Sustainable Benchmarking Tool

Ashlin Iser[1]
Karlsruhe Institute of Technology, Germany

Marie Anastacio[1]
AIM, RWTH Aachen University, Germany

Théo Matricon[1]
Univ. Rennes, Inria, CNRS, IRISA, France

Laurent Simon
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, Talence, France

Holger H. Hoos
AIM, RWTH Aachen University, Germany
LIACS, Leiden University, The Netherlands
University of British Columbia, Vancouver, Canada

Abstract

Solvers for NP-hard problems from areas such as automated reasoning or optimisation are complex systems in which many different components interact. The performance of these solvers is the result of an intricate interplay between implementation details, algorithmic concepts and heuristics. This, alongside the complexity of the problem instances to be solved, makes it challenging to assess the effect of a single idea on the overall performance of a given solver. It is therefore not only crucial, but also challenging to evaluate the performance impact of new ideas. Existing reliable evaluation methods require large sets of diverse benchmark instances and considerable amounts of computing resources. This makes empirical evaluation a bottleneck for solver development, as it is time-consuming and energy-intensive, often requiring several CPU years of computation to evaluate the impact of a single idea. In recent years, this bottleneck has led to the development of data-driven approaches that can dynamically select a smaller number of instances that provide sufficient statistical evidence to evaluate the relative performance of a given set of solvers. However, these methods are typically not easily accessible. In this work, we present a tool that implements these methods and makes them readily accessible to solver developers, thus enabling them to obtain swifter feedback on their ideas.

2012 ACM Subject Classification Software and its engineering → Software performance; Theory of computation → Design and analysis of algorithms

Keywords and phrases Sustainability, Empirical performance comparison, Benchmarking, Problem instance selection

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.36

Category Tool Paper

Supplementary Material *Software*: `https://github.com/ADA-research/DikeBenchmarker` archived at `swh:1:dir:2b23e37b127142ec35af40c22d7bcc39e501cc34`

Funding This work has been made possible by the Randomised Optimisation Algorithms Research Network (ROAR-NET), CA22137, COST Action (European Cooperation in Science and Technology). *Holger H. Hoos*: Alexander von Humbold professorship.

[1] These authors contributed equally to this work.

1 Introduction

Evaluating a new version of a solver, whether it incorporates a novel heuristic, a refined preprocessing step or an improved data structure, typically requires running it on several hundreds of benchmark instances. This process is both time- and energy-consuming, often requiring several CPU years of computation for a single thorough comparison. For solver developers iterating on new ideas, this constitutes a major bottleneck: the feedback loop between implementing a modification and understanding its impact on performance is prohibitively slow and expensive. This problem also arises when developing massively parallel solvers, for which solving a single problem instance may require hundreds of CPU.

In recent years, statistical methods have been developed to address this challenge by carefully selecting a small subset of benchmark instances that suffice for drawing reliable conclusions about relative algorithm performance [2, 9, 16, 17]. These methods can substantially reduce the computational cost of solver evaluation by exposing a trade-off between cost and statistical confidence. However, despite their demonstrated effectiveness and their application to related domains (see *e.g.* [15]), these techniques have seen limited adoption in practice, in part, because they are not readily available in an easy-to-use form.

Here, we present a tool that bridges this gap by implementing state-of-the-art instance selection methods and making them directly and easily accessible to solver developers. Given a set of benchmark instances with known performance data for existing solvers, our tool automatically selects an informative subset of instances on which to evaluate a new solver variant, and provides a statistically sound assessment of its relative performance. By lowering the technical barrier to rigorous empirical evaluation and making the cost–confidence trade-off explicit, our tool aims to accelerate the development cycle of solvers for NP-hard problems such as SAT, and more broadly, to support more sustainable benchmarking practices.

