Automated Tuning and Portfolio Selection for SAT Solvers

Master's Thesis of

Tobias Fuchs

At the Department of Informatics,
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Peter Sanders
Advisors:	Dr. Markus Iser
	M.Sc. Jakob Bach

Time Period: November 11th, 2022 – May 31st, 2023

Zusammenfassung

Das NP-vollständige aussagenlogische Erfüllbarkeitsproblem (SAT) ist ein nützliches Werkzeug für die Modellierung verschiedener Anwendungen von der Kryptographie [66] bis hin zu Planungsaufgaben [34]. In der Regel schneiden verschiedene SAT-Solver in verschiedenen Teilbereichen des Instanzraums am besten ab. Die Arbeit befasst sich mit der instanzspezifischen Algorithmenkonfiguration, die Algorithmenkonfiguration [42, 5] und -selektion [41] miteinander verbindet. Wir wollen die beste SAT-Solver-Konfiguration, einschließlich des zu verwendenden Solvers, für jede Partition des Instanzraums finden. Allerdings gibt es eine gegenseitige Abhängigkeit zwischen der Suche nach guten Konfigurationen für alle Partitionen und der Suche nach guten Partitionierungen, die gut konfiguriert werden können. Um dieses Problem zu lösen, schlagen wir einen neuartigen evolutionären Algorithmus vor, der Entscheidungsbäume zur Partitionierung des Instanzraums verwendet. Er erfüllt mehrere wünschenswerte Kriterien, darunter die eingebaute Interpretierbarkeit von Entscheidungsbäumen und Robustheit, die experimentell gezeigt wird. Darüber hinaus hat unser evolutionärer Ansatz eine vergleichbare Leistung wie der aktuelle Algorithmenkonfigurator Optuna [5].

Abstract

The NP-complete propositional satisfiability problem (SAT) is a useful tool for modeling various kinds of applications ranging from cryptography [66] to planning [34]. Usually, different SAT solvers perform best on different subregions of problem instance space. We address the per-set algorithm configuration problem, which combines both, algorithm configuration [42, 5] and algorithm selection [41]. We want to find the best SAT solver configuration, including which solver to use, for each partition of instance space. There is a recurrence between finding good configurations for all partitions and finding good partitionings that can be tuned well, though. To tackle this problem, we propose a novel evolutionary algorithm that evolves decision trees for partitioning instance space. It fulfills several desirable criteria including the built-in interpretability of decision trees and robustness, that we show experimentally. Furthermore, our evolutionary trees approach has comparable performance to the state-of-the-art algorithm configurator Optuna [5].

Contents

1 Introduction

The propositional satisfiability problem (SAT) is the archetypal NP-complete problem. It has various applications ranging from planning and scheduling [34] over model checking [20] to cryptoanalysis [66]. Usually, different SAT solvers and their configurations perform best on different subregions of problem instance space. Tuning SAT solvers [42, 41, 27], creating portfolios of complementary solvers [11, 6], as well as selecting a solver out of a portfolio for each problem instance [69, 25] are well-established in this field. Tuning algorithms is also known as algorithm configuration. It seeks to find the best configurations for a solver on a given benchmark. Portfolio creation tries to exploit the complementarity of solvers by bundling a small number of solvers that, together, solve more instances than individually. Finally, algorithm selection has the goal of selecting an algorithm out of a portfolio for each problem instance. It tries to predict which algorithm is best at solving a particular instance. This is also known as per-instance algorithm selection [2].

Advances in the field of automated machine learning (AutoML) show the benefit of combining algorithm configuration as well as selection [67, 30, 51] within the framework of the Cash problem (combined algorithm selection and hyperparameter optimization), i.e., find me the best algorithm with the best configuration out of a set of possible algorithms and configurations in the least amount of time. Thornton et al. [67], for example, show that their implementation outperforms a greedy baseline, i.e., running all algorithms with a standard hyperparameter setting and then individually tuning the most promising candidates. Furthermore, per-instance algorithm configuration (PIAC) is a superset of this. Its goal is, similar to ours, to find the best algorithm or configuration for each problem instance, respectively. An example of this is the ISAC approach [48]. Kadioglu et al. [48] cluster the instance space based on problem-instance features and then tune solvers for each cluster with established general algorithm configurators, e.g., SMAC [42]. This sequential cluster-then-tune process assumes however that instances that are close in feature space are solved best by the same configurations. There are mainly three problems with this. First, it is unclear whether this is true in general as this is mainly dependent on the employed features. Second, the instance feature space is highly dimensional. Finding clusters in high-dimensional data is difficult because of the *curse of dimensionality* [29, 4]: The higher the dimension, the more similar feature vectors seem. Third, the initial clustering is static. It cannot be altered, i.e., another grouping of instances that is informed by already obtained runtimes could lead to better configuration improvements overall.

1.1 Contributions

In this work, we present a novel tuning and algorithm selection approach. Given a fixed portfolio of SAT solvers, we propose an evolutionary algorithm that tries to choose the best algorithm with the best configuration for each region of problem instance space. We initially partition instances into groups. This partitioning is subject to change throughout the algorithm. Then, we tune the portfolio members on their assigned partitions of the benchmark set. Finally, we adapt the partitioning of the instance space informed by how well solvers can be tuned on it and repeat the aforementioned steps. The evolutionary algorithm tackles this recurrence between partitioning instance space and how well instance partitions can be tuned. The problem we solve is specified in Definition 1.1. Note that different SAT solver choices are implicitly included within the configuration space Θ.

Definition 1.1 (Per-Set Algorithm Configuration). *Given solver configurations Θ, instances $\mathcal{I}$, and a runtime acquisition function $\mathrm{r} : \Theta \times \mathcal{I} \to \mathbb{R}^+$, find an instance partitioning $\pi : \mathcal{I} \to \{1, \dots, k\}$ and a per-set algorithm configuration policy $\psi : \{1, \dots, k\} \to \Theta$ that minimizes the overall runtime $R_{\psi,\pi} := \sum_{i \in \mathcal{I}} \mathrm{r}(\psi(\pi(i)), i)$. The number k determines the maximal number of partitions.*

Our approach satisfies several desirable criteria (cf. Table 1): Since solving SAT problem instances is generally expensive, we want to focus on approaches that are *computationally feasible*. This means that we want to also minimize the accumulated cost of evaluating r within the algorithm since SAT solving is known for costly benchmarking [13]. Also, rather than statically clustering instance space and then finding good configurations for each cluster, we gradually evolve the partitioning of instances based on how well solvers can be tuned on these partitions (*non-static partitioning*). Furthermore, we want our per-set algorithm configuration policy to be *robust* so that it can be applied to new unseen instances and *explainable* so that statements about which configuration options are beneficial for what kind of instances are possible.

We evaluate our approach on a subsample of the SAT Competition 2022 Anniversary Track dataset [13]. The employed dataset has been randomly sampled using stratification regarding the instance families. It consists of 1000 instances. Furthermore, we make use of the state-of-the-art Kissat solver [33, 19] and SeqFrost solver [61]. Analysis has shown that they form a good portfolio (cf. Section 5.5). Also, we restrict ourselves to binary configuration options since they have shown a relatively high impact on SAT solver performance.

1.2 Outline

The subsequent section (Section 2) briefly introduces definitions and notations used throughout this work. Section 3 summarizes existing work in relevant areas including SAT-solver portfolios, algorithm selection, and algorithm configuration. Thereafter, we present our novel genetic algorithm in Section 4. Section 5 is about our experimental design and the conducted experiments. Section 6 shows evaluation results. Finally, we conclude this thesis by presenting takeaways as well as listing possible future work in Section 7.

2 Preliminaries

This section presents various definitions and explanations that aim to improve the reader's understanding of this work. In Section 2.1, we present helpful definitions regarding algorithm selection and configuration. Besides that, we briefly discuss the topic of population-based evolutionary algorithms in Section 2.2.

2.1 Algorithm Selection and Configuration

Although algorithm selection and algorithm configuration are closely related, they have separate goals. Figure 1 gives an overview of the relationships. An arrow indicates that a framework generalizes to another one. The per-instance algorithm configuration framework is the most general one. Throughout the section, we will define each of the depicted terms. Apart from the definitions, an actual implementation of those frameworks should also fulfill the requirements listed in Section 1.

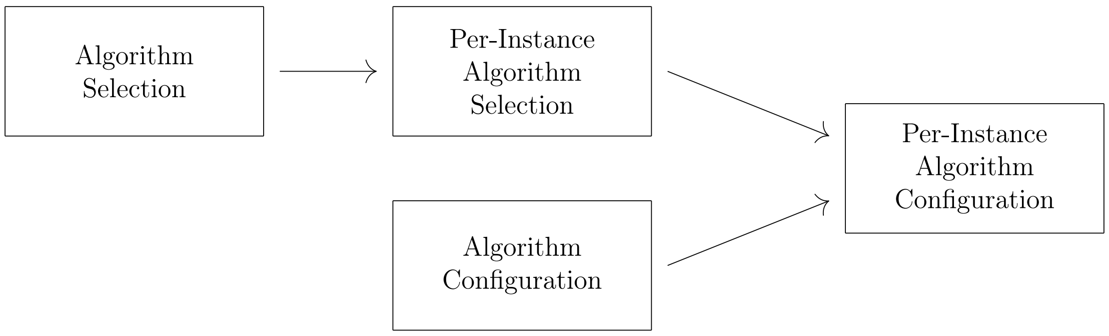

Algorithm Selection
Per-Instance Algorithm Selection
Per-Instance Algorithm Configuration
Algorithm Configuration

Figure 1: Relationship between algorithm selection and algorithm configuration. An arrow indicates that a framework generalizes to another one. The depiction is inspired by Kerschke et al. [49].

2.1.1 Algorithm Selection

Algorithm selection (AS) has the goal of selecting a single algorithm out of a fixed set to solve instances from a given distribution. The algorithm selection problem is specified in Definition 2.1.

Definition 2.1 (Algorithm Selection). *Given a set of solvers $\mathcal{S}$, problem instances $\mathcal{I}$, and a runtime acquisition function $\mathrm{r}\colon \mathcal{S} \times \mathcal{I} \to \mathbb{R}^+$, find a single problem solver $s \in \mathcal{S}$ that minimizes the overall runtime $R_s^{(AS)} := \sum_{i \in \mathcal{I}} \mathrm{r}(s, i)$.*

2.1.2 Per-Instance Algorithm Selection

Per-instance algorithm selection (PIAS) is an extension of algorithm selection. Its goal is to choose an algorithm out of a fixed set of algorithms for *each* problem instance individually. The per-instance algorithm selection problem is specified in Definition 2.2. Algorithm selection can be considered a specialization of per-instance algorithm selection with constant ψ.

Definition 2.2 (Per-Instance Algorithm Selection)**.** *Given a set of solvers $\mathcal{S}$, problem instances $\mathcal{I}$, and a runtime acquisition function* $\mathrm{r} : \mathcal{S} \times \mathcal{I} \to \mathbb{R}^+$*, find a per-instance selection policy* $\psi : \mathcal{I} \to \mathcal{S}$ *that minimizes the overall runtime* $R_\psi^{(PIAS)} := \sum_{i \in \mathcal{I}} \mathrm{r}(\psi(i), i)$.

2.1.3 Algorithm Configuration

The goal of *algorithm configuration* (AC) is somewhat different from algorithm selection: Given a single problem solver, the goal is to find its best-performing configuration. The algorithm configuration problem is specified in Definition 2.3. Although there is no relation between algorithm selection and configuration (regarding generalization) in Figure 1, one could transform an algorithm selection into a configuration problem and vice versa. To do so, the set of problem solvers among which to choose can be combined into a single meta-solver using configuration options Θ to pick the underlying algorithm.

Definition 2.3 (Algorithm Configuration)**.** *Given a set of configurations Θ, problem instances $\mathcal{I}$, and a runtime acquisition function* $\mathrm{r} : \Theta \times \mathcal{I} \to \mathbb{R}^+$*, find a configuration* $\theta \in \Theta$ *that minimizes the overall runtime* $R_\theta^{(AC)} := \sum_{i \in \mathcal{I}} \mathrm{r}(\theta, i)$.

2.1.4 Per-Instance Algorithm Configuration

Per-instance algorithm configuration (PIAC) is the most general framework out of the shown ones (cf. Figure 1). Given a problem solver, its goal is to find the best-performing configuration for *each* problem instance individually. The per-instance algorithm configuration problem is specified in Definition 2.4.

Definition 2.4 (Per-Instance Algorithm Configuration)**.** *Given a set of configurations Θ, problem instances $\mathcal{I}$, and a runtime acquisition function* $\mathrm{r} : \Theta \times \mathcal{I} \to \mathbb{R}^+$*, find an algorithm configuration policy* $\psi : \mathcal{I} \to \Theta$ *that minimizes the overall runtime* $R_\psi^{(PIAC)} := \sum_{i \in \mathcal{I}} \mathrm{r}(\psi(i), i)$.

2.1.5 Per-Set Algorithm Configuration

Because choosing configurations for each instance is prone to over-fitting and not robust (cf. requirements in Section 1 and 3), we also consider a simplification of the per-instance algorithm configuration framework: *Per-set algorithm configuration.* This is the exact problem that we want to tackle in this work. It has already been introduced in Definition 1.1 and is further discussed in Section 4. Having a dedicated instance partitioning π allows for more robust results since only one configuration is chosen for each partition. One may also regard per-set algorithm configuration as a version of per-instance algorithm configuration with a constraint on the maximal number of different configurations. Because of this, there is no separate box for it in Figure 1.

2.2 Population-based Evolutionary Algorithms

To handle the reciprocal dependency between partitioning the instance space and finding good configurations for them, we make use of a population-based evolutionary algorithm. Refer to Section 4 for details. This section briefly introduces such algorithms to ease the understanding of this work's main part. Many algorithmic ideas use evolutionary principles, e.g., evolution strategies, differential evolution, simplex search, etc. Compare for example the surveys by Bartz-Beielstein et al. [14] or by Bäck and Schwefel [12].

Population-based evolutionary algorithms evolve a population of individuals $\mathcal{P}^{(t)}$ over time $t \in \{1, \ldots, T\}$ using the Darwinian concept of fitness. Such algorithms follow a simple five-step procedure:

1. Generate the initial population $\mathcal{P}^{(1)}$. This is usually done randomly. Initializing the population to a heuristic solution is also common though.
2. Evaluate the fitness $\mathrm{f}\colon \mathcal{P}^{(t)} \to \mathbb{R}^{+}$ of each individual.
3. Select individuals for recombination based on their fitness and discard the least-fit individuals. This is usually done using either fitness-proportionate selection or tournament selection. Fitness-proportionate selection samples the population with the individual's fitness as weights. In contrast, tournament selection repeatedly runs tournaments between a handful of random individuals selecting the winner with a certain probability. Thereby, the tournament size determines the selection pressure. In practice, tournament selection is often preferred because it is easy to parallelize and has less stochastic noise [21].
4. Create offspring by mutation and cross-over. The goal of this step is to, either, randomly find fitter individuals by mutating some properties of selected individuals or combining good attributes of individuals through cross-over. By doing a cross-over step, we hope that the combination of good properties of two or more individuals yields better offspring. While mutation may be regarded as the *exploration* of solution space, cross-over can be described by *exploiting* properties of good individuals.
5. Repeat from step 2 until a certain stopping criterion is reached. This can be either a fixed number of iterations or a criterion based on the improvement of the individual's fitness over time. Increment t by one.

