liberto, S. Kadioglu, K. Leo, and Y. Malitsky, "DASH: Dynamic approach for switching heuristics," *Eur. J. Oper. Res.*, vol. 248, no. 3, pp. 943–953, 2016.

[15] D. Bridge, E. O'Mahony, and B. O'Sullivan, *Case-Based Reasoning for Autonomous Constraint Solving*. Berlin, Germany: Springer, 2012, pp. 73–95.

[16] M. Helmert, G. Röger, and E. Karpas, "Fast downward stone soup: A baseline for building planner portfolios," in *Proc. ICAPS Workshop Plan. Learn.*, 2011, pp. 28–35.

[17] K. Eggensperger, M. Lindauer, and F. Hutter, "Pitfalls and best practices in algorithm configuration," *J. Artif. Intell. Res.*, vol. 64, pp. 861–893, Jan. 2019.

[18] X. Bouthillier and G. Varoquaux, "Survey of machine-learning experimental methods at NeurIPS2019 and ICLR2020," INRIA Saclay, Ile-de-France, Palaiseau, France, Rep. hal-02447823, 2020.

[19] M. T. Lindauer, H. H. Hoos, F. Hutter, and T. Schaub, "AutoFolio: An automatically configured algorithm selector," *J. Artif. Intell. Res.*, vol. 53, pp. 745–778, Aug. 2015.

[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proc. 5th Int. Conf. Learn. Intell. Optim. (LION)*, 2011, pp. 507–523.

[21] N. Hansen, A. Auger, R. Ros, O. Mersmann, T. Tušar, and D. Brockhoff, "COCO: A platform for comparing continuous optimizers in a black-box setting," *Optim. Methods Softw.*, vol. 36, no. 1, pp. 114–144, 2021.

[22] J. Dreo et al., "Paradiseo: From a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of Paradiseo," in *Proc. GECCO Companion*, 2021, pp. 1522–1530.

[23] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Oper. Res. Perspect.*, vol. 3, pp. 43–58, Jan. 2016.

[24] C. Doerr, F. Ye, N. Horesh, H. Wang, O. M. Shir, and T. Bäck, "Benchmarking discrete optimization heuristics with IOHprofiler," *Appl. Soft Comput. J.*, vol. 88, Mar. 2020, Art. no. 106027.

[25] A. Aziz-Alaoui, C. Doerr, and J. Dreo, "Towards large scale automated algorithm design by integrating modular benchmarking frameworks," in *Proc. Genet. Evol. Comput. Conf. Companion*, 2021, pp. 1365–1374.

[26] T. Bartz-Beielstein, C. W. Lasarczyk, and M. Preuss, "Sequential parameter optimization," in *Proc. IEEE Congr. Evol. Comput.*, vol. 1, 2005, pp. 773–780.

[27] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An automatic algorithm configuration framework," *J. Artif. Intell. Res.*, vol. 36, pp. 267–306, Sep. 2009.

[28] Y. Pushak and H. H. Hoos, "Golden parameter search: Exploiting structure to quickly configure parameters in parallel," in *Proc. Genet. Evol. Comput. Conf.*, New York, NY, USA, 2020, pp. 245–253.

[29] F. Hutter et al., "AClib: A benchmark library for algorithm configuration," in *Proc. 8th Int. Conf. Learn. Intell. Optim.*, Gainesville, FL, USA, Feb. 2014, pp. 36–40.

[18]https://www.docker.com/

[19]https://kubernetes.io/

[30] B. Bischl et al., "ASlib: A benchmark library for algorithm selection," *Artif. Intell.*, vol. 237, pp. 41–58, Aug. 2016.

[31] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Auto-sklearn: Efficient and robust automated machine learning," in *Automated Machine Learning*. Cham, Switzerland: Springer, 2019, pp. 113–134.

[32] N. Erickson et al., "AutoGluon-tabular: Robust and accurate AutoML for structured data," 2020, *arXiv:2003.06505*.

[33] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "OpenML: Networked science in machine learning," *SIGKDD Explorations*, vol. 15, no. 2, pp. 49–60, Jun. 2014.

[34] C. Nell, C. Fawcett, H. H. Hoos, and K. Leyton-Brown, "HAL: A framework for the automated design and analysis of high-performance algorithms," in *Proc. 5th Int. Conf. Learn. Intell. Optim. (LION 5)*, 2011, pp. 600–615.

[35] J. Styles, H. H. Hoos, and M. Müller, "Automatically configuring algorithms for scaling performance," in *Proc. 6th Int. Conf. Learn. Intell. Optim. (LION 6)*, Paris, France, Jan. 2012, pp. 205–219.

[36] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann, "Automated algorithm selection: Survey and perspectives," *Evol. Comput.*, vol. 27, no. 1, pp. 3–45, 2019.

[37] L. Kotthoff, "Algorithm selection for combinatorial search problems: A survey," *AI Mag.*, vol. 35, no. 3, pp. 48–60, 2014.