`Dike Benchmarker` serves as a platform that enables researchers to easily test and compare new benchmarking methods. Thanks to its modular, extensible architecture, new instance selection techniques, stopping criteria and data handling methods can be seamlessly integrated. Once embedded into our tool, these can be made available directly to practitioners for use in their evaluations. More specifically, our contributions are the following:

- an open-source tool providing implementations of state-of-the-art techniques to run efficient and reliable performance comparisons;
- a modular framework to easily experiment with new benchmarking methods for researchers and to make them accessible to practitioners;
- a demonstration of use cases of such techniques.

In the remainder of this work, we first introduce the problem at hand and present relevant related work (Section 2), next, we describe the architecture of our tool (Section 3) and the instance selection methods and stopping criteria included in it (Appendix A). We then show results on a simple use case (Section 4) and finally provide some general conclusions and an outlook on future work (Section 5).

2 Background

The empirical evaluation of the performance of algorithms plays a key role in assessing advancements of the state of the art in solving a given problem. Many research communities within AI and beyond have adopted a culture of regularly assessing the performance of algorithms on a large set of benchmark instances – see *e.g.* the competitions in Boolean Satisfiability (SAT) [11], Satisfiability Modulo Theories (SMT) [22] and automated planning [19].

This practice has lead to significant progress in the respective research areas, but also typically incurs high computational cost. To illustrate the cost of benchmarking SAT solvers: the 2024 SAT Competition [11] required the evaluation of the 14 participating solvers on 400 benchmark instances each, with a running time limit of 5 000 seconds and a verification time limit for certificates of solved unsatisfiable instances of 45 000 seconds. Taking into account only the running times of the solvers and proof checkers, this setting resulted in a total computational cost of approximately 217 CPU days.[1]

In algorithmic research, the benchmark instances and time budgets used in such competitions serve as a standard for evaluating new algorithmic ideas. Such ideas themselves often result in a combinatorial explosion of possible concrete implementations and parameter settings, such that researchers often evaluate many more variants than there are solvers participating in a competition. This means that the computational cost of the evaluations in the algorithm engineering cycle far exceeds that of running a solver competition. These evaluation costs create a bottleneck in the solver development cycle, as algorithm engineers may have to wait days or even weeks for a full evaluation of the design space of new ideas.To mitigate this cost, they may rely on a hand-picked subset of problem instances, which can lead to overfitting and optimising the solver for those specific instances.

Moreover, each of those ideas might come with parameters to tune (such as the probability of a mechanism to activate or a weight given to a heuristic). The tuning of those parameters and selection between variants of a single algorithm can be done automatically, using automated algorithm configuration (see, e.g., [12]), or manually. Whilst automating it allows to avoid the computational cost of a full evaluation, it does not provide statistical guarantees with regard to the performance of the algorithm on the full set of instances. On the contrary, automated algorithm configuration has been shown to be prone to overfitting to the instance set used for training [7]. These factors give groups with access to extensive computing resources a competitive advantage when it comes to evaluating exhaustive sets of solver variants on a large number of instances, as they can test more ideas on a wider range of instances.

The goal of our tool is to lower the computational cost of evaluating new solvers or versions thereof. Given a set of problem instances $\mathcal{I}$, the performance rank of a given solver within a set of solvers $\mathcal{A}$ is typically induced by a score based on the running times for all instances solved within a pre-specified time limit, as well as how many times that time limit was reached. Our research aims to achieve a desired level of statistical confidence in the performance rank, while minimising the total running time of the methods used to produce a benchmark by dynamically selecting a smaller number of instances. To formalise this problem, we follow the definition of the *New Solver Problem* from Fuchs *et al.* [9], reformulated here with greater precision in Definition 1.

▶ **Definition 1** (New Solver Problem). *Given a set of solvers $\mathcal{A}$, a set of problem instances $\mathcal{I}$, a performance metric $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$ with τ an upper bound, e.g., induced by a limit on running time, and a new solver $\hat{A} \in \mathcal{A}$, iteratively select benchmark instances $I \in \mathcal{I}$ and evaluate $r(\hat{A}, I)$ to reach a threshold statistical confidence C in predicting the rank of $\hat{A}$, while minimising the total running time used.*

Several methods to dynamically select instances and to estimate the confidence are described in Appendix A.