3 Related Work

Related work is divided into four sections. First, we take a brief look at portfolio creation. Second, we discuss relevant work regarding algorithm selection. Thereafter, we look at algorithm configuration, including the configuration of SAT solvers. Finally, we give an overview of per-instance algorithm configuration. It can be considered as a generalization of, both, algorithm selection and configuration.

3.1 Portfolio Creation

By the term portfolio, we generally mean a small set of complementary algorithms [49]. Bach et al. [11] present an optimization approach for finding optimal portfolios of SAT solvers with different sizes. Amadini et al. [6] also analyze portfolios of different sizes for constraint-satisfaction problem solvers. Both studies show the diminishing advantages of gradually bigger portfolio sizes. Bach et al. [11] have also shown that for bigger portfolios selection models become less accurate.

Within the SAT domain, there are mainly two strategies to profit from portfolios, i.e., schedules [36, 23] and per-instance selection [69, 48]. Schedules interleave runs of different problem solvers [36, 23] as some solvers might make more progress in a particular solving phase than others. In contrast to that, per-instance selection chooses a single algorithm to use for each problem instance. Within this work, we also limit ourselves to the latter. Algorithm selection makes use of, both, supervised [69] as well as unsupervised machine-learning methods [48]. We take a look at them in the subsequent section.

3.2 Algorithm Selection

Algorithm selection (AS) has the goal of selecting one algorithm out of a set of algorithms to solve a problem. It is closely related to algorithm configuration. Algorithm configuration can be thought of as a more general and flexible concept though. Both involve minimizing the cost of acquiring the best algorithm. Algorithm selection has been studied as early as 1976 by Rice [64]. In a more recent survey, Raschka [63] discusses the topic of algorithm selection in the field of machine learning. Thereby, he especially focuses on statistical tests for comparing model performance as well as on model evaluation.

An important extension of algorithm selection is per-instance algorithm selection (PIAS). Here, one algorithm is selected for each problem instance. Often, different algorithms perform best on different regions of the instance space. A survey by Kerschke et al. [49] provides an overview of per-instance algorithm selection approaches. This includes research within propositional satisfiability solving. A survey by Hoos et al. [41] provides an overview of PIAS dedicated to SAT.

One of the first successful per-instance algorithm selectors within SAT is the portfolio-based approach SATzilla [69]. Xu et al. [69] use a selection model to find the approximately best solver out of a portfolio of solvers for each instance. They make use of problem instance features like general instance size and graph features among others. In a more recent version [71], they use an empirical instance hardness model that estimates how difficult a particular instance is and a cost-sensitive feature model that estimates how expensive the acquisition of a particular feature is.

Kadioglu et al. [48] propose a clustering-based method called ISAC (instance-specific algorithm configuration). Although the approach proposes a strategy for per-instance algorithm configuration, it can also be used as an algorithm selector by modeling different portfolio members as configuration options. Their approach clusters instances and assigns each cluster of sufficient size a solver that showed to be best for those instances in a training phase. Given a new instance, they select a solver based on the k-nearest neighbors in feature space. This assumes that instances with similar features are solved well by similar solvers.

Collautti et al. [25] propose a portfolio approach called SNNAP (solver-based nearest neighbor for algorithm portfolios) that is a continuation of the ISAC idea. It entails both supervised as well as unsupervised methods. To be more specific, they use random forests to predict individual solver runtimes on instances and cluster instances based on the similarity of these runtimes. Given a new instance, they predict solver runtimes and pick a solver based on the best solver among the k-nearest neighbor clusters. SNNAP shows stronger and more robust results than ISAC on the SAT Competition benchmarks [25].

The tool AutoFolio [53] seeks to automate the process of per-instance algorithm selection even further. It combines the aforementioned techniques, i.e., SATzilla [69], ISAC [48], and SNNAP [25], among others using automated algorithm configuration to find the optimal hyperparameter setting.

Per-instance algorithm selectors are also an important part of *cooperative competitions*. Established competitions like the annual SAT Competition [13] rank solvers based on their performance on all benchmark instances. In contrast to that, cooperative competitions like the Sparkle Competition [54] rank solvers based on their marginal contribution [68]. Roughly speaking, this score makes use of the number of instances that are solemnly solved by a particular solver. The Sparkle Competition 2018 [54] shows that even solvers that performed suboptimally within the SAT Competition 2018 [39] have high marginal contributions to the resulting Sparkle portfolio, indicating the complementarity of solvers and the advantage of portfolio-based SAT solving.

3.3 Algorithm Configuration

Algorithm configuration (AC) has the goal of choosing the best or a close-to-best configuration of a single algorithm regarding a given benchmark. Although algorithm configuration and selection are somewhat related, the task of configuration is generally harder as it involves modeling continuous, integer, and categorical configuration parameters as well as a generally bigger configuration space. Also, configuration parameters may either represent high-level data-structure decisions, low-level tweaking, or something in between. In comparison, algorithm selection *only* has to choose among a handful of portfolio members.

The programming-by-optimization paradigm [40] argues for encoding a multitude of design choices within an algorithm's configuration options, leaving the optimization to an automatic tuner. Eggensperger et al. [27] give an overview of best practices regarding algorithm configuration. They also name common pitfalls of humans tuning algorithms manually.

ParamILS is one of the first general-purpose algorithm configurators [44, 43]. It employs iterative local search starting from the default configuration to find minima. When reaching local minima, it alters parameters randomly to escape. ParamILS also makes use of *adaptive capping*: Assuming that the algorithm with the best configuration needs runtime

t, all further experiments are not run for longer than t. Knuth [50] shows that ParamILS provides significant performance improvements over manual optimization in SAT solving.

The gender-based genetic algorithm (GGA) [8] is also a general-purpose algorithm configuration system. It uses a genetic approach to evolve configurations, employing the concept of gender to maintain diversity within the population. Pairs of configurations with appropriate genders produce offspring. GGA also applies *adaptive capping* to its evaluation phase. GGA++ [7] is an extension of GGA utilizing random forests for performance prediction. This advancement is inspired by the sequential model-based algorithm configurator (SMAC) [42, 52].

SMAC [42, 52] makes use of an empirical performance model to determine configurations of interest. Given parameter values, Hutter et al. want to predict their performance impact and run experiments with maximum expected improvement. To do so, they fit a regression model to runtime data acquired so far. Hutter et al. [42] also show that SMAC performs significantly better than ParamILS in 65 % and significantly better than GGA in 76 % of cases.

Advances in the field of automated machine learning (AutoML) also show the benefit of combining, both, algorithm selection and configuration within the framework of the Cash problem [67], which has been introduced in Section 1.

Thornton et al. [67, 51] propose the tool Auto-WEKA. Similar to SMAC [42], they make use of model-based bayesian optimization to predict the performance of an algorithm-configuration pair. They show that their implementation significantly outperforms a time-limited random grid search as well as a greedy baseline, i.e., running all algorithms with a standard hyperparameter setting and then individually tuning the most promising candidates. Feurer et al. [30] propose the tool Auto-sklearn. It extends Auto-WEKA [67] to be usable with the popular models from the *sklearn* library [62].

While ParamILS [44] and SMAC [42] use multivariate Gaussian processes to estimate the shape of hyperparameter space, the Optuna framework [5] uses tree-structured Parzen estimation. Bergstra et al. [17] show that this is a less expensive method compared to Bayesian optimization with Gaussian processes. In comparison to Gaussian kernels, a Parzen-Tree estimator maintains two separate kernel-density estimates indicating which regions have *good* hyperparameter performance and which have *bad* hyperparameter performance. By finding the point that maximizes the expected improvement in both kernel-density estimates, we are more robust in identifying regions in hyperparameter space with good performance. The Optuna framework [5] is a state-of-the-art hyperparameter-optimization tool that employs this kind of model. In Section 6.4, we use Optuna to estimate the cost of acquiring a good SAT solver configuration for given a dataset. Regarding our desired features (cf. Table 1), Optuna is *feasible* for the configuration of SAT solvers and reliably produces *robust* results. However, it does not leverage the diversity of problem instances through *partitioning* instance space, since we use it to find a single configuration with good performance regarding all instances in our dataset.

Algorithm configuration also plays a key role within the Configurable SAT Solver Challenges [45]. Established competitions like the annual SAT Competition [13] rank solvers based on their performance on all benchmark instances. Thereby, solvers are optimized on the distribution of benchmark instances within this competition. In contrast to that, the Configurable SAT Solver Challenge (CSSC) evaluates SAT solver performance after a time-limited configuration phase. This incentivizes the usage of effective configuration parameters. The CSSC makes use of the most common algorithm configuration methods

Table 1: Comparison of desired features of (per-set) algorithm configuration approaches including a simple grid search, the clustering-based ISAC [48] approach, the CMA evolutionary strategy [16], the Optuna framework [5], as well as our evolutionary trees approach.

Feature	Grid Search	ISAC [48]	CMA-ES [16]	Optuna [5]	Our approach
Feasible in Context	✗	✓	✗	✓	✓
Non-static Partitioning	✗	✗	✓	✗	✓
Robust	✓	✓	✗	✓	✓
Explainable	✗	✓	✗	✗	✓

including the aforementioned configurators, i.e., ParamILS [44], GGA [8], SMAC [42], and Optuna [5].

3.4 Per-Set / Per-Instance Algorithm Configuration

Per-instance (PIAC) and per-set algorithm configuration (PSAC) have the goal of finding the best configuration for each instance or each set of instances respectively (cf. Definition 1.1). The former is also often referred to as instance-specific algorithm configuration (ISAC). Per-instance algorithm configuration is the most general problem definition out of the presented problems. Per-instance algorithm selection can be modeled within the per-instance algorithm configuration framework by encoding different algorithm selection decisions within the configuration options of a single meta solver. However, per-instance algorithm configuration is also one of the most difficult among the presented problems as it involves searching a probably big configuration space for many instances. Also, modeling each instance individually is often prone to over-fitting. To make it more robust, we tackle the problem of per-set algorithm configuration as it tunes the algorithm on sets of instances rather than single instances. This improves performance on unseen instances.

Table 1 gives an overview of the desired features introduced in Section 1. We compare their fulfillment regarding a simple grid search, the ISAC approach [48], the CMA-ES strategy [16], as well as our approach. A full grid search provides the best configuration for each instance. Unfortunately, it is infeasible for hard combinatorial problems like SAT.

One way to tackle per-set algorithm configuration is through clustering. Kadioglu et al. [48] use clustering within their ISAC framework. We discussed it earlier in the context of algorithm selection. Their approach clusters instances in feature space and tunes each cluster of sufficient size individually. To do so, they use the general algorithm configurators introduced in Section 3.3. This assumes, though, that instances, that are close in feature space, have similar best configurations. It is unclear whether this is in general true. Apart from that, the approach has the benefit of being *runtime-efficient* and *robust* since an unseen instance can simply be assigned to a configuration using a k-nearest neighbors approach.

Based on this, Abell et al. [1] propose features for black-box problems to solve them using the clustering-based ISAC method. This includes problem definition features, e.g., the dimensionality of configuration space, as well as hill climbing features, which estimate the convexity of configuration space, among others.

Muñoz et al. [58] propose a black-box algorithm configurator with an empirical performance model (cf. SMAC [42]) using the aforementioned meta-features.

Advances in the field of *continuous* black-box optimization regarding per-instance algorithm configuration also suggest the usage of evolutionary strategies. This strategy is only adaptable to continuous configuration options, though. Belkhir et al. [16, 15] make use of a covariance-matrix-adaptation evolution strategy (CMA-ES) first introduced by Hansen et al. [38]. It evolves a multi-variate Gaussian distribution by tweaking its covariance matrix. In other words, it evolves a model with the aforementioned black-box configuration-space features for each instance by altering a multi-variate distribution. Additionally, they also use a surrogate model of the mapping between configuration space and meta-features to reduce the number of function evaluations required. However, black-box algorithm configurators still require a vast number of function evaluations. CMA-ES [16] requires in the order of 1000 function evaluations per configuration-space dimension, rendering it infeasible for hard combinatorial problems. Also, CMA-ES [16] lacks *robustness* and *explainability* as introduced in Section 1 since each instance is modeled independently. An advantage of this strategy is, though, that instances are *not statically* grouped into a cluster as with the ISAC approach [48], which theoretically prevents each instance from reaching its optimal configuration.

4 Automated Tuning and Portfolio Selection

This section describes our main contribution: an evolutionary algorithm for automatically tuning and performing portfolio selection for SAT solvers as given in Definition 1.1.

Algorithm 1 outlines our evolutionary tuning and selection approach. It takes all available configuration options Θ, all instances $\mathcal{I}$ with features, the runtime acquisition function r as well as the maximum number of iterations as input. To start the algorithm off, we initialize the population of our evolutionary algorithm in Line 1 (Section 4.1). Then, we perform several evolution steps until we reach a maximum of evolution iterations (Line 2). Each iteration consists of tuning the current population for a fixed amount of time in Line 3 (Section 4.5), computing the fitness value for each individual based on the acquired or predicted runtimes in Line 4 (Section 4.2), selecting promising individuals in Line 6 (Section 4.3), and producing offspring in Line 7 (Section 4.4). When the main loop finishes, we return the fittest individual's instance partitioning and configuration policy.