[38] M. A. Muñoz, M. Kirley, and S. K. Halgamuge, "Exploratory landscape analysis of continuous space optimization problems using information content," *IEEE Trans. Evol. Comput.*, vol. 19, no. 1, pp. 74–87, Feb. 2015.

[39] D. Le Berre and L. Simon, "The essentials of the SAT 2003 competition," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Germany: Springer, 2004, pp. 452–467.

[40] D. Le Berre and L. Simon, "Fifty-five solvers in Vancouver: The SAT 2004 competition," in *Theory and Applications of Satisfiability Testing*, H. H. Hoos and D. G. Mitchell, Eds. Berlin, Germany: Springer, 2005, pp. 321–344.

[41] O. Kullmann, "The SAT 2005 solver competition on random instances," *J. Satisfiability Boolean Model. Comput.*, vol. 2, nos. 1–4, pp. 61–102, 2006.

[42] D. Le Berre and L. Simon, "Preface to the special volume on the SAT 2005 competitions and evaluations," *J. Satisfiability Boolean Model. Comput.*, vol. 2, nos. 1–4, pp. 1–14, 2006.

[43] A. Balint, A. Belov, M. Järvisalo, and C. Sinz, "Overview and analysis of the SAT challenge 2012 solver competition," *Artif. Intell.*, vol. 223, pp. 120–155, Jun. 2015.

[44] T. Balyo, A. Biere, M. Iser, and C. Sinz, "SAT race 2015," *Artif. Intell.*, vol. 241, pp. 45–65, Dec. 2016.

[45] T. Balyo, M. J. H. Heule, and M. Järvisalo, "SAT competition 2016: Recent developments," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 5061–5063.

[46] M. J. H. Heule, M. Järvisalo, and M. Suda, "SAT competition 2018," *J. Satisfiability Boolean Model. Comput.*, vol. 11, no. 1, pp. 133–154, 2019.

[47] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm selection and scheduling," in *Proc. 17th Int. Conf. Princ. Pract. Constraint Program. (CP)*, 2011, pp. 454–469.

[48] H. Hoos, M. T. Lindauer, and T. Schaub, "Claspfolio 2: Advances in algorithm selection for answer set programming," *Theory Pract. Logic Program.*, vol. 14, nos. 4–5, pp. 569–585, 2014.

[49] C. Cameron, H. H. Hoos, K. Leyton-Brown, and F. Hutter, "OASC-2017: *Zilla submission," in *Proc. Open Algorithm Selection Challenge*, vol. 79. Brussels, Belgium, Nov./Dec. 2017, pp. 15–18.

[50] M. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Schaub, "Automatic construction of parallel portfolios via algorithm configuration," *Artif. Intell.*, vol. 244, pp. 272–290, Mar. 2017.

[51] A. Fréchette, L. Kotthoff, T. P. Michalak, T. Rahwan, H. H. Hoos, and K. Leyton-Brown, "Using the Shapley value to analyze algorithm portfolios," in *Proc. 30th AAAI Conf. Artif. Intell. (AAAI)*, 2016, pp. 3397–3403.

[52] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, "Evaluating component solver contributions to portfolio-based algorithm selectors," in *Proc. 15th Int. Conf. Theory Appl. Satisfiability Test. (SAT)*, Trento, Italy, Jun. 2012, pp. 228–241.

[53] L. Kotthoff, "On algorithm selection, with an application to combinatorial search problems," Ph.D. dissertation, Univ. St Andrews, St Andrews, U.K., 2012.

[54] M. Vallati, L. Chrpa, and D. E. Kitchin, "ASAP: An automatic algorithm selection approach for planning," *Int. J. Artif. Intell. Tools*, vol. 23, no. 6, 2014, Art. no. 1460032.

[55] F. Hutter and S. Ramage. "Manual for SMAC version v2.10.03-master." 2015. [Online]. Available: http://www.cs.ubc.ca/labs/beta/Projects/SMAC/v2.10.03/manual.pdf

[56] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, and K. Tierney, "Model-based genetic algorithms for algorithm configuration," in *Proc. 24th Int. Joint Conf. Artif. Intell. (IJCAI)*, 2015, pp. 733–739.

[57] A. F. Bocchese, C. Fawcett, M. Vallati, A. E. Gerevini, and H. H. Hoos, "Performance robustness of AI planners in the 2014 international planning competition," *AI Commun.*, vol. 31, no. 6, pp. 445–463, 2018.

[58] C. Luo, H. Hoos, and S. Cai, "PbO-CCSAT: Boosting local search for satisfiability using programming by optimisation," in *Parallel Problem Solving From Nature (PPSN XVI)*. Cham, Switzerland: Springer, 2020, pp. 373–389.

[59] C. Luo, H. H. Hoos, S. Cai, Q. Lin, H. Zhang, and D. Zhang, "Local search with efficient automatic configuration for minimum vertex cover," in *Proc. 28th Int. Joint Conf. Artif. Intell. (IJCAI)*, Macao, China, Aug. 2019, pp. 1297–1304.