[1] Additional running time costs arise from decompressing instances, file I/O and verifying the models found for satisfiable instances, as well as other overheads.

In recent years, several methods have been developed to reduce the evaluation cost of a new algorithm [9, 17]. Those methods rely on the careful selection of a subset of instances that provides sufficient statistical evidence to decide which algorithm performs best.

Matricon *et al.* [17] proposed several methods that combine two mechanisms: an instance selector that selects instances based on their ability to discriminate between two given algorithms, and a stopping criterion that efficiently decides when sufficient information has been collected to make a decision. These methods have been shown to reduce the computing time needed to compare algorithms to less than a third of a full evaluation. However, they are focused on the comparison of two algorithms and do not directly apply to the evaluation of a new version of an algorithm against a large set of state-of-the-art solvers. A follow-up work [2] showed that those methods are also efficient when comparing variants of a single algorithm.

Fuchs *et al.* [9] proposed a method inspired by active learning to select instances that are informative for the evaluation of a new algorithm. This method has been shown to be effective in reducing the number of instances needed to reach the correct ranking of an algorithm in a competition, and ranked correctly 90% of the solver pairs using 5% of the resources needed for a full evaluation. However, this method lacks an easily accessible implementation.

Existing benchmarking tools address important but complementary parts of the workflow. Specifically, `runsolver` [18] and BenchExec [5] focus on controlled execution and resource management, while GBD [14] supports benchmark and metadata management, and Sparkle [20] as well as OptiLog [1] support broader benchmarking and meta-algorithmic workflows.

In contrast, the primary contribution of `Dike Benchmarker` is to provide ready-to-use implementations of statistically grounded, cost-efficient benchmarking methods, such as those proposed by Matricon *et al.* [17] and Fuchs *et al.* [9]. More specifically, `Dike Benchmarker` provides a modular framework that allows practitioners to obtain quick feedback on new solver ideas and researchers to develop and compare new benchmarking methods.

3 Architecture of the Tool

`Dike Benchmarker`, named after "Dike" the greek goddess of fair judgement, is organised in a modular system with clear separation of concerns. It is available on GitHub at `https://github.com/ADA-research/DikeBenchmarker`. Figure 1 illustrates the main components and their interactions. The central `benchmarkingmethods` module orchestrates the benchmarking process by coordinating instance selection and stopping criteria. This is supported by specialised adaptor modules: `benchmarkadaptors` for accessing benchmark instances, `solveradaptors` for running solvers, `infrastructureadaptors` for managing different execution environments and `dataadaptors` for handling data persistence. Finally, `resultconsumers` processes the results of the benchmarking process. This modular design allows researchers and developers to extend the tool by implementing custom methods for instance selection, stopping criteria and data handling.

Benchmarking Method

The `benchmarkingmethods` module is the central component of our tool. Its basic function is to provide means for returning the next instance on which a solver should be run and to decide when to stop the evaluation. The module provides abstract base classes to implement methods for instance selection and stopping criteria, including means for combining them in a flexible way. The tool already implements the best performing methods from previous works [17, 2]. A detailed description of the methods is included in Appendix A.

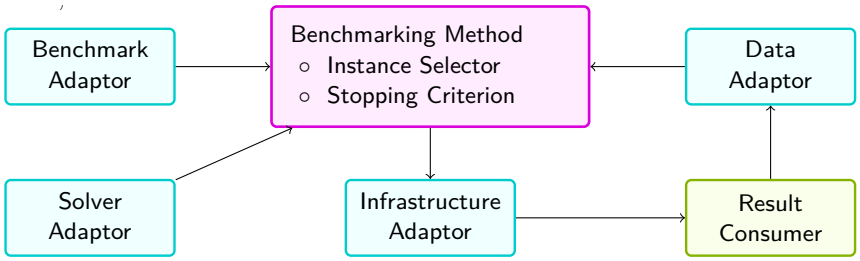


Figure 1 Architecture diagram of `Dike Benchmarker`.