Algorithm 1: Evolution of Partitioning Trees Framework

```
Input:  A fixed portfolio of solvers with possible configuration options $\Theta$ (solver
        selection included within the configuration), Instances $\mathcal{I}$ with features
        feat: $\mathcal{I} \to \mathbb{R}^d$, Runtime acquisition function r: $\Theta \times \mathcal{I} \to \mathbb{R}^+$, Number of
        maximum iterations $T$, Maximum number of partitions $k$
Output: Instance partitioning model $\hat{\pi} : \mathbb{R}^d \to \{1, \dots, k\}$ mapping instance
        feature vectors to instance partitions, Per-set algorithm configuration
        policy $\psi : \{1, \dots, k\} \to \Theta$ assigning each instance partition a
        configuration
1 $\mathcal{P}^{(1)} \leftarrow$ initPopulation($\mathcal{I}$, feat, $k$)                                  // Sec. 4.1
2 for $t = 1$ to $T$ do
3     runtimes $\leftarrow$ tuneAndEvaluatePopulation $\left(\mathcal{P}^{(t)}, \Theta, \mathrm{r}, \mathrm{feat}\right)$             // Sec. 4.5
4     fitness $\leftarrow$ computeFitness($\mathcal{P}^{(t)}$, runtimes)                            // Sec. 4.2
5     if not last iteration then
6         $\mathcal{P}^{(t+1)} \leftarrow$ selectPopulation $\left(\mathcal{P}^{(t)}, \mathrm{fitness}\right)$                       // Sec. 4.3
7         $\mathcal{P}^{(t+1)} \leftarrow \mathcal{P}^{(t+1)} \cup$ produceOffspring $\left(\mathcal{P}^{(t+1)}, \mathrm{fitness}, \mathrm{feat}, k\right)$   // Sec. 4.4
8 $p^* \leftarrow \arg\max_{p \in \mathcal{P}^{(T)}}$ fitness($p$)
9 return ($p^*.\hat{\pi}$, $p^*.\psi$)
```

4.1 Individuals and Initial Population

Each individual consists of an instance partitioning model $\hat{\pi} : \mathbb{R}^d \to \{1, \dots, k\}$ and a per-set configuration policy $\psi : \{1, \dots, k\} \to \Theta$. The former is an extension of π in Definition 1.1. It maps instances represented by a d-dimensional feature vector to one out of k partitions. The latter is identical to ψ in Definition 1.1.

To represent the aforementioned instance partitioning model, we use decision trees. Decision trees provide the advantage of being easily modifiable, easy to interpret, and a good choice for supervised prediction tasks, although they are nowadays predominantly used within random forest ensembles [22].

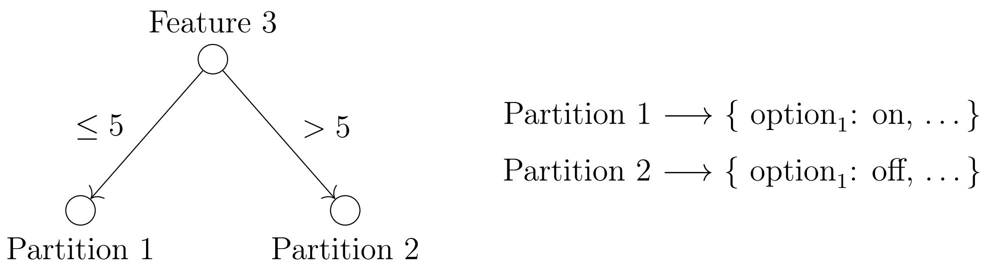

Feature 3
≤ 5
> 5
Partition 1
Partition 2
Partition 1 ⟶ { option₁: on, ... }
Partition 2 ⟶ { option₁: off, ... }

Figure 2: Depiction of an individual's components including the instance partitioning decision tree model as well as the per-set algorithm configuration policy.

Figure 2 shows an example of an instance partition tree with a per-set algorithm configuration policy. Thereby, the instance partitioning decision tree only consists of a single split. Instances with a value at most 5 regarding the third feature are assigned to the first instance partition. Instances with a value above 5 regarding the third feature are assigned to the second instance partition. The per-set algorithm configuration policy on the right-hand side of the figure then maps each instance partition to a configuration.

The individuals' instance partitioning models are modified in Line 7 of Algorithm 1. Mutating the instance partitioning decision tree is discussed in Section 4.4. The per-set algorithm configuration policy is modified in Line 3. Tuning the algorithm configuration of each partition is discussed in Section 4.5. The first line of Algorithm 1 initializes both by creating the genesis population.

We make use of three different strategies for creating the initial population. First, we can start with a trivial decision tree with no splits. Having only a single instance partition is equivalent to algorithm configuration (cf. Definition 2.3). Incrementally, more complex decision trees are formed through mutation (cf. Section 4.4). Second, we can initially cluster the instance feature space into k clusters. Thereafter, we train a decision tree classifier to predict the cluster label for each instance. We then use the classifier's decision tree as an instance partitioning model. Finally, we can also use instance-family information to build a decision tree classifier as mentioned above. Within all three possible initialization choices, the per-set algorithm configuration policy assigns the default configuration to each instance partition. Naturally, a combination of all three strategies is also possible. The number of individuals within the initial population is subject to hyperparameter optimization.

4.2 Population Fitness

To evaluate an individual's fitness, we use either of two types of metrics: population-based and default-runtime-based metrics. While population-based fitness metrics compare the individual's performance to the other individuals in the current iteration, default-runtime-based fitness metrics do not depend on the fitness of other individuals, i.e., it is possible to compare individuals' fitness values from different time steps. Note the time parameter t in Equation 4.1 indicates a fitness scoring that is dependent on the complete population at time t. For our purposes, a fitness metric is a function $\text{fitness}\colon \mathcal{P}^{(t)} \to \mathbb{R}^+$. Higher fitness values indicate a fitter individual. Fitter individuals are more likely to survive (cf. Section 4.3).

The fitness computation takes as input a function of runtimes per individual and instance, i.e., $\text{runtimes}\colon \mathcal{P}^{(t)} \times \mathcal{I} \to \mathbb{R}^+$. The runtimes have previously been computed or predicted in Line 3 of our algorithm. We use the runtimes function as a surrogate model for the runtime acquisition function r as it is costly to evaluate (cf. Section 4.5). Within our algorithm, we consider either of three possible fitness metrics including one population-based and two default-runtime-based ones. The population-based fitness metric is defined in Definition 4.1. Furthermore, Definition 4.2 and 4.3 present the default-runtime-based fitness metrics.

Definition 4.1 (Population-based Fitness Metric)**.** *Given a current evolution step t and populous $\mathcal{P}^{(t)}$, the population-based fitness metric* fitness_1 *compares individuals on a per-instance basis followed by computing the geometric mean, i.e.,*

$$\text{fitness}_1^{(t)}(p) := \left(\prod_{i \in \mathcal{I}} \frac{\text{median}\left\{ \text{runtimes}(q, i) \;\middle|\; q \in \mathcal{P}^{(t)} \right\}}{\text{runtimes}(p, i)} \right)^{\frac{1}{|\mathcal{I}|}} \quad . \tag{4.1}$$

Definition 4.2 (Default-Runtime-based Fitness Metric)**.** *Given a current evolution step t and populous $\mathcal{P}^{(t)}$, the default-runtime-based fitness metric* fitness_2 *compares the individual's PAR-2 scores to the default solver configuration, i.e.,*

$$\text{fitness}_2(p) := \frac{\sum_{i \in \mathcal{I}} \text{r}(\theta_0, i)}{\sum_{i \in \mathcal{I}} \text{runtimes}(p, i)} \quad . \tag{4.2}$$

Definition 4.3 (Default-Runtime-based Fitness Metric (Per-Instance))**.** *Given a current evolution step t and populous $\mathcal{P}^{(t)}$, the default-runtime-based fitness metric* fitness_3 *compares individuals on a per-instance basis to the default solver configuration followed by, again, computing the geometric mean, i.e.,*

$$\text{fitness}_3(p) := \left(\prod_{i \in \mathcal{I}} \frac{\text{r}(\theta_0, i)}{\text{runtimes}(p, i)} \right)^{\frac{1}{|\mathcal{I}|}} \quad . \tag{4.3}$$

Thereby, we denote the default configuration of the single best solver with θ_0. Refer to Section 5.2 for the definition of the single best solver.

While fitness_2 favors individuals that improve an individual's overall performance, the metrics fitness_1 and fitness_3 favor stronger improvements on single instances as we aggregate over each problem instance separately. Since we optimize our approach for PAR-2 runtime, it is expected that fitness_2 performs best as it is inversely proportional to an individual's PAR-2 performance sum.

4.3 Evolutionary Selection

Since there are not enough computational resources to evaluate all individuals that are produced by mutation or cross-over, an evolutionary selection is performed. Therein, we select individuals that we want to continue evaluating based on their fitness value and randomness. As introduced in Section 2.2, there are mainly two types of evolutionary selection: fitness-proportionate selection and tournament selection. The former is given in Definition 4.4, whereas we show the latter in Definition 4.5.

Definition 4.4 (Fitness-Proportionate Selection)**.** *Given an evolution step t, a fitness metric* $\text{fitness} : \mathcal{P}^{(t)} \to \mathbb{R}^+$*, and a sample size k, the probability of picking individual $p \in \mathcal{P}^{(t)}$ is then*

$$P(\textit{picking } p \textit{ at time } t) = \frac{\text{fitness}(p)}{\sum_{q \in \mathcal{P}^{(t)}} \text{fitness}(q)} \quad . \tag{4.4}$$

We pick k individuals without replacement. To do so, the normalizing denominator is adjusted after each draw by omitting already chosen individuals.

Definition 4.5 (Tournament Selection)**.** *Given an evolution step t, a fitness metric* $\text{fitness} : \mathcal{P}^{(t)} \to \mathbb{R}^+$*, and a disered population size of k, we randomly partition $\mathcal{P}^{(t)}$ into k tournament brackets. Then, we select the fittest individual among each bracket. Partitioning $\mathcal{P}^{(t)}$ can be done using random shuffling.*

Tournament selection uses the analogy of tournaments to select individuals. In practice, it is often favored because it has less stochastic noise [21]. The fittest individual will always be selected in tournament selection since there is no tournament bracket choice against which the fittest individual loses. Within the fitness-proportionate selection, however, the fittest individual might not be selected at all when unlucky. Less fit individuals may still survive tournament selection if paired with equally fit individuals as with fitness-proportionate selection. For our purposes, we implement, both, fitness-proportionate selection as well as tournament selection as defined above.

4.4 Mutation

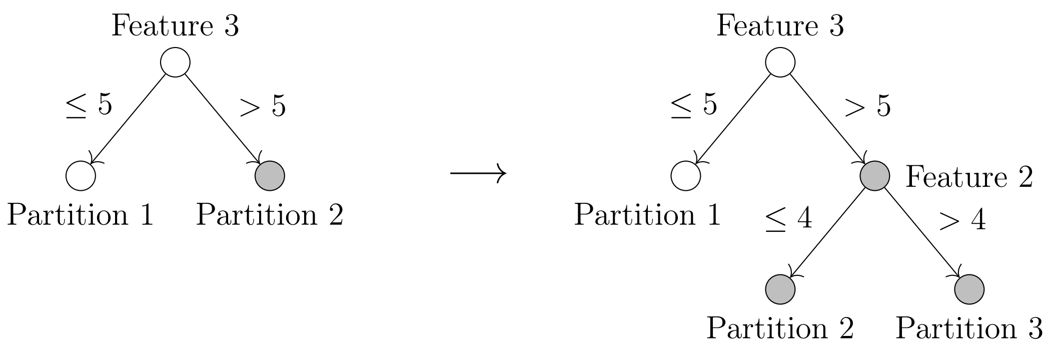
By mutating individuals, we explore the space of all possible individuals (cf. Algorithm 1, Line 7). In our case, we mutate the individuals' instance partitioning decision trees. Figure 3 shows this. To do so, there are two possibilities: We can either add a new split under a leaf (Figure 3a) or remove a split having at least one leaf (Figure 3b). Definition 4.6 shows when we choose to add and when we choose to remove a split.

Definition 4.6 (Add-Remove-Split Decision)**.** *Given the current number of partitions Λ of the instance partitioning tree of an individual and a desired number of partitions Λ^*, we use a simple step function to determine the probability of adding a split, i.e.,*

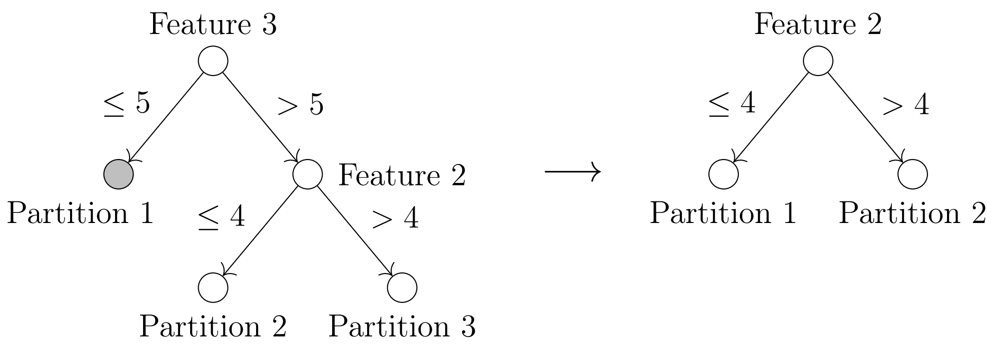
$$\text{add\_split}(\Lambda) := \begin{cases} 1 & \textit{if } \Lambda = 1 \\ 0 & \textit{if } \Lambda = \Lambda^* \\ \Lambda' & \textit{otherwise} \end{cases} \quad . \tag{4.5}$$

With $\Lambda' \in (0.5, 1.0)$, we denote the constant growth rate. The remove-split probability is then $\text{remove\_split}(\Lambda) := 1 - \text{add\_split}(\Lambda)$*. A value of, for example, $\Lambda' = 0.6$ means that it is expected that 6 out of 10 mutations add a split and 4 remove a split. This means that the tree grows by expected $6 - 4 = 2$ partitions every 10 mutations.*

We use a constant add-split probability greater than 50 % with a maximum number of splits. Once this number is reached, the constant add-split strategy returns a zero probability of adding another split. By doing so, we gradually grow the tree until we reach its final number of partitions. Slowly growing the tree makes sure that we have promising individuals for smaller tree sizes before growing. When reaching the desired number of partitions Λ^*, we continuously restructure the decision tree by removing a split, i.e., $\text{add\_split}(\Lambda^*) = 0$. If the tree has no splits, so only one partition, we always add a split, i.e., $\text{add\_split}(1) = 1$.



(a) Adding a decision tree splitter under the leaf with the partition identifier 2.



(b) Removing the decision tree splitter that produces partition 1.

Figure 3: Examples for mutating the instance partitioning decision tree of one particular individual.

4.4.1 Adding a Split

To add a new decision tree splitter, we follow a two-step approach. First, we compute a score $\zeta\colon \{1,\ldots,k\} \times \{1,\ldots,d\} \to \mathbb{R}_{\geq 0}$ for each combination of leaf and possible splitting feature of the current decision tree. If a certain splitting attribute $\alpha \in \{1,\ldots,d\}$ at a leaf $\lambda \in \{1,\ldots,k\}$ produces a trivial split (any of the created partitions is empty), then $\zeta(\lambda,\alpha) = 0$. A leaf-feature combination with a higher score indicates that adding a split there is more beneficial. Second, we choose a leaf λ and a split attribute α as given in Definition 4.7.