[60] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, "MachSMT: A machine learning-based algorithm selector for SMT solvers," in *Tools Algorithms Construction Analysis of Systems*. Cham, Switzerland: Springer, 2021, pp. 303–325.

[61] M. Vallati, L. Chrpa, and T. L. McCluskey, "What you always wanted to know about the deterministic part of the international planning competition (IPC) 2014 (but were too afraid to ask)," *Knowl. Eng. Rev.*, vol. 33, p. e3, Apr. 2018.

[62] D. Pellier and H. Fiorino, "From classical to hierarchical: Benchmarks for the HTN track of the international planning competition," 2021, *arXiv:2103.05481*.

[63] C. Fawcett and H. H. Hoos, "Analysing differences between algorithm configurations through ablation," *J. Heuristics*, vol. 22, no. 4, pp. 431–458, 2016.

[64] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artif. Intell.*, vol. 126, nos. 1–2, pp. 43–62, 2001.

[65] B. A. Huberman, R. M. Lukose, and T. Hogg, "An economics approach to hard computational problems," *Science*, vol. 275, no. 5296, pp. 51–54, 1997.

[66] R. Amadini, M. Gabbrielli, and J. Mauro, "Why CP portfolio solvers are (under)Utilized? issues and challenges," in *Proc. LOPSTR*, 2015, pp. 349–364.

[67] V. M. Manquinho and O. Roussel, "The first evaluation of pseudo-Boolean solvers (PB)," *J. Satisfiability Boolean Model. Comput.*, vol. 2, nos. 1–4, pp. 103–143, 2006.

[68] O. Roussel, "Controlling a solver execution with the runsolver tool," *J. Satisfiability Boolean Model. Comput.*, vol. 7, no. 4, pp. 139–144, 2011.

[69] L. Xu, H. H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," in *Proc. 24th AAAI Conf. Artif. Intell. (AAAI)*, 2010, pp. 210–216.

[70] A. Blot, H. H. Hoos, L. Jourdan, M. Kessaci-Marmion, and H. Trautmann, "MO-ParamILS: A multi-objective automatic algorithm configuration framework," in *Proc. 10th Int. Conf. Learn. Intell. Optim. (LION 10)*, 2016, pp. 32–47.

[71] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Identifying key algorithm parameters and instance features using forward selection," in *Proc. 7th Int. Conf. Learn. Intell. Optim.*, Catania, Italy, Jan. 2013, pp. 364–381.

[72] F. Hutter, H. Hoos, and K. Leyton-Brown, "An efficient approach for assessing hyperparameter importance," in *Proc. 31st Int. Conf. Mach. Learn. (ICML)*, vol. 32. Beijing, China, Jun. 2014, pp. 754–762.

Koen van der Blom received the Ph.D. degree in multiobjective evolutionary optimization for early-stage building design from Leiden University, Leiden, The Netherlands, in 2019.

He is a Postdoctoral Researcher with LIP6, Sorbonne Université, Paris, France. His research interests include the accessibility of meta-algorithms, such as automated algorithm selection and configuration, evolutionary computation for real-world applications, and multiobjective optimization.

Dr. van der Blom's thesis received an honorable mention for the 2020 ACM SIGEVO Best Dissertation Award.

Holger H. Hoos received the Ph.D. (Dr.rer.nat.) degree in computer science from the Technical University Darmstadt, Darmstadt, Germany, in 1998.

He holds an Alexander von Humboldt Professorship of AI with RWTH Aachen University, Aachen, Germany, as well as a Professorship of Machine Learning with Universiteit Leiden, Leiden, The Netherlands, and an Adjunct Professorship of Computer Science with The University of British Columbia, Vancouver, BC, Canada.

Dr. Hoos is the past President of the Canadian Association for Artificial Intelligence and one of the initiators of CLAIRE, an initiative by the European AI community that seeks to strengthen European excellence in AI research and innovation. He is known for his work on machine learning and optimization methods for the automated design of high-performance algorithms and on stochastic local search, he has developed—and vigorously pursues—the paradigm of programming by optimization; he is also one of the originators of the concept of automated machine learning. He has a penchant for work at the boundaries between computing science and other disciplines, and much of his work is inspired by real-world applications. He is a Fellow of the Association of Computing Machinery, the Association for the Advancement of Artificial Intelligence, and the European AI Association (EurAI).

Chuan Luo received the Ph.D. degree in computer science from Peking University, Beijing, China, in 2016.

He is currently an Associate Professor with the School of Software, Beihang University, Beijing. His current research interests include heuristic search, combinatorial optimization, software testing, and cloud computing.

Jeroen G. Rook is currently pursuing the Ph.D. degree in multiobjective meta-algorithmics with the Data Management and Biometrics Group, University of Twente, Enschede, The Netherlands.

His research focuses on creating multiobjective methods for automated algorithm configuration and selection to make them more statistically robust, versatile, and powerful. More broadly, he is interested in multiobjective optimization and algorithm benchmarking.