Benchmark Instance Adaptors

The `benchmarkadaptors` module provides the necessary abstractions to implement adaptors for benchmark instances as well as means for making them available locally for the benchmarking process. It is designed to be agnostic to the actual format of the instances and to be able to access them from different sources, such as local files, online repositories or generators. For SAT instances, an adaptor that fetches instances from the online repository of the Global Benchmark Database (GBD) [14] and caches them locally is implemented and ready to use. The benchmark instance set can be restricted to user-defined subsets, enabling separate evaluations on subdomains (for example, domain partitions available in benchmark metadata).

Solver Adaptors

The `solveradaptors` module provides access to solvers as well as means to configure and run them. Some solvers come with multiple binaries each with their own set of command-line arguments, and a script that ties them together. Output certificates proving satisfiability or unsatisfiability need to be verified using dedicated checkers which can differ between solvers. To handle this complexity, our tool allows users to configure how to run each solver and which checker to use for it. It also provides a way to wrap solver and checker runs using tools such as `runsolver` [18] in order to impose resource limits.

Infastructure Adaptors

The `infrastructureadaptors` module provides adaptors to run solvers under different execution environments. While running solvers on a local machine is straightforward, running them on a cluster can be more complex, and the mechanisms for doing so can differ from one cluster to another. Our tool is designed to be agnostic to the actual environment, and to be able to run solvers on any of them, providing abstractions for implementing execution environment adaptors and means for running solvers in them. Currently, adaptor implementations for local machines and for clusters using the `SLURM` workload manager have been implemented and are ready to use.

Data Adaptors

Non-trivial benchmarking methods typically require data to support the benchmarking process. This data can consist of running times of other solvers, features of the problem instances, or any other kind of potentially useful information. The reference data can be configured, including filtered subsets, to analyse method robustness under different conditions. The `dataadaptors` module provides abstractions to enable uniform access to data from various

sources, including local files, online repositories and databases. Implementations for `CSV` files and for `SQLite` databases have been implemented and are ready to use. We also provide our own data source in the form of an `SQLite` database, which is stored in an online repository and can be used directly to guide benchmarking processes.

Result Consumers

The `resultconsumers` module provides glue code for handling the results of completed solver runs. The results of the benchmarking process can be used in various ways, such as updating a database or the data underlying the instance selection methods, or providing feedback to the user. This module provides abstractions for implementing adaptors for result consumers and for linking them together.

4 Example Use Case

As an example use case, we applied `Dike Benchmarker` to the performance data from main tracks of the 2023 and 2024 SAT Competitions [3, 11], as available in GBD [14]. Following the question formulated in Section 2, we aimed to evaluate how efficiently and accurately the methods implemented in our tool can predict the rank of a single algorithm in those competitions.