Definition 4.7 (Choosing Add-Split Mutations)**.** *Given a scoring function $\zeta\colon \{1,\ldots,k\}\times\{1,\ldots,d\} \to \mathbb{R}_{\geq 0}$, we randomly choose a leaf λ and an attribute α with a probability of*

$$P(\textit{ choosing leaf } \lambda \textit{ and splitting atribute } \alpha\,) = \frac{\zeta(\lambda,\alpha)}{\sum_{m=1}^{k}\sum_{b=1}^{d}\zeta(m,b)} \quad . \tag{4.6}$$

We may also consider choosing λ and α with the highest score deterministically depending on a hyperparameter of our evolutionary algorithm. Adding the splitter with feature α under leaf λ is done as shown in Figure 3a.

To be more specific, we make use of three possible instantiations of ζ: (1) Uniformly random, (2) leaf sizes, and (3) split entropy with leaf size scaling. To uniformly sample possible leaves and split attributes, we make use of Definition 4.8. By randomly adding splits, we explore the space of possible instance partitioning decision trees.

Definition 4.8 (Add-Split Mutation – Uniformly Random)**.** *Given possible valid choices of $\lambda \in \{1, \dots, k\}$ and $\alpha \in \{1, \dots, d\}$, we set $\zeta_1(\lambda, \alpha) := 1$. This indicates uniformly random selection among all possible choices (cf. Definition 4.7).*

Using leaf sizes is shown in Definition 4.9. By splitting leaves with more instances with a higher probability, we make sure that all partitions are sufficiently big. Bigger partitions are more likely to generalize to unseen instances. Also, Section 6.2 shows that optimal decision trees for instance partitioning have, in general, relatively big leaves.

Definition 4.9 (Add-Split Mutation – Leaf Sizes)**.** *Given valid choices of $\lambda \in \{1, \dots, k\}$ and $\alpha \in \{1, \dots, d\}$, we define leaf-size scoring as $\zeta_2^{(p)}(\lambda, \alpha) := |\mathcal{I}^{(p,\lambda)}|$. The index $p \in \mathcal{P}^{(t)}$ indicates the current individual that we are looking at. Furthermore, $\mathcal{I}^{(p,\lambda)}$ denotes all instances that are part of the instance partition represented by leaf λ within the individual p. Note that α is not used on the right-hand side of the definition, indicating a random selection of splitting attributes.*

Finally, Definition 4.10 shows how to use a variation of the split entropy criterion [28] for our purposes. It takes the leaf sizes from before into account as well as prefers split attributes that divide the partition evenly. Section 6.2 shows that the optimal decision trees for instance partitioning split instances quite evenly within each split decision.

Definition 4.10 (Add-Split Mutation – Split Entropy with Leaf-Size Scaling)**.** *Again, given valid choices of $\lambda \in \{1, \dots, k\}$ and $\alpha \in \{1, \dots, d\}$, we define split entropy with leaf-size scaling as*

$$\begin{aligned} \zeta_3^{(p)}(\lambda, \alpha) \;:=\; & -|\iota^{(p)}(\lambda, \alpha)| \log_2\left(\frac{|\iota^{(p)}(\lambda, \alpha)|}{|\mathcal{I}^{(p,\lambda)}|}\right) \\ & - \left(|\mathcal{I}^{(p,\lambda)}| - |\iota^{(p)}(\lambda, \alpha)|\right) \log_2\left(\frac{|\mathcal{I}^{(p,\lambda)}| - |\iota^{(p)}(\lambda, \alpha)|}{|\mathcal{I}^{(p,\lambda)}|}\right) \end{aligned} \tag{4.7}$$

$$\text{with } \iota^{(p)}(\lambda, \alpha) := \left\{ e \in \mathcal{I}^{(p,\lambda)} \mid \operatorname{feat}(e)_\alpha \le \beta_{\lambda,\alpha} \right\} .$$

Thereby, $\iota^{(p)}(\lambda, \alpha)$ denotes all instances that would end up in the left subpartition if adding a decision tree splitter under leaf λ for the feature α within individual p. The value $\beta_{\lambda,\alpha}$ denotes the split threshold that we choose for each split attribute α within each leaf λ deterministically. The definition is equivalent to the information entropy regarding the relative sizes of the newly created partitions multiplied by $|\mathcal{I}^{(p,\lambda)}|$.

To determine the splitting threshold $\beta_{\lambda,\alpha}$ for all possible criteria, we use hierarchical clustering of the values of feature α, i.e., $\operatorname{feat}(e)_\alpha$, constrained to the instances $e \in \mathcal{I}^{(p,\lambda)}$. We experiment with different linkage types within the hierarchical clustering (cf. Section 5.6). Initially, each feature value starts in a cluster of its own. Then, we gradually merge feature value clusters until only two clusters remain. We set $\beta_{\lambda,\alpha}$ to the average of the maximum of the cluster with the smaller values and the minimum of the cluster with the bigger values.

We choose $\beta_{\lambda,\alpha}$ deterministically for mainly three reasons. First, we hope to create more robust partitionings by grouping instances with similar instance features together. Second, there is no ground truth as, for example, in decision trees for supervised learning tasks, i.e., we can only calculate the split entropy regarding the partition sizes. Finally, looking at all possible split positions is computationally costly as the runtime evaluation (cf. Section 4.5) would have to be done many times more. This is not feasible as the runtime evaluation is the bottleneck regarding the strategy's overall runtime cost.

4.4.2 Removing a Split

Similarly to adding a split, we follow a two-step approach to remove a split from the decision tree. For simplicity, we only remove splits with at least one leaf as no tree restructuring is needed. Removing a split is simpler compared to adding a split as there is no choice of the feature and threshold to split involved. First, we again compute a score $\xi\colon \{1,\dots,k\} \to \mathbb{R}_{\geq 0}$ for each possible leaf $\lambda \in \{1,\dots,k\}$. A leaf with a higher score indicates that removing the split above it is more beneficial. Second, we randomly choose to remove a split above a certain leaf as given in Definition 4.11.

Definition 4.11 (Choosing Remove-Split Mutations)**.** *Given a score $\xi : \{1,\dots,k\} \to \mathbb{R}_{\geq 0}$, we randomly choose a leaf λ with a probability of*

$$P(\textit{ choosing leaf } \lambda\,) = \frac{\xi(\lambda)}{\sum_{m=1}^{k} \xi(m)} \quad . \tag{4.8}$$

We may also consider choosing the leaf λ with the highest score deterministically depending on a hyperparameter of our evolutionary algorithm. Removing the splitter above leaf λ is done as shown in Figure 3b.

Moreover, we consider three possible instantiations of ξ: (1) Uniformly random, (2) inverse leaf sizes, and (3) inverse split entropy. To choose leaves, above which to remove splits, uniformly random, we make use of Definition 4.12. By randomly removing splits, we explore the space of possible instance partitioning decision trees.

Definition 4.12 (Remove-Split Mutation – Uniformly Random)**.** *Given possible valid choices of $\lambda \in \{1,\dots,k\}$, we set $\xi_1(\lambda) := 1$. This indicates uniformly random selection among all possible choices (cf. Definition 4.11).*

Definition 4.13 introduces the inverse-leaf-size scoring. It favors the removal of decision tree splits above small leaves as splits with few instances do not generalize well on unseen instances. Section 6.2 shows that optimal decision trees for instance partitioning have rather big leaves, in general.

Definition 4.13 (Remove-Split Mutation – Inverse Leaf Sizes)**.** *Given possible valid choices of $\lambda \in \{1,\dots,k\}$ within an individual $p \in \mathcal{P}^{(t)}$, we define inverse-leaf-size scoring as*

$$\xi_2^{(p)}(\lambda) := \frac{1}{|\mathcal{I}^{(p,\lambda)}|} \quad . \tag{4.9}$$

Finally, the inverse-split-entropy scoring is introduced in Definition 4.14. The intuition behind the inverse split entropy is similar to the inverse leaf size scoring function: It favors the removal of unbalanced decision tree splits. Section 6.2, again, shows that optimal decision trees for instance partitioning split instances rather evenly, in general.

Definition 4.14 (Remove-Split Mutation – Inverse Split Entropy)**.** *Given possible valid choices of $\lambda \in \{1,\dots,k\}$ within an individual $p \in \mathcal{P}^{(t)}$, we define inverse-split-entropy scoring as*

$$\xi_3^{(p)}(\lambda) := \frac{1}{\zeta_3^{(p)}(\lambda, \alpha_\lambda)} \quad , \tag{4.10}$$

using the definition of ζ_3 (see Definition 4.10). Thereby, α_λ denotes the splitting feature in the decision tree splitter above leaf λ.

4.5 Tuning SAT Solvers on Instance Partitions

To tune SAT solvers on the instance partitions of the individuals, we use several techniques that we have previously published [32]. These include runtime discretization and prediction techniques.

Algorithm 2: Tuning Configuration Policies within Individuals

```
Input:  The current population $\mathcal{P}^{(t)}$, configurations $\Theta$, runtime acquisition
        function $\mathrm{r}\colon \Theta \times \mathcal{I}$, and instance features $\mathrm{feat}\colon \mathcal{I} \to \mathbb{R}^d$
Output: Runtimes per individual and instance, i.e., $\mathrm{runtimes}\colon \mathcal{P}^{(t)} \times \mathcal{I} \to \mathbb{R}^+$
1  $\mathrm{runtimes}(p, e) \leftarrow \perp$ for all $p \in \mathcal{P}^{(t)}$ and $e \in \mathcal{I}$
2  for $p \in \mathcal{P}^{(t)}$ do
3      for $\lambda \in \{1, \ldots, k\}$ do
           // Evaluate random configurations
4          $\Theta' \leftarrow$ Uniformly randomly sample $c$ configurations from $\Theta \setminus \Theta_0$
5          for $\theta \in \Theta'$ do
6              Acquire at least $R$ runtimes $\mathrm{r}(\theta, e)$ with $e \in \mathcal{I}^{(p,\lambda)}$    // Uses caching
7              Fit model to predict the cluster labels $\gamma_m(\theta, e)$ of all $e \in \mathcal{I}^{(p,\lambda)}$
           // Update the currently best configuration
8          $\psi^{(p)}(\lambda) \leftarrow \operatorname*{arg\,min}_{\theta \in \Theta' \cup \Theta_0} \sum_{e \in \mathcal{I}^{(p,\lambda)}} \kappa(\gamma_m(\theta, e), e)$
           // Assign each instance to the currently best runtime
9          for $e \in \mathcal{I}^{(p,\lambda)}$ do
10             $\mathrm{runtimes}(p, e) \leftarrow \gamma_m\left(\psi^{(p)}\left(\pi^{(p)}(e)\right), e\right)$
11 return runtimes
```

Algorithm 2 shows the tuning procedure as used in Algorithm 1, Line 3. Thereby, we tune each instance partition $\lambda \in \{1, \ldots, k\}$ of each individual $p \in \mathcal{P}^{(t)}$ separately by finding better configurations $\psi^{(p)}(\lambda)$ for partition λ (cf. Definition 1.1). This is useful in practice as the evaluation can be parallelized over all individuals and partitions. With $\mathcal{I}^{(p,\lambda)}$, we again denote the instances that are in partition $\lambda \in \{1, \ldots, k\}$ within individual $p \in \mathcal{P}^{(t)}$.

We randomly sample c configurations Θ' from $\Theta \setminus \Theta_0$ that we want to test within a partition λ of a given individual p (Algorithm 2, Line 4). We sample configurations uniformly random and independently regarding each partition. With Θ_0, we denote configurations with known runtimes for all instances in $\mathcal{I}$, e.g., the used solver's default configurations.

Thereafter, we evaluate each candidate configuration within Θ' by acquiring R runtimes $\mathrm{r}(\theta, e)$ with $e \in \mathcal{I}^{(p,\lambda)}$ (Line 6), or less if $|\mathcal{I}^{(p,\lambda)}| < R$. We re-use runtime labels that we have previously acquired or predicted within a cache. To be more specific, we use all previously acquired or predicted runtimes of the partition's instances $\mathcal{I}^{(p,\lambda)}$ and uniformly randomly sample additional runtimes if we have less than R observations in total. If there are already at least R data points for a given partition, nothing has to be done.

Then, we use a supervised machine-learning model to predict the runtime labels of the candidate configurations on the remaining instances within the respective partition (Line 7). The following paragraphs show how the runtime transformation and prediction work. In

Line 8, we update the per-set algorithm configuration policy ψ. We set it to the configuration with the minimal predicted runtime sum. We compute the minimum regarding all candidate configurations Θ and configurations Θ_0, for which we know all runtimes, e.g., the portfolio's default solver configurations. In Lines 9 and 10, we collect all results within the runtimes function. It is used to determine the fitness of an individual (cf. Section 4.2 and Algorithm 1). To compute the fitness values (cf. Section 4.2), it is obligatory to predict the runtime labels of all instances since different individuals use different instance partitionings.

4.5.1 Runtime Transformation

We assume that the runtimes of a subset of configurations $\Theta_0 \subseteq \Theta$ are known. We then assign each of those runtimes $\{\mathrm{r}(\theta', e) \mid \theta' \in \Theta_0, e \in \mathcal{I}\}$ one out of m clusters $Cl_1, \ldots, Cl_m$. Thereby, the fastest runtimes per instance end up in the first cluster Cl_1 and the slowest runtimes per instance in cluster Cl_{m-1}. Timeouts have a separate cluster: Cl_m. This runtime transformation $\gamma_m : \Theta_0 \times \mathcal{I} \rightarrow \{1, \ldots, m\}$ can be defined as

$$\gamma_m(\theta', e) = j \Leftrightarrow \mathrm{r}(\theta', e) \in Cl_j \ , \tag{4.11}$$

for all configurations $\theta' \in \Theta_0$ and instances $e \in \mathcal{I}$. Moreover, we extend γ_m to $\Theta \times \mathcal{I}$ by assigning each configuration-instance tuple (θ, e) with associated runtime $\mathrm{r}(\theta, e)$ to the cluster $\gamma_m\left(\hat{\theta}, e\right)$ of the closest log-runtime

$$\hat{\theta} = \underset{\theta' \in \Theta_0}{\arg\min} \ |\log \mathrm{r}(\theta', e) - \log \mathrm{r}(\theta, e)| \ . \tag{4.12}$$

4.5.2 Runtime Prediction

In preliminary experiments, we achieved higher accuracies for predicting discrete runtime labels rather than raw runtimes. Furthermore, research in SAT-solver portfolio selectors supports this claim [25, 59]. Given a new configuration $\theta \in \Theta \setminus \Theta_0$ and runtimes $\{\mathrm{r}(\theta, e') \mid e' \in \mathcal{I}'\}$ for a subset of instances $\mathcal{I}' \subseteq \mathcal{I}$, we want to predict the performance of θ for all remaining instances $\mathcal{I} \setminus \mathcal{I}'$ without acquiring any further runtimes. To do so, we fit a supervised machine-learning model, e.g., a standard random-forest model. Its inputs are instance features $\mathrm{feat}(e')$ and the clusters of known configurations $\gamma_m(\theta', e')$ for all $\theta' \in \Theta_0$ and $e' \in \mathcal{I}'$. Moreover, the target labels of the model are the clusters of the new configuration $\gamma_m(\theta, e')$ for all instances $e' \in \mathcal{I}'$. We then use the fitted model to make predictions about $\gamma_m(\theta, e)$ for $e \in \mathcal{I} \setminus \mathcal{I}'$. In the end, we have discrete cluster labels $\gamma_m(\theta, e)$ for the new configuration θ and all instances $e \in \mathcal{I}$.