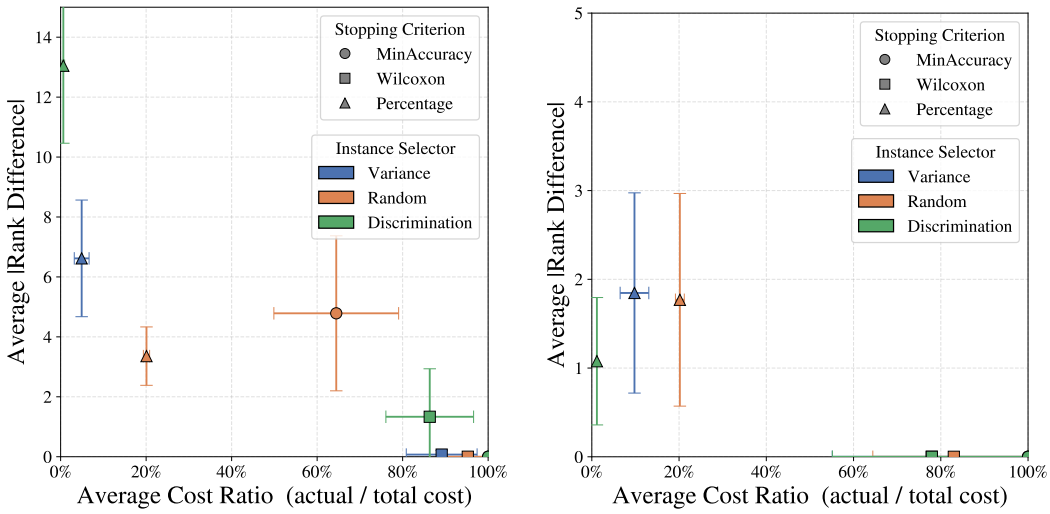
Goal. $\mathcal{A}$ is the set of algorithms evaluated for the SAT competition and $\mathcal{I}$ is the set of benchmark instances used during the competition. For a new algorithm $\hat{A}$, we attempt to find the correct ranking of $\hat{A}$ based on the running time data of all algorithms $A \in \mathcal{A}$, while minimising the cost of running $\hat{A}$ on the instances in $\mathcal{I}$.

Experiments. We applied each combination of selection method and stopping criteria described in Appendix A. For each algorithm $\hat{A} \in \mathcal{A}$, we evaluated the accuracy and the resources required when predicting its ranking based on the running time data of all other algorithms $A \in \mathcal{A} \setminus \hat{A}$. Figure 2 shows the accuracies thus obtained against the time spent for reaching it.

Figure 2 reveals a clear accuracy–cost trade-off across all method combinations and both competition datasets. Points closer to the bottom-left are preferable, since they achieve lower rank error at lower evaluation cost. In general, earlier stopping reduces cost but increases error, whereas configurations that approach near-perfect rank accuracy typically consume a large fraction of the full benchmark budget.

Among stopping criteria, the Wilcoxon-based criterion consistently reaches average rank differences close to zero, using approximately 80% of the full evaluation cost. The Minimum Accuracy criterion also achieves near-zero error, but typically requires $80 - 100\%$ of the total cost, offering no meaningful resource savings. The percentage-based criterion, while cheapest ($5 - 20\%$ of running time), incurs substantial ranking error, making it suitable as a rough, low-confidence estimate.

Regarding instance selectors, the discrimination-based and variance-based methods both reduce the evaluation cost compared to random selection, but typically reduce the average ranking accuracy. The methods exhibit consistent behaviour across both competition years. Notably, rank errors are considerably lower on the 2024 data across all configurations, which is likely due to a difference in the number of participating solvers (42 in 2023 *vs* 13 in 2024).



(a) 2023 SAT Competition (main track). (b) 2024 SAT Competition (main track).

Figure 2 Rank accuracy against cost trade-off for different instance selectors and stopping criteria. Error bars show 95% CI; bottom-left is ideal.

Overall, these results show that the methods implemented in `Dike Benchmarker` can provide useful ranking estimates at a reduced fraction of the full competition cost, while making the associated accuracy–cost trade-off explicit. The percentage stopping criterion is suitable for quick, low-confidence feedback during early prototyping, whereas the Wilcoxon criterion paired with an informed instance selector supports higher-confidence evaluation at a substantially higher cost.

Code Example. Listing 1 illustrates how to obtain the previously described results. It ranks a new solver with 95% confidence in the main track of the 2024 SAT Competition, using the Wilcoxon stopping criterion and random instance selection, running locally on a single core. This code compares the median performance of the new solver with respect to the other solvers of the competition. Therefore when it stops, the rank assigned by `Dike Benchmarker` is the one given by the median performance of the new solver on the instances it has been run on.

5 Conclusions and Future Work

In this work, we described `Dike Benchmarker`, a tool that allows developers of automated reasoning solvers to quickly obtain a rank for their new (version of) solver among state-of-the-art solvers on a set of benchmark instances of interest. We showed that the benchmarking methods currently included in `Dike Benchmarker` can provide informative estimates of comparative solver performance, with substantial savings possible in lower-confidence settings and higher-confidence estimates requiring a larger fraction of the resources needed for a full evaluation.

While the current implementation uses model-free instance selection methods, we intend to include more in the future, such as methods based on active [9] or unsupervised learning [6]. We are also planning a light API version of the tool, to allow it to be called from

Listing 1 Example code for `Dike Benchmarker`.

```
competition = "main2024"
solver_id = "my_solver_id"
confidence = 0.95
# Load prior data from existing database
adap = SqlDataAdaptor("my_db.db")

# Load prior data from database (here a competition)
solvers_challenged = adap.get_competition_solver_id(competition)
benchmark_ids = adap.get_competition_instance_id(competition)
# Create benchmarking method
benchmarker = Benchmarker(
        RandomInstanceSelector(benchmark_ids, solver_id, seed=1),
        WilcoxonStoppingCriterion(benchmark_ids, solver_id,
                solvers_challenged, confidence, adap),
        benchmark_ids,
        solver_id)
# Load files locally
solver_adaptor = SolverAdaptor()
solver_adaptor.read_registry("./solvers/sat/")
instance_adaptor = SATInstanceAdaptor("./instances/sat/",
        "./instances/cnf_data.db")
# Now make a runner so that we actually run something
runner = LocalRunner(solver_adaptor, instance_adaptor, parallel=1)
runner.run([benchmarker])
```

other benchmarking platforms. In particular, the integration of the selection methods into Sparkle [20] will allow further development in the direction of frugal algorithm selection [15] and algorithm configuration with instance selection [2].

While adapting the tool to other domains is already supported at the level of configuration – users can change data sources, benchmark sets, solver sets, and execution infrastructure without modifying the core benchmarking pipeline – we plan to add explicit support for additional types of NP-hard problems such as planning [19] or SAT Modulo Theories (cf. `SMT-LIB` [4]). In the current implementation, however, the evaluation target is a running-time-derived ranking. Supporting exchangeable ranking functions and alternative performance metrics while reusing the same benchmarking-method code is part of future work. For optimisation settings, this includes future support for domain-specific quality metrics and anytime-style ranking criteria.

In the longer term, it would be useful to support logging the execution environment in which given solvers are evaluated (in terms of hardware and software), and our tool already includes the ground work for implementing this functionality. Furthermore, in the current version, to be able to obtain the ranking of a new algorithm, the developer needs to run the baseline solvers on the complete set of instances of interest in the same execution environment as the new solver. We plan to develop approaches to avoid having to collect all performances on the same machine since this would also allow for a large gain in terms of computing resources. However, this involves addressing some open research question in the context of ensuring meaningful and fair performance comparisons.

Before concluding, we would like to acknowledge an important limitation of this and any work relying on rankings. Recent research has shown that traditional rankings in competitions are highly sensitive to the choice of benchmark instances. For example, Fawcett *et al.* [8] showed that in the SAT competition 2016, 17 solvers were statistically tied for first place, and Van Gelder [21] proposed an alternative ranking strategy accounting for ties in cases were the difference between solvers is not statistically significant. While significant

advances have been achieved since then in terms of transparency and redundancy checks on the competition instance set [13], the stability of a ranking still depends on the selected, underlying composition of classes of benchmark instances. Thus, when selecting competition data as input, the obtained ranking reflects the performance of the solver on that competition and offers no guarantees when it comes to other instances and future competitions. Active research in this direction exists and approaches to combine them with efficient and sustainable benchmarking will be included in the future versions of `Dike Benchmarker`.