Since each instance has a different weight regarding the resulting mean PAR-2 performance of a solver configuration, we transform the discrete runtime labels $\gamma_m(\theta, e)$ back using $\kappa : \{1, \ldots, m\} \times \mathcal{I} \rightarrow \mathbb{R}^+$. Easy instances have less impact on the overall performance of a configuration. The transformation back can be defined as

$$\kappa(j,\, e) := \frac{1}{|\{\theta \in \Theta_0 \mid \gamma_m(\theta, e) = j\}|} \sum_{\theta \in \Theta_0,\, \gamma_m(\theta, e) = j} \mathrm{r}(\theta, e) \ , \tag{4.13}$$

for a discrete cluster label $j \in \{1, \ldots, m\}$ and instance $e \in \mathcal{I}$. Less formally speaking, we map a cluster label $\gamma_m(\theta, e)$ to the mean runtime of known configurations whose runtimes are within the respective runtime cluster regarding a single instance e.

4.5.3 Runtime Capping

To further reduce the cost of acquiring runtimes, we make use of an adaptive runtime capping strategy as already mentioned in related work [42, 52]. Using PAR-2 runtime scoring, experiments are conducted with a fixed timeout of $\tau = 5000\,\mathrm{s}$ [31]. Timeouts are penalized with a runtime of 2τ. In contrast to that, we adaptively determine the timeout for a given experiment as stated in Definition 4.15. If a solver exceeds this threshold, we also use PAR-2 scoring, i.e., $2\tau_a(e)$. If $2\tau_a(e) \geq \tau$, then we use a penalty runtime of 2τ to be consistent.

Definition 4.15 (Adaptive Timeout)**.** *Given an instance $e \in \mathcal{I}$ and solver configurations $\Theta_0 \subseteq \Theta$ for which all runtimes $\{\mathrm{r}(\theta, e) \mid \theta \in \Theta_0\}$ are known, we define the adaptive timeout value $\tau_a : \mathcal{I} \to \mathbb{R}^+$ as follows:*

$$\tau_a(e) := \begin{cases} t_a & \text{if } \tau'_a(e) \leq t_a \\ \tau & \text{if } \tau'_a(e) \geq \tau \\ \tau'_a(e) & \text{otherwise} \end{cases} \qquad \text{with } \tau'_a(e) := T_a \cdot \min\{\mathrm{r}(\theta, e) \mid \theta \in \Theta_0\} \quad . \tag{4.14}$$

Thereby, $t_a \geq 0$ denotes the minimum and $\tau > t_a$ the maximum timeout value possible. With $T_a \geq 1$, we control how much worse than the best-known runtime the timeout value can be.

5 Experimental Design

This section presents all parts of our experimental design and all done experiments. First, we briefly explain the used data set and problem instance features. Second, we list baselines to which we want to compare our algorithm. Third, we describe competing algorithms. Fourth, we present our experimental methodology. Fifth, we shortly describe the fixed portfolio of SAT solvers that we use as well as the configuration options that we optimize. Then, we look at the hyperparameter choices within our evolutionary algorithm. And finally, we show implementation details as well as how we run the SAT solvers.

5.1 Data

We use the SAT Competition 2022 Anniversary Track dataset [13] for all our experiments. Additionally, we also use a database of 56 instance features[1] from the Global Benchmark Database (GBD) by Iser et al. [47]. They comprise instance size features and node distribution statistics for several graph representations of SAT instances, among others, and are primarily inspired by the SATzilla 2012 features described in [70]. Further, the SATzilla 2012 features are originally inspired by Nudelman et al. [60]. All features are numeric and free of missing values. We drop 10 out of 56 features because of zero variance on the given instances. Also, we drop the feature computation time, which is one of the features, as it is hardware-specific. In total, we use 45 features that we normalize using a log-normal transformation $v \mapsto (\log v - \mathrm{mean}(\log v)) \,/\, \mathrm{std}(\log v)$ for feature values v.

Moreover, we make use of instance-family information[2] for the stratified sampling of the complete dataset. Instance families comprise instances from the same application domain, e.g., planning, cryptography, etc., and are a valuable tool for analyzing solver performance.

All experiments use a random subsample of 1000 out of 5355 instances. It is selected using stratification regarding the instances' family. All instance families that entail less than 1 % of instances are put into one meta-family for stratification. This smaller dataset allows for more extensive experimentation.

5.2 Baselines

This section lists all baseline approaches to which we want to compare our algorithm. We evaluate the performance of each baseline as discussed in Section 5.4.

5.2.1 Single Best Solver (SBS).

The single best solver is simply the solver with the overall best PAR-2 performance out of the available solvers: Kissat [33, 19] and SeqFrost [61]. Thereby, SAT-solver configurations are fixed to the default one. Optimally, every approach that follows has to outperform this baseline. Otherwise, there will not be much use to a portfolio approach if the single best solver is already better. In our particular case, the Kissat solver takes the role of the single best solver (cf. Section 5.5). As the SBS is known beforehand, no additional tuning is required. We note, however, that finding the default configuration supposedly also precedes heavy experimentation and cannot be taken for granted.

[1] https://benchmark-database.de/getdatabase/base_db
[2] https://benchmark-database.de/getdatabase/meta_db

5.2.2 Single Best Configuration (SBC).

The single best configuration is the one single SAT-solver configuration with the best performance on the given benchmark out of the available solvers: Kissat [33, 19] and SeqFrost [61]. In our presented per-set algorithm configuration context, this means that π is constant, i.e., each instance is assigned the same instance partition. In our case, the single best configuration is a non-default configuration of the Kissat solver. Our approach ought to outperform this baseline. Note that the SBC solver is not *free* whatsoever as it is not known beforehand, i.e., we have to perform tuning to find it. In Section 6.4, we use a fixed-size random sampling strategy to determine the SBC solver for comparison purposes. Additionally, we also benchmark an established tuning framework, i.e., Optuna [5].

5.2.3 Virtually Best Solver with Default Configurations (VBS).

The virtually best solver is a common comparison for portfolio selection approaches. It uses an oracle that selects the best solver for each instance. The two available solvers over which we calculate the VBS are the Kissat [33, 19] and SeqFrost solvers [61]. As there is no such oracle in reality, it is only used for comparison purposes. For this baseline, we also keep the solvers' configurations fixed to the default ones. Because we may tune SAT solvers, we can outperform the virtually best solver of the default configurations.

5.2.4 Virtually Best Configuration (VBC).

Similar to the virtually best solver, we also want to report the performance of an oracle picking the solver with the best configuration for each instance. This requires a full grid search on all instances of the dataset. The VBC is an upper bound for *all* other algorithm selectors using the same set of possible configurations.

5.3 Competitors

This section describes all competing algorithms. We evaluate their performance as discussed in Section 5.4.

5.3.1 Upper-bound of ISAC approaches (ISAC).

As listed in Table 1, the only feasible competitors are ISAC-based approaches. To compare ourselves to any possible ISAC approach, we first cluster instance space with the clustering approach described in the original ISAC paper [48]. Then, we select the single best configuration for each of those clusters. An approach that outperforms this upper bound also outperforms any ISAC approach, that uses the same clustering technique transitively, no matter which algorithm configuration system is used to tune the SAT solvers.

5.3.2 Upper-bound of Family prediction (Family).

Another competing idea is to simply tune each sufficiently big instance family. Thereby, too small families are grouped into one meta-family. We select the single best configuration for each of those family clusters. Also, preliminary experiments showed that a standard random-forest classifier can predict those instance families reasonably well ($> 99\,\%$ Matthew's correlation coefficient performance). An approach that outperforms this upper bound also outperforms any approach, that uses the same family grouping transitively, no matter which algorithm configuration system is used to tune the SAT solvers.

5.3.3 Evolutionary Trees.

Naturally, we also want to obtain the performance metrics for our approach. Optimally, our approach lives in between the upper bounds of ISAC approaches and family prediction (on the lower end) and the virtually best configuration (on the upper end).

5.4 Evaluation Framework

We perform a 5-times repeated 5-fold cross-validation with stratification regarding instance families. Each of the repetitions uses a different random seed, i.e., 0, 1, 2, 3, and 4. Within each of the five cross-validation folds, we use 80 %, i.e., 4 of the 5 dataset folds, to train each of the aforementioned approaches, i.e., to cluster instances within the ISAC approach and to evolve the instance partitioning within our evolutionary algorithm (ψ and π). Thereafter, we use the remaining 20 %, i.e., 1 of the 5 dataset folds, to evaluate performance. Therein, ψ and π are fixed. We tune hyperparameters regarding the evaluation datasets. Also, we want to report the differences between determining the baselines on the complete dataset versus only the training dataset. This is done for each of the five folds and all aforementioned competitors and baselines. Performance metrics are averaged over all folds and all repetitions. We report common statistics on the result distribution of five folds times five repetitions.

We evaluate performance metrics regarding two categories: First, we compare the SAT-solver PAR-2 scores for all approaches with a timeout of $\tau = 5000\,\mathrm{s}$ [31]. Since each of the data folds has a slightly different runtime, we scale each observed PAR-2 runtime r_d with the VBC baseline runtime r_{vbc} of that dataset fold, i.e., $perf := r_{vbc}/r_d$. Since the VBC PAR-2 runtime is an upper-bound for all approaches, $r_{vbc}/r_d \in (0, 1]$. A value of 0.8, for example, denotes that we have achieved 80 % of the VBC PAR-2 performance. A perfect score of 1.0 indicates that we have reached the VBC performance.

Also, we show *runtime cumulative-distribution* plots for each approach using PAR-2 runtimes as well as those scaled runtimes. A runtime cumulative distribution function plot reports the fraction of instances that is solved with at most a given PAR-2 runtime, i.e.,

$$\mathrm{par2\_rcdf}_{\psi,\pi}\colon \mathbb{R} \to [0,1]\,,\ \mathrm{par2\_rcdf}_{\psi,\pi}(t) := \frac{|\,\{e \in \mathcal{I} \mid \mathrm{r}(\psi(\pi(e))\,, e) \leq t\}\,|}{|\mathcal{I}|}\ . \tag{5.1}$$

Thereby, $\mathrm{par2\_rcdf}_{\psi,\pi}(\tau)$ denotes the fraction of instances that is solved with less than the timeout value of τ, for example. Similarly, we can also show runtime cumulative-distribution plots regarding the VBC PAR-2 performance of each instance: It reports the fraction of instances that is solved with at most a given multiple of the VBC PAR-2 runtime, i.e.,

$$\mathrm{vbc\_rcdf}_{\psi,\pi}\colon \mathbb{R} \to [0,1]\,,\ \mathrm{vbc\_rcdf}_{\psi,\pi}(t) := \frac{|\,\{e \in \mathcal{I} \mid \mathrm{r}(\psi(\pi(e))\,, e) \leq t \cdot \mathrm{r}_{vbc}(e)\}\,|}{|\mathcal{I}|}\ . \tag{5.2}$$

Table 2: The optimal portfolios (VBS) consisting of two members regarding the SAT Competition 2022 Anniversary Track results [13] with default configurations.

Portfolio members		Mean PAR-2
`Kissat_MAB_ESA`	`kissat-sc2022-bulky`	2552.71
`Kissat_MAB_ESA`	`kissat-els-v2`	2557.22
`Kissat_MAB_ESA`	`kissat-sc2022-light`	2557.75
`Kissat_MAB-HyWalk`	`kissat-els-v2`	2559.40
`Kissat_MAB-HyWalk`	`kissat-sc2022-bulky`	2559.81
`SeqFROST-NoExtend`	`kissat-sc2022-bulky`	2563.08

Thereby, $\mathrm{r}_{vbc}(e)$ denotes the VBC PAR-2 performance for each problem instance $e \in \mathcal{I}$.

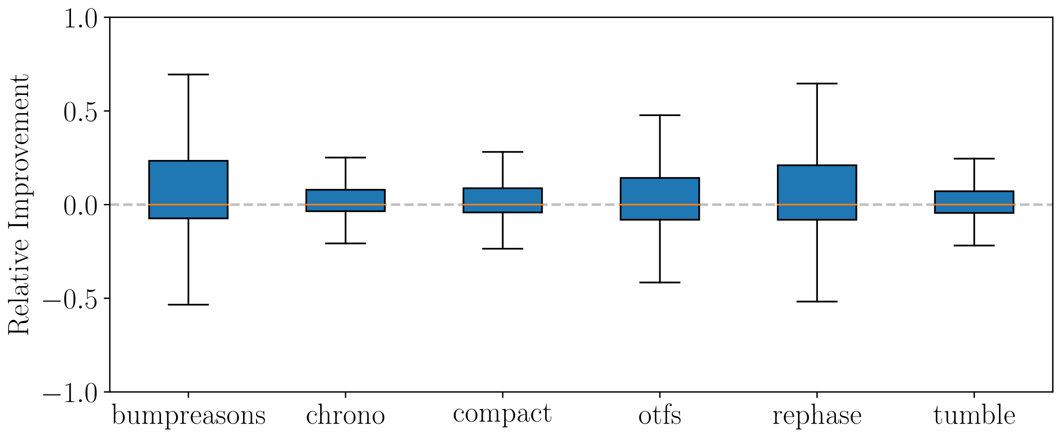
Second, we compare the cost $cost_{tuning}$ of the tuning phases. Within our evolutionary algorithm, this includes all iterations and solver experiments. To avoid time-expensive searches and runtime acquisitions, we use several techniques as described in Section 4.5. For better comparability, we express the total tuning cost $cost_{tuning}$ in multiples of the time it would take to evaluate the default solver on all instances, i.e., $cost_{tuning} / \sum_{e \in \mathcal{I}} \mathrm{cost}_{sbs}(e)$. An example of this is that an approach might need x times the tuning cost in comparison to evaluating the default solver on all instances cost_{sbs}. Note that the evaluation time of a solver is smaller than or equal to the respective PAR-2 score as we do not penalize runtimes, i.e., $\mathrm{cost}_{sbs}(e) \leq \mathrm{r}_{sbs}(e)$ for all instances e.