References

1 Josep Alòs, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. OptiLog V2: Model, Solve, Tune and Run. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2022. doi:10.4230/LIPIcs.SAT.2022.25.

2 Marie Anastacio, Théo Matricon, and Holger Hoos. Instance selection for configuration performance comparison. In *ECMLPKDD Workshop on Meta-Knowledge Transfer*, pages 11–23. PMLR, 2022.

3 Tomas Balyo, Marijn Heule, Ashlin Iser, Matti JÃđrvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Helsinki, 2023.

4 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2016. URL: https://www.SMT-LIB.org.

5 Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. doi:10.1007/s10009-017-0469-y.

6 Gjorgjina Cenikj, Ryan Dieter Lang, Andries Petrus Engelbrecht, Carola Doerr, Peter KoroÅąec, and Tome Eftimov. Selector: selecting a representative benchmark suite for reproducible statistical comparison. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2022.

7 Katharina Eggensperger, Marius Lindauer, and Frank Hutter. Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research JAIR*, 64(1):861–893, 2019. doi:10.1613/jair.1.11420.

8 Chris Fawcett, Mauro Vallati, Holger H. Hoos, and Alfonso E. Gerevini. Competitions in ai – robustly ranking solvers using statistical resampling, 2023. doi:10.48550/arXiv.2308.05062.

9 Tobias Fuchs, Jakob Bach, and Ashlin Iser. Active learning for sat solver benchmarking. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–425, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-30823-9_21.

10 Ian P. Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Glenna F. Nightingale, and Peter Nightingale. Discriminating instance generation for automated constraint model selection. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP*, volume 8656 of *Lecture Notes in Computer Science*, pages 356–365. Springer, 2014. doi:10.1007/978-3-319-10428-7_27.

11 Marijn J.H. Heule, Ashlin Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Helsinki, 2024.

12 Holger H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous Search*, pages 37–71. Springer, 2012. doi:10.1007/978-3-642-21434-9_3.

13 Ashlin Iser. Benchmark Compilation for SAT Competition 2023. In *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, 2023.

14 Ashlin Iser and Christoph Jabs. Global benchmark database. In *International Conference on Theory and Applications of Satisfiability Testing, SAT*, 2024. doi:10.4230/LIPIcs.SAT.2024.18.

15 Erdem Kuş, Özgür Akgün, Nguyen Dang, and Ian Miguel. Frugal Algorithm Selection. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2024.38.

16 Théo Matricon, Mathieu Acher, Helge Spieker, and Arnaud Gotlieb. Efficiently ranking software variants with minimal benchmarks. *arXiv preprint arXiv:2509.06716*, 2025. doi:10.48550/arXiv.2509.06716.

17 Théo Matricon, Mario Anastacio, Nathanaël Fijalkow, Laurent Simon, and Holger H Hoos. Statistical comparison of algorithm performance through instance selection. In *International Conference on Principles and Practice of Constraint Programming (CP 2021)*, 2021. doi:10.4230/LIPIcs.CP.2021.43.

18 Olivier Roussel. Controlling a solver execution with the runsolver tool: System description. *Journal on Satisfiability, Boolean Modelling and Computation*, 7(4):139–144, 2011. doi:10.3233/SAT190083.

19 Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel FiÅąer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. The 2023 International Planning Competition. *AI Magazine*, 45(2):280–296, 2024. doi:10.1002/aaai.12169.

20 Koen van der Blom, Holger H. Hoos, Chuan Luo, and Jeroen G. Rook. Sparkle: Towards accessible meta-algorithmics for improving the state of the art in solving challenging problems. *IEEE Transactions on Evolutionary Computation*, 26(6):1351–1364, 2022. doi:10.1109/TEVC.2022.3215013.

21 Allen Van Gelder. Careful ranking of multiple solvers with timeouts and ties. In *Proceedings of Theory and Applications of Satisfiability Testing, SAT*. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-21581-0_25.

22 Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019. doi:10.3233/SAT190123.

A Included Methods

We implemented methods stemming from our prior works on instance selection [2, 9, 17]. Following a similar approach, we divided the problem into two main components: an instance selector and a stopping criterion. The instance selector selects the next instance on which $\hat{A}$ should run. Then, the information about the run is given to the stopping criterion, which decides if sufficient statistical evidence has been collected to decide the ranking of the solver according to the threshold C set by the user.

Instance Selection Methods

The current version of our tool includes the best performing methods from Matricon *et al.* [17], which also showed strong performance in the context of comparing configurations of a single solver [2]. They do not require to learn an empirical performance model and can thus be used widely. The included methods are the following:

Random Selection. This instance selectors selects an instance uniformly at random.

Discrimination-based Selection. This instance selector, inspired by the work of Gent *et al.* [10], pre-ranks all candidate instances before any runs are performed and submits them in decreasing order of score. For each instance I, a score is computed from the vector of running times $\mathbf{R}(I) = (t_1, \ldots, t_k)$, where t_n is the observed running time of the n^{th} solver from all k prior solvers in $\mathcal{A}$:

$$\text{score}(I) = \frac{|\{n \mid t_n \geq \rho \cdot \min(\mathbf{R}(I))\}|}{\text{Mean}(\mathbf{R}(I))},$$

where $\rho \geq 1$ is a discrimination threshold (with default $\rho = 1.2$). The numerator counts the number of solvers whose running time on I is at least ρ times the best performing solver on that instance. It captures instances on which a some solvers perform poorly. Dividing by the mean running time normalises for instance difficulty, penalising instances that are uniformly expensive and thus provide little discriminative information relatively to their computational cost.

Variance-based Selection. This instance selector pre-ranks all candidate instances before any runs are performed and submits them in decreasing order of score. Using the same vector of solver running times $\mathbf{R}(I)$ defined above, the score is defined as the relative variance:

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

Stopping Criteria

The current version of our tool includes one criterion that we designed specifically for the New Solver Problem, and two criteria adapted from Matricon *et al.* [17], where they were designed for the specific case of the New Solver Problem with only two solvers. The two methods identified in that study are a simple baseline that showed strong results and the best performing method.

MinAccuracy. This method computes the pairwise concordance rate, equivalent to the normalised concordant-pairs count used in Kendall's τ test (without penalty for ties). We compute how solvers in $\mathcal{A}$ would rank if we considered only the benchmark instances evaluated using the new solver $\hat{A}$. Then, we measure the accuracy of the ranking based on the fraction of solver pairs whose relative order in this partial ranking matches the ground-truth ranking, *i.e.*:

$$\text{accuracy} = \frac{|(i,j) : i < j \text{ in ground-truth and } i < j \text{ in partial ranking}|}{\binom{n}{2}}.$$

The benchmarking process stops when this accuracy reaches the given threshold.

Percentage. This is a baseline stopping criterion that stops when a certain percentage of all the instances have been run, set by default at 20%, following the results of prior work [17].

Wilcoxon. The best-performing method in our prior work [17] was based on a Wilcoxon statistical test with the null hypothesis that the performance of two solvers are identical. To adapt this method to the New Solver Problem, we apply the test as follows: Given the new solver $\hat{A}$, the known solvers from $\mathcal{A}$ are sorted according to their performance. Starting from the weakest solver $A \in \mathcal{A}$, a paired comparison is done between A_0 and $\hat{A}$. If the Wilcoxon

test rejects the null hypothesis with a confidence threshold set by the user (and a default value of 0.05), a decision is taken: Either the mean performance of $\hat{A}$ is better, A is discarded, and the process continues with the next solver from $\mathcal{A}$, or the mean performance of $\hat{A}$ is worse, and the returned ranking is one below the ranking of A. If no solvers are left in $\mathcal{A}$ and A is discarded, $\hat{A}$ is assigned rank 1.