5.5 SAT Solvers and their Configuration Options

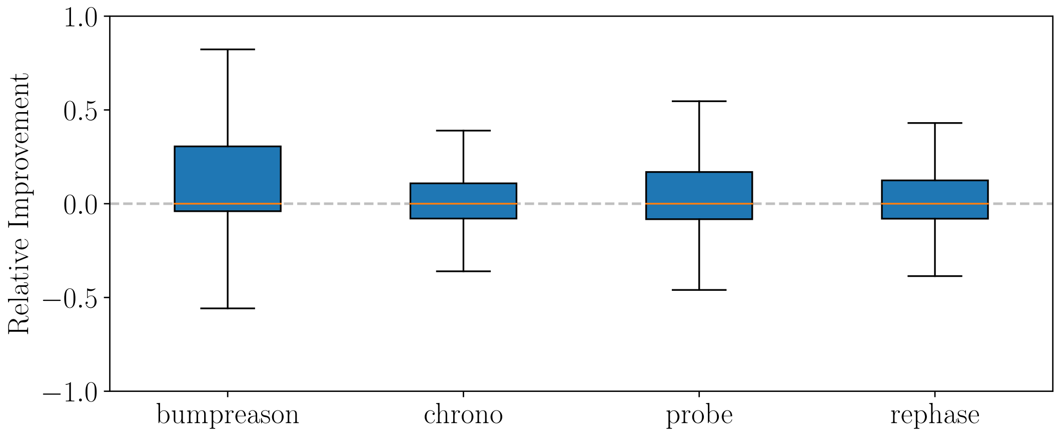
Because of time constraints in the acquisition and analysis of solver runtimes, we limit ourselves to two available SAT solvers: Kissat [33, 19] and SeqFrost [61]. We have picked those two solvers based on an analysis of the SAT Competition 2022 Anniversary Track results [13]. Table 2 shows the six best portfolios (consisting of two solvers) regarding the virtually best solver baseline using the aforementioned dataset (cf. Section 6.1). The five best portfolios solemnly consist of different versions of the Kissat solver. The first non-Kissat solver involved is the SeqFrost solver forming the sixth best portfolio together with a version of the Kissat solver. For this reason, we have picked those two for all further experiments and analysis. Note that the mean PAR-2 scores are quite close for all listed portfolios.

Both solvers are conflict-drive clause-learning solvers (CDCL). CDCL solving is derived from the Davis-Putnam-Logemann-Loveland algorithm [26], often referred to as DPLL. Marques-Silva et al. [55] give an overview of recent advances in the field of CDCL SAT solving. Apart from DPLL techniques, CDCL solvers employ various kinds of improvements including learning policies [65], branching heuristics [57] and clause deletion policies [35].

For the configuration options that we want to tune, we only pick binary simplification options. Simplifications are heuristics within a SAT solver that show practical performance benefits on some problem instances. However, they may provide no performance advantage or may even produce worse runtimes on others. Preliminary experiments have shown that such configuration options have a relatively big impact on the SAT solver



(a) Kissat.



(b) SeqFrost.

Figure 4: Relative improvement regarding the PAR-2 scores of configuration options turned on versus off for the chosen configuration options. Results are based on the dataset described in Section 5.1. We aggregate the relative improvements over all instances.

performance while keeping the configuration space rather small and discrete. In comparison, real-valued configuration options would need further modeling, e.g., Bayesian optimization techniques [52].

Figure 4 shows all configuration options that we want to tune within our algorithm. Within the Kissat solver (cf. Figure 4a), we tune the simplification options `bumpreasons`, `chrono`, `compact`, `otfs`, `rephase`, and `tumble`. Within the SeqFrost solver (cf. Figure 4b), we tune the simplification options `bumpreason`, `chrono`, `probe`, and `rephase`. Refer to the documentation of Kissat[3] and SeqFrost[4] respectively for more information. All displayed configuration options are turned on by default. To obtain the boxplots shown in Figure 4, we perform a complete grid search of all parameter choices using our 1000-instances dataset as mentioned in Section 5.1. Then, we produce boxplots of the relative improvements regarding the PAR-2 runtimes of experiments where a particular option is turned on versus experiments where this option is turned off. The `bumpreasons`

[3]https://github.com/arminbiere/kissat
[4]https://github.com/muhos/SeqFROST

configuration option of the Kissat solver, for example, provides no performance benefit on average. There are, however, several problem instances where configurations with the `bumpreasons` option turned on perform 50 % better. On the other hand, there are also several problem instances where configurations with the `bumpreasons` option turned on perform 50 % worse. This behavior of configuration options is desirable as choosing parameter values on a per-instance or per-set basis can provide a huge performance advantage.

5.6 Evolutionary Trees Hyperparameters

For convenience, this section gives an overview of all hyperparameters of our evolutionary algorithm using decision trees.

Initial Population. To create the initial population (see Section 4.1), we use three different strategies: (1) We either start from an empty tree, (2) start with a decision tree predicting the clustering of instance feature space using k-means clustering with k being the number of desired partitions, or (3) start with a decision tree predicting the instances' families. In all cases, we only consider a single individual within the initial population.

Population Fitness. To evaluate an individual's fitness (see Section 4.2), we use three metrics: (1) We either use a population-based fitness score that compares individuals on a per-instance basis followed by computing the geometric mean, (2) a default-runtime-based fitness score comparing an individual's PAR-2 score to the default solver configuration, or (3) a default-runtime-based fitness score that compares individuals on a per-instance basis to the default solver configuration followed by computing the geometric mean.

Evolutionary Selection. To select individuals for mutations (see Section 4.3), we use two approaches: (1) We either use tournament selection or (2) fitness-proportionate selection.

Add-Remove-Split Decision. To determine whether to add or remove a decision tree split (see Section 4.4), we make use of a simple step function. Possible values for Λ^* are 2, 3, 4, and 5 as preliminary experiments have shown that smaller trees generalize better to unseen instances. The growth rate Λ' is set to 0.6 to accomplish a slow and steady growth of the partitioning tree.

Add-Split Mutation. To find the position where to add a split (see Section 4.4.1), we use three criteria: (1) Uniformly random, (2) leaf sizes, and (3) split entropy with leaf size scaling.

Split Threshold. To find the split threshold for a given leaf (see Section 4.4.1), we use hierarchical clustering with one of four linkage types: (1) ward linkage, (2) single linkage, (3) complete linkage, and (4) average linkage.

Remove-Split Mutation. To find the position where to remove a split (see Section 4.4.2), we use three criteria: (1) Uniformly random, (2) inverse leaf sizes, and (3) inverse split entropy.

Partially Deterministic Mutations. If enabled, add-split and remove-split positions are determined by the maximum score with random tie-breaking (see Section 4.4.1).

Size of Configuration Sample. Within Algorithm 2 in Line 4, we sample $c = 5$ configurations. Thereby, we always sample four Kissat configurations and one SeqFrost configuration as optimal portfolios use both solvers (cf. Table 3).

Minimum Runtime Observations. Within Algorithm 2 in Line 6, we want to acquire at least R runtimes for a given configuration. We use the values 5, 10, 15, and 20.

Runtime Transformation. To discretize runtimes, we assign each value to one out of m clusters. We only consider $m = 3$ as previously published results show that this is beneficial [32].

Runtime Capping. We instantiate the parameters in Definition 4.15 with $t_a = 10\,\mathrm{s}$, $\tau = 5000\,\mathrm{s}$, and $T_a \in \{1.0, 1.1, 1.5\}$.

5.7 Implementation

Our implementation uses Python with *scikit-learn* [62] for making predictions and *gbd-tools* [47] for SAT-instance and feature retrieval. We use fixed random seeds for reliable reproduction. In Section 6.4, we also benchmark the performance of an established tuning framework, i.e., Optuna [5].

5.8 Experimental Setup

We have run all experiments on a server with a 32-core AMD EPYC 7551 processor and 128 GiB of main memory. All obtained solver runtimes have been measured using the *CPU time* of the solver process. We use 5000 s as a timeout for all solver runs, following the annual SAT competitions [31]. Also, we limit the memory consumption to 16 GiB per solver run. The memory constraint has never been surpassed.

6 Evaluation

Within this section, we evaluate all our approaches and baselines as presented in Section 5. But first, we take a look at optimal portfolios of size k consisting of possible configurations in Section 6.1. Second, we show optimal decision trees for partitioning instances in Section 6.2 (cf. Section 4.1). In Section 6.3, we show PAR-2 performances of all approaches and baselines using a configuration oracle. Given an instance partition, this oracle provides the best possible configuration. By doing so, it is possible to compare the partitioning strategies on their own. Thereby, we also look at the impact of possible hyperparameter choices in Section 6.3.1. Finally, Sections 6.4 and 6.5 compare the strategies' tuning costs and the overall PAR-2 performances of all approaches and baselines.

6.1 Optimal Configuration Portfolios

Having obtained full grid-search results for our dataset (see Section 5.1), we are naturally interested in the optimal portfolios of a fixed size k regarding all possible configurations Θ. A fixed-size portfolio consists of a subset of configurations $\tilde{\Theta} \subseteq \Theta$ with $|\tilde{\Theta}| = k$. The optimal portfolio is then $\Theta_k^* = \arg\min \left\{ \frac{1}{|\mathcal{I}|} \sum_{e \in \mathcal{I}} \min \left\{ \mathrm{r}(c, e) \mid c \in \tilde{\Theta} \right\} \mid \tilde{\Theta} \subseteq \Theta,\, |\tilde{\Theta}| = k \right\}$. Thereby, $\mathcal{I}$ denotes the set of instances, Θ the set of possible configurations (see Section 5.5), and $\mathrm{r} \colon \Theta \times \mathcal{I} \to \mathbb{R}^+$ the runtime cost as already given in Definition 1.1.

Table 3: The optimal fixed-size portfolios regarding all possible configuration options (see Section 5.5) using the dataset as described in Section 5.1.

Size	PAR-2	Portfolio Members
(SBS)	3159	Kissat (111111)
1	3106	Kissat (111010)
2	2894	Kissat (111010), SeqFrost (1111)
3	2765	Kissat (101111, 110001), SeqFrost (1101)
4	2688	Kissat (010011, 110001, 111010), SeqFrost (1001)
5	2631	Kissat (010111, 101010, 101111, 110001), SeqFrost (1111)
6	2589	Kissat (010111, 101111, 110001, 111010, 111100), SeqFrost (1001)
7	2553	Kissat (010111, 101111, 110001, 111010, 111100), SeqFrost (0010, 1001)
8	2542	Kissat (000000, 000010, 000111, 101000, 101101, 101111), SeqFrost (0010, 1001)
9	2516	Kissat (000000, 000010, 000111, 101000, 101101, 101111, 0110011), SeqFrost (0010, 1001)
10	2513	Kissat (000000, 000001, 000010, 000111, 101000, 101101, 101111, 110011), SeqFrost (0010, 1001)
(VBC)	2330	All 80 configurations

Table 3 shows the optimal fixed-size portfolios on our dataset as described in Section 5.1. The solver configurations that are part of the optimal fixed-size portfolios are given in

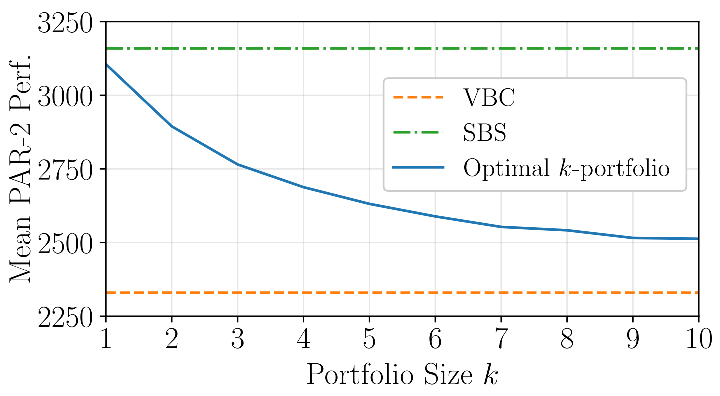


Figure 5: Mean PAR-2 performances of optimal fixed-size portfolios. Results are based on the dataset described in Section 5.1.

the brackets with `1` and `0` denoting that the option is turned on and off, respectively. Kissat options are `bumpreasons`, `chrono`, `compact`, `otfs`, `rephase`, and `tumble` in that order. SeqFrost options are `bumpreason`, `chrono`, `probe`, and `rephase` in that order (cf. Section 5.5). Figure 5 also depicts the mean PAR-2 performance of optimal portfolios with different fixed sizes.

Thereby, we notice three interesting observations. First, the performance improvement of adding one more portfolio member diminishes with bigger portfolio sizes: While the performance difference between the optimal portfolios of size one and two is 211.125 s, the difference between the optimal portfolios of size nine and ten is just 2.804 s. This is also in line with the findings of Bach et al. [11]. Second, all examined portfolios of a size of at least two employ configurations of both solvers: Kissat and SeqFrost. This is not obvious at all since the single-best solver is Kissat by some margin: The default configuration PAR-2 performance of the SeqFrost solver is 3460.267 s, whereas the default configuration PAR-2 performance of the Kissat solver is 3159.195 s. The latter is also equivalent to the single-best solver (SBS) within Figure 5. Finally, the virtually-best configuration baseline (VBC) makes use of *all* 80 configurations rather than using only a subset of the best configurations. This is consistent with the boxplots in Figure 4. For each configuration, there is at least one problem instance on which it is the best. The VBC baseline is also shown in Figure 5. The line representing the optimal fixed-size portfolios will touch the virtually-best configuration baseline for a value of $k = 80$, but not before.

Supervised Configuration Prediction. Since we have obtained ground truth in the form of a complete grid search, we are interested in how well a supervised machine-learning model can predict the best configuration for each problem instance based on instance features. To make the prediction task easier, we only look at the best k-portfolios and predict solver configurations out of those optimal portfolios for all instances. We use 45 features as already introduced in Section 5.1. We then perform a 5-times repeated 5-fold cross-validation with different seeds, i.e., 0, 1, 2, 3, and 4. As a prediction model, we use well-studied random forests [22] with a forest size of 100 trees and the Gini impurity criterion [72]. Table 4 reports Matthew's correlation coefficient scores [24] for predicting the best portfolio member for each problem instance. MCC scores are better suited for imbalanced class labels as is the case here. They range from -1 to 1 with 1 being optimal and 0 being a random guess. To decrease noise, we add a *don't-care* class containing instances

Table 4: Supervised prediction-making of optimal portfolios using the dataset as described in Section 5.1. The table shows the mean performances of all cross-validation test sets.

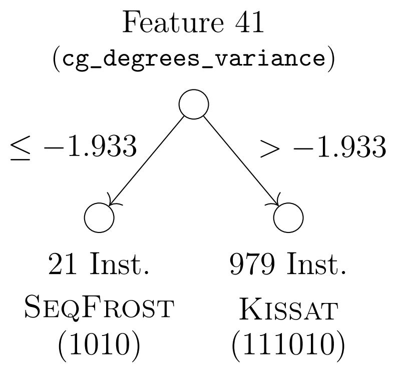
Size	MCC	PAR-2 compared to VBC (only portfolio members)	PAR-2 compared to VBC (all configurations)
1	–	1.000	0.750
2	0.234	0.941	0.757
3	0.232	0.891	0.751
4	0.244	0.873	0.757
5	0.207	0.844	0.748
6	0.248	0.828	0.746
7	0.232	0.822	0.751
8	0.215	0.812	0.745
9	0.207	0.802	0.743
10	0.212	0.802	0.744

with too close portfolio member performances regarding a fixed threshold. Overall, the direct prediction performances are quite poor (MCC scores around 0.2). This is also in line with past observations [25, 48]. Table 4 also shows the performances compared to, both, the virtually-best configuration baseline of only the portfolio members as well as the VBC of all configurations. Thereby, one can observe that none of the portfolio selectors is significantly better than simply picking the single-best configuration (first row of Table 4). The global SBC achieves a score of 0.75 in comparison to the VBC of all configurations. We show those results nevertheless as it explains that steering away from the supervised setting is beneficial.

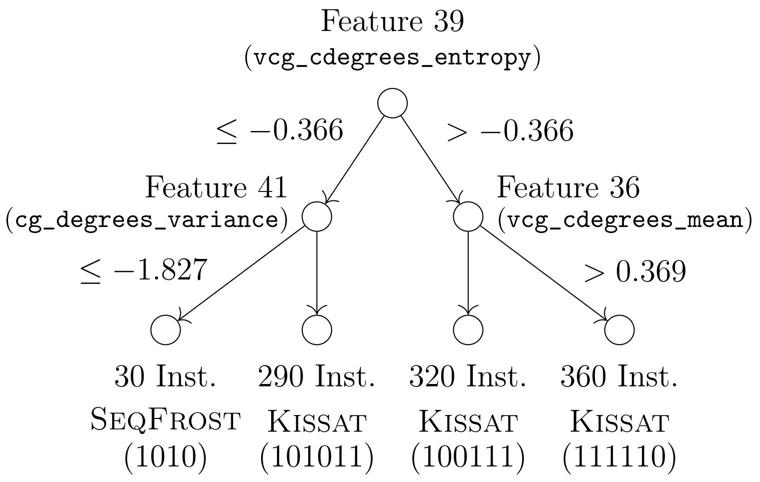
6.2 Optimal Decision Tree Partitioning

Part of this work is concerned with partitioning instance space using decision trees. We, therefore, look at the optimal instance partitionings using decision trees of fixed depth. In other words, we find optimal decision tree splits such that the overall mean PAR-2 performance is optimal. Each partition uses the single-best configuration. We use the dataset as described in Section 5.1 consisting of 1000 instances with full runtime information on 80 configurations.

To find optimal trees, we employ an exhaustive search of all possible partitioning trees of limited depth. Because of the exponentially growing number of leaves and the big number of split-position-feature combinations, we only have been able to compute optimal partitioning trees up to a maximum depth of two. Building optimal decision trees for supervised prediction tasks is also an open research question as this problem is NP-complete [46]. Current state-of-the-art approaches transform optimal-decision-tree problem instances to either SAT instances [10] or mixed-integer-programming instances [18] or employ a branch-and-bound search [3]. The provided implementations are, however, only able to tackle a few hundred instances with up to a small two-digit number of features. Within preliminary experiments, we found that optimal decision trees on a smaller subset of the dataset are not optimal on the original dataset.



(a) Maximum Depth 1.



(b) Maximum Depth 2.

Figure 6: Optimal instance partitioning decision trees with a fixed depth. Results are based on the dataset described in Section 5.1.

Table 5: Quality of optimal partitioning decision trees in comparison to the VBC baseline using only the selected configurations within the decision tree. Results are based on the dataset described in Section 5.1.

Max. Tree Depth	PAR-2 Perf. Optimal Tree	PAR-2 VBC using only the selected configurations within the tree	Fraction
0	3106	3106	1.000
1	3044	2942	0.966
2	2962	2789	0.942

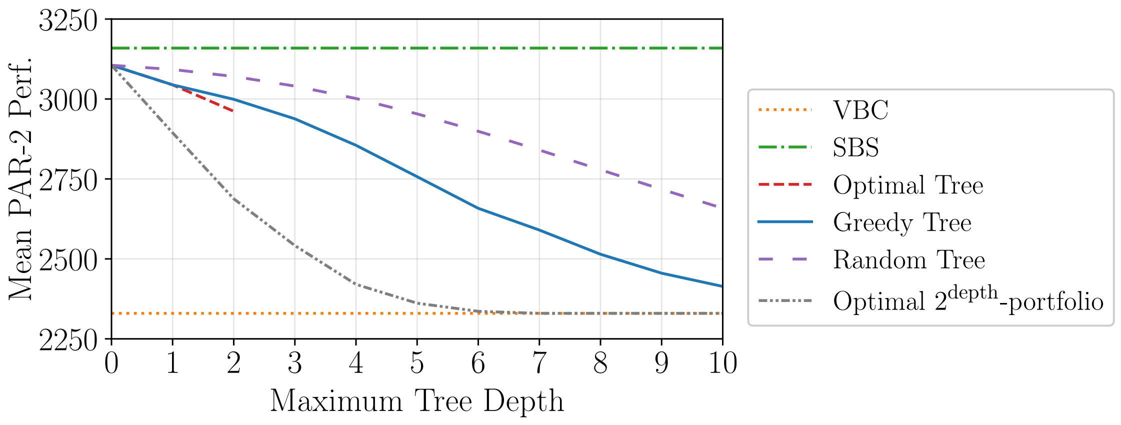


Figure 7: Mean PAR-2 performances of optimal, greedy, and random trees. We average random tree results over 1000 runs. Results are based on the dataset described in Section 5.1.

Figure 6 shows the optimal partitioning decision trees with a fixed depth of 1 and 2, respectively. Each inner node is annotated with the splitting attribute[5]. Refer to Section 5.1 for details on the employed features. In both optimal trees, there is at least one imbalanced split regarding instances in the split's leaves. To exploit the complementarity of the Kissat and SeqFrost solvers, optimal trees try to identify which instances are solved best by which solver. Among leaves mapping to the same solver (cf. Leaf 2-4; Figure 6b), instances are distributed quite evenly.

Moreover, Table 5 shows the mean PAR-2 performance of those optimal trees within the second column for maximum tree depths of 0 (SBC), 1, and 2, respectively. Note that the difference between the optimal tree of depth 0 and 1 (= 61.222 s) is smaller than the difference between depths 1 and 2 (= 82.730 s). This differs from the diminishing returns for increasing portfolio sizes in Figure 5. This is not a contradiction, though, as the number of configurations within a tree is subject to exponential growth with increasing maximum depth.

The third column of Table 5 shows the VBC baseline performance using only the configurations that have been selected by the optimal trees. It is desired that both numbers are close together as this suggests that decision trees are well suited for partitioning instances. The optimal partitioning tree of depth 1 achieves about 97 % of the optimal portfolio performance using the two configurations that have been selected within the optimal tree (Figure 6a). In contrast to that, the optimal partitioning tree of depth 2 only achieves about 94 % of the optimal portfolio performance using the four configurations that have been selected within the optimal tree (Figure 6b). The gap between the optimal tree and the optimal portfolio is expected to further grow with bigger tree sizes since the problem of picking a configuration for each instance is getting more complicated with more configurations to decide amongst.

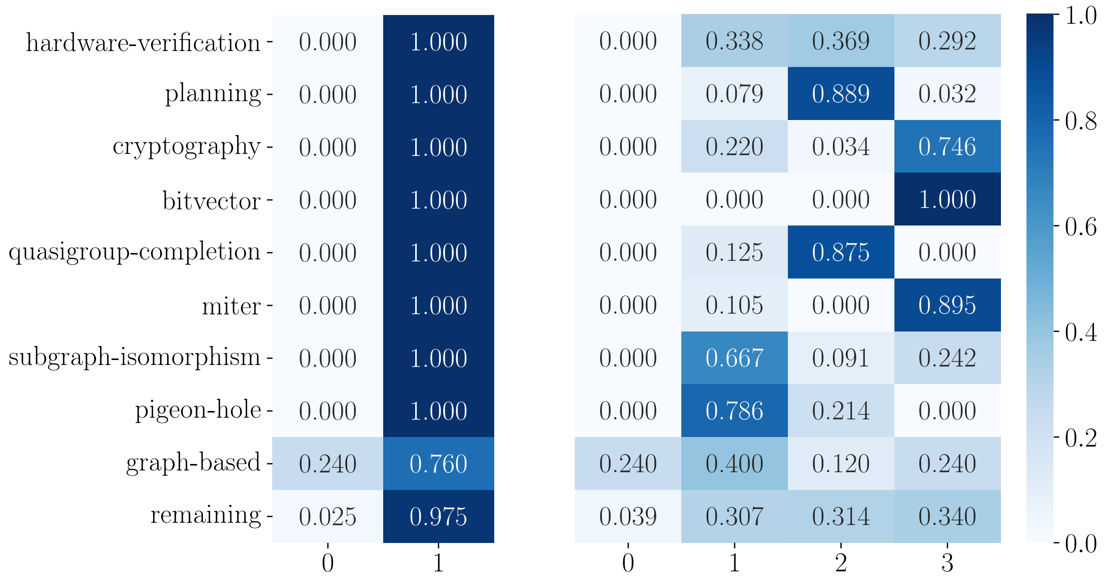
Being unable to produce optimal trees of bigger sizes, we default to the greedy generation of trees, level by level. From the optimal trees of depth 1 and 2 (see Figure 7), it is already clear that this greedy approach is suboptimal. Figure 7 shows the mean PAR-2 performance of optimal and greedy trees in a single plot. Thereby, the single-best default solver (SBS) is an upper-bound and the virtually best configuration (VBC) is a lower-bound for all approaches.

Additionally, we plot how random trees perform (see Figure 7). Thereby, we randomly construct trees of fixed depth and assign the single-best configuration to each leaf. We repeat this 1000 times for each maximum tree depth and average results. Both, greedy and optimal trees are better than random ones by a big margin.

Finally, we also plot the performance of optimal 2^{depth}-portfolios. For a depth of 3, for example, we identify the $2^3 = 8$ solver configurations that form the best portfolio together (cf. Table 3 and Figure 5). Thereby, the margin between the greedy tree performance and the optimal 2^{depth}-portfolio performance is increasing until 2^{depth} exceeds the number of available configurations, i.e., 80 configurations. A possible reason for the increasing difference between both is the more complex decision boundary of assigning each instance to the best possible configuration.

Moreover, we also want to analyze how instance families are distributed among the optimal decision trees' leaves. This provides useful insights into which families are solved best by which configuration if only a limited number of configurations are allowed. Figure 8 shows how instances of the nine biggest instance families are distributed among the optimal tree's

[5] https://benchmark-database.de/getdatabase/base_db



(a) Maximum Depth 1.

(b) Maximum Depth 2.

Figure 8: Fraction of instances of the nine biggest instance families that occur in each of the optimal tree's leaves. Results are based on the dataset described in Section 5.1.

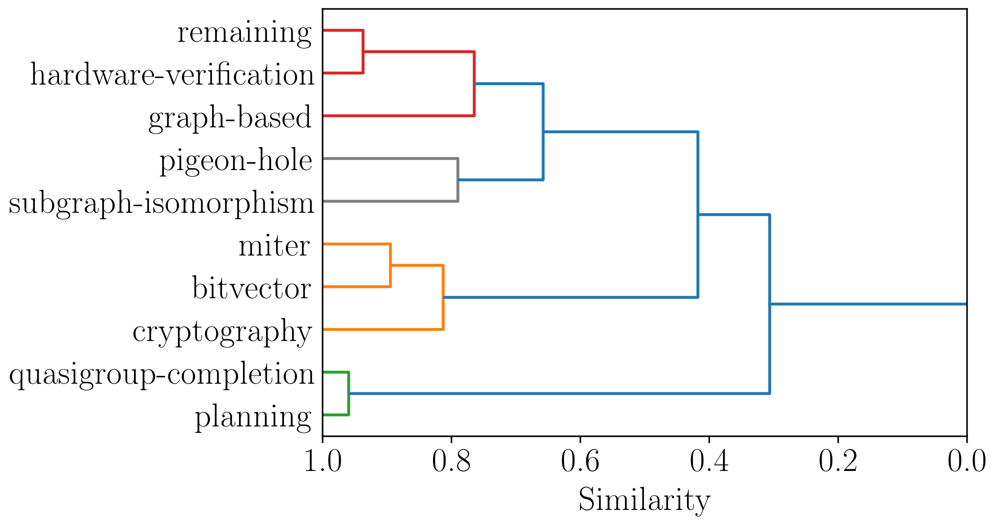


Figure 9: Clustering of the nine biggest instance families based on their distributions in Figure 8b. The more left a joint is, the more similar those families are regarding Euclidean distance. We merge clusters using the linkage type *average*.

partitions. The remaining instances are grouped into a single meta-family. Regarding the optimal tree of depth 1 (see Figure 8a), only a few instances of the *graph-based* family are separated to be solved by SeqFrost. Regarding the optimal tree of depth 2 (see Figure 8b), the picture is more diverse.

When being allowed to pick only four configurations, it is interestingly optimal to choose one configuration of the non-dominant solver within the portfolio, i.e., SeqFrost (Partition 0, Figure 8b), one configuration covering *sugraph-isomorphism* and *pigeon-hole* instances (Partition 1), one configuration covering *planning* and *quasigroup-completion* instances (Partition 2), and one configuration covering *cryptography*, *bitvector*, and *miter* instances (Partition 3). *Hardware-verification* instances and *graph-based* instances have no clear mapping to a single partition.

In addition to that, Figure 9 shows a hierarchical clustering of instance families based on how similar their distributions are regarding the optimal-tree partitions in Figure 8b. We incrementally merge the most similar clusters regarding the *average* linkage criterion using Euclidean distance. To be more specific, given probabilities $p_{f,i} \in [0,1]$ that a problem instance of family f is within partition i of the optimal tree with a maximum depth of two (cf. Figure 6b), the similarity of families f_1 and f_2 is then $1 - \mathrm{dist}((p_{f_1,0}, p_{f_1,1}, p_{f_1,2}, p_{f_1,3}), (p_{f_2,0}, p_{f_2,1}, p_{f_2,2}, p_{f_2,3}))/\sqrt{2}$. We use the Euclidean distance to determine the distance between two families. By doing so, we observe that the *quasigroup-completion* and *planning* families are the most similar. Problem instances of those families end up most often within the same partition of the optimal partitioning decision tree of a maximum depth of two, i.e., partition 2 in Figure 8b. Hence, such instances are solved best by the same configurations. The second most similar cluster is formed by the *miter*, *bitvector*, and *cryptography* families. Problem instances of those families are grouped into partition 3 within Figure 8b. Thereafter, the *pigeon-hole* and *subgraph-isomorphism* families are merged into one cluster. Their instances prefer to be solved best by the same configuration. Finally, problem instances of the *hardware-verification* and *graph-based* family, as well as all remaining instances, have no clear mapping to a single partition as mentioned before.

6.3 PAR-2 Performance with Configuration Oracle

Before looking at the PAR-2 performances and runtime costs of all approaches in Section 6.4 and 6.5, we compare the stand-alone quality of the instance partitioning strategies by using a configuration oracle. Given the set of instances $\mathcal{I}$ and configurations Θ, we use a runtime oracle $\mathcal{R} : \mathcal{P}(\mathcal{I}) \to \Theta$ that returns the single-best configuration for a given subset of instances. We can answer such queries since we have obtained full grid search results on our dataset. Using this oracle, we can examine the quality of the discussed partitioning techniques.

6.3.1 Evolutionary Algorithm Hyperparameters

Using the runtime oracle $\mathcal{R}$, we are interested in the impact that the hyperparameters listed in Section 5.6 have on the overall performance. Figure 10 shows boxplots of the performance impact of all hyperparameters. The boxplots show the distributions of all experiments with the respective hyperparameter value. Again, performance is given relative to the VBC baseline. Also, we only report the performance on the cross-validation test sets. *All* the differences among the impacts of all hyperparameter choices are significant regarding a Wilcoxon signed-rank test with $\alpha = 0.05$ since we deal with a rather large number of hyperparameter combinations: 25 920 combinations. Half of those use tournament selection, for example, and the other half fitness-proportionate selection. The performance differences are quite small, however. We print the single-best combination of hyperparameters in bold letters within Figure 10. We use those hyperparameter values throughout the remaining part of this work. In Table 6, we also give the top-5 single-best hyperparameter choices. Again, the differences among the top-5 combinations are quite small. The differences between the top-5 single-best combinations are *not* significant as the 25 observations, i.e., the 5-times repeated 5-fold cross-validation results, of each hyperparameter combination are not enough to distinguish such a close performance difference.

Initial Population. Regarding the choice of the initial population (cf. Section 4.1), the top-5 single-best configurations (cf. Table 6) make use of either an empty tree or a pre-initialized tree based on a decision tree predicting family membership of instances. The boxplots in Figure 10 support this observation, i.e., they show the advantage of empty-tree or family initialization. By gradually building a tree, the evolutionary algorithm is better at exploring the space of possible trees since we only mutate at the leaf level. By using a family-prediction decision tree, the partitioning tree is already pre-initialized with splitting attributes that help in distinguishing instances.

Fitness. Regarding the employed fitness metric (cf. Section 4.2), there is a tie between the population-based (cf. Definition 4.1) as well as the default-runtime-based fitness metric that acts on a per-instance level (cf. Definition 4.3). Both, produce the exact same performances. For each of the top-5 single-best hyperparameter combinations in Table 6, there exists a hyperparameter combination with equal performance no matter which of the two fitness metrics is used. We omit those duplicates to show a wider range of hyperparameter combinations.

This result is somewhat counter-intuitive: We expected the default-runtime-based strategy, which uses the sum of PAR-2 scores over all instances, to be the best as we optimize for the mean PAR-2 score of an approach. It is inversely proportional to the PAR-2 performance sum. The other two fitness metrics, however, aggregate performances on a *per-instance* level rather than aggregating the performance on all instances. This shows the benefit of looking at the SAT solver performance of all instances separately.

Selection. Regarding the selection component (cf. Section 4.3), the winner among the top-5 hyperparameter combinations is tournament selection (cf. Table 6). It has, however, a similar impact on performance compared to fitness-proportionate selection in Figure 10. We expected a more clear winner as tournament selection involves less stochastic noise. Related work [21] shows the advantage of tournament selection.

Number of Partitions. In related work regarding decision trees, it is long known that smaller trees generalize better to unseen instances [56]. For this reason, we examine rather small numbers of instance partitions. Regarding, both, the overall hyperparameter impact in Figure 10 and the single-best hyperparameter combinations in Table 6, there is no clear

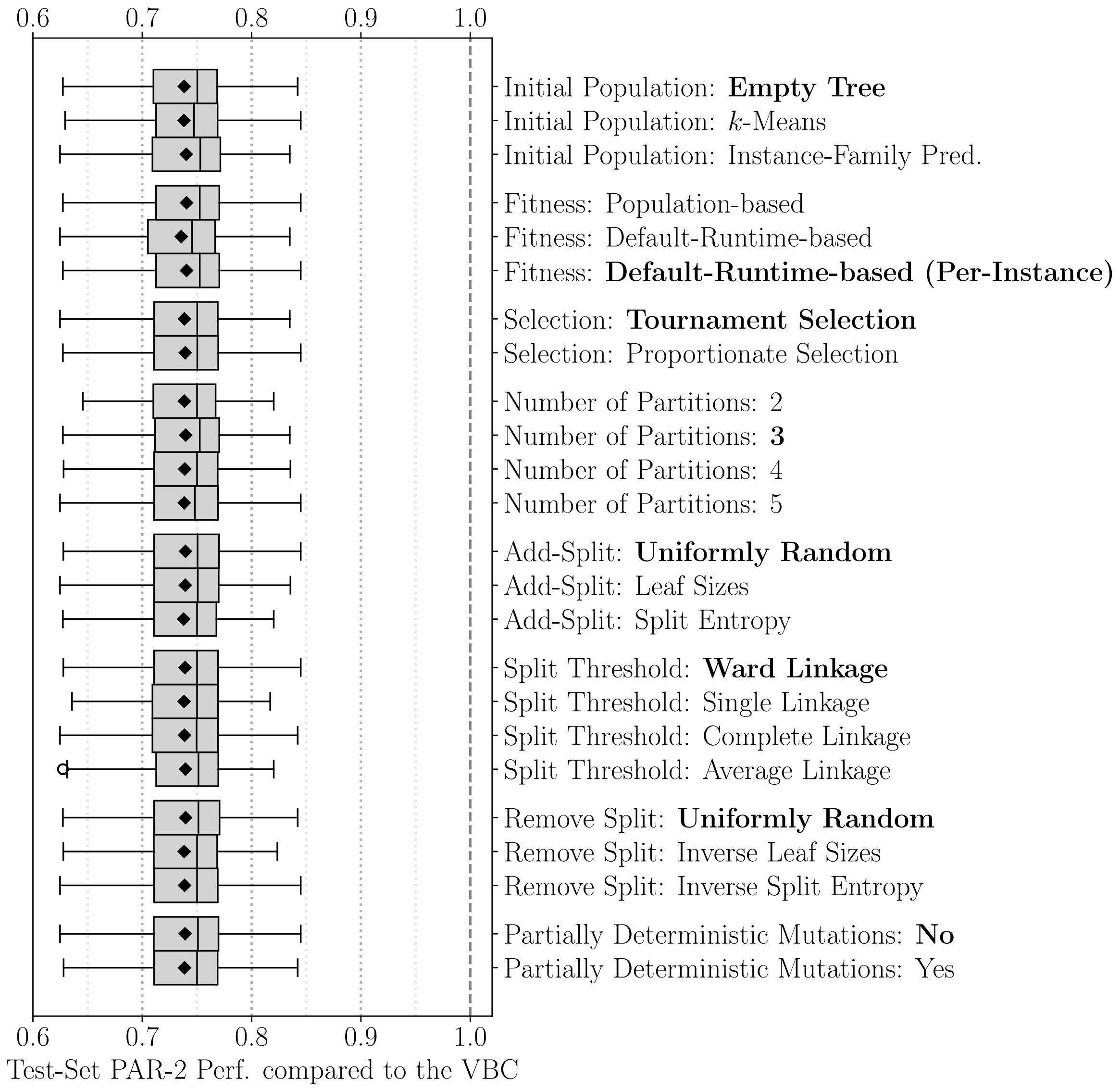


Figure 10: Boxplots showing the impact of all hyperparameters listed in Section 5.6 relative to the VBC baseline. We use a grid search with a runtime oracle (cf. Section 6.3). The diamonds indicate the distributions' means. Moreover, the emphasized choices denote the single-best combination.

Table 6: The top-5 single-best combinations of hyperparameters.

	1	2	3	4	5
Initial Pop.	Empty T.	Fam. Pred.	Fam. Pred.	Empty T.	Empty T.
Fitness	Per Inst.	Per Inst.	Per Inst.	Per Inst.	Per Inst.
Selection	Tournament	Tournament	Tournament	Tournament	Tournament
No. Parts	3	3	4	5	3
Add Split	Random	Random	Random	Leaf Size	Random
Threshold	Ward	Ward	Complete	Ward	Ward
Remove Split	Random	Random	Random	Random	Random
Determinist.	No	No	No	No	No
Perf.	0.7489	0.7484	0.7481	0.7480	0.7477

winner. For robustness, we choose a rather small number of three partitions. Using three partitions is the significantly best option in Figure 10. Differences are quite small though.

Adding a Split. Regarding the add-split criterion (cf. Section 4.4), we select splits uniformly random as it appears among the top-5 single-best configurations. It is significantly better than the other two splitting criteria. Differences are quite small though. By growing the partitioning trees fully randomly, we explore the space of all possibilities.

Splitting Threshold. Regarding the choice of the splitting threshold (cf. Section 4.4), we use hierarchical clustering with one of four possible linkage types: ward, single, complete, or average linkage. Thereby, the ward linkage type is significantly better than the other three linkage types. Differences are quite small though.

Removing a Split. Regarding the remove-split criterion (cf. Section 4.4), the winner is a uniformly random choice of the split to remove. It is significantly better than any of the other approaches. This is also consistent with the add-split criterion. By adding and removing splits fully randomly, we explore the space of possible partitioning trees the most effectively.

Partially Deterministic Mutations. Finally, we also examine whether deterministically choosing the leaves to split or merge has any benefit (cf. Section 4.4). However, choosing splits fully randomly performs the best with a significant difference, which is rather small though.

Summary. To sum up our hyperparameter choices, we evolve partitioning decision trees starting from an empty tree. To do so, we use a per-instance default-runtime-based fitness metric in combination with tournament selection. Moreover, we determine the tree insertion and removal positions for mutations uniformly randomly. Interestingly, the complete selection strategy of our evolutionary approach is fully random. Only the selection pressure is responsible for optimizing the performance of individuals. We use hierarchical clustering with a ward linkage criterion to find the splitting threshold when adding a new splitter. The runtime tuning hyperparameters are the subject of Section 6.4. Therein, we discuss the trade-off between runtime cost and quality of the evolutionary approach.

6.3.2 Baseline Performances

Figure 11 shows performances of all the baselines and approaches within a 5-times repeated 5-fold cross-validation using our dataset (cf. Section 5.1). Thereby, the boxplots show the distributions of the 25 obtained data points per approach. Since each fold of the dataset has slightly varying runtimes, we scale each observed data point r_d by the VBC PAR-2 runtime r_{vbc} of that dataset fold, i.e., $r_{vbc}/r_d \in (0, 1]$, as already mentioned in Section 5.4. We report this relative performance metric for train and test sets separately. As all approaches make use of the runtime oracle $\mathcal{R}$, we achieve different results when adding the runtime tuning component in Section 6.4.

The upper half of Figure 11 shows the baselines from Section 5.2. The SBS boxplots show the performance of the single-best solver, i.e., the Kissat solver with default options.

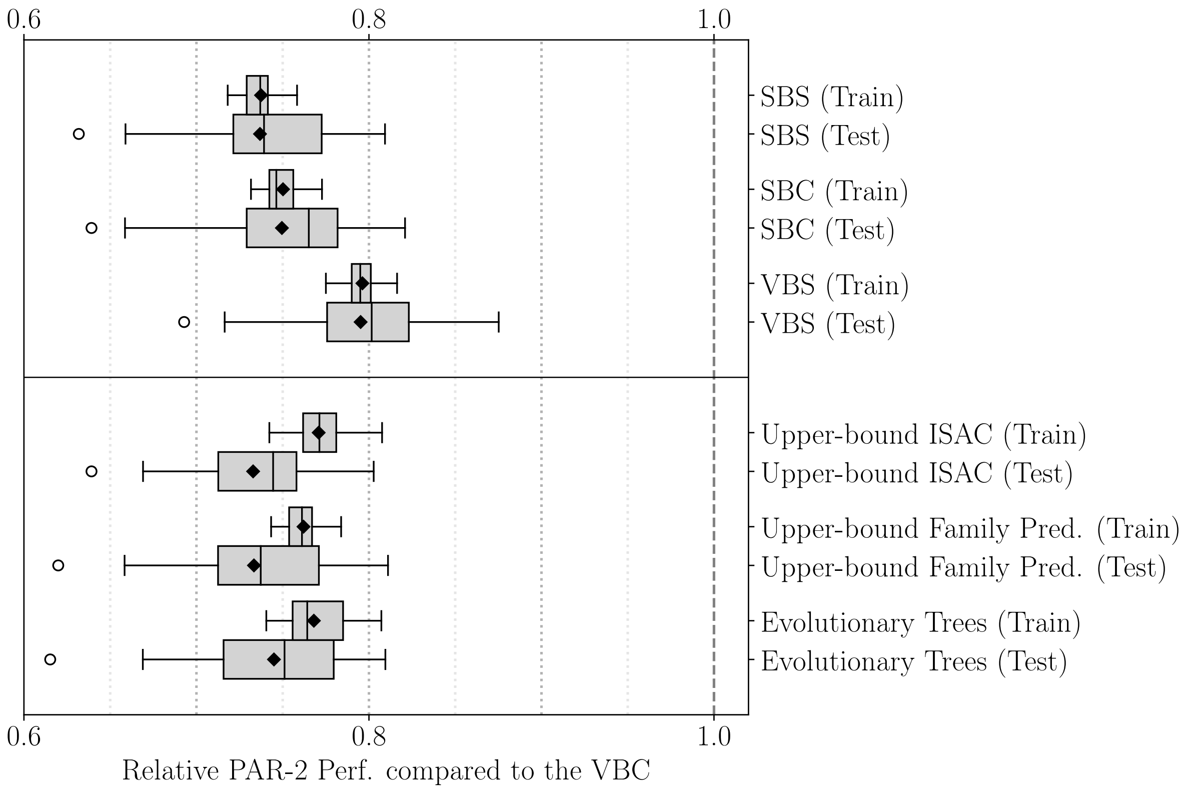


Figure 11: Boxplots of relative PAR-2 performances compared to the VBC baseline. The boxplots contain 25 data points each, i.e., the 5-times repeated 5-fold cross-validation results. Results are based on the dataset as described in Section 5.1. The diamonds indicate the mean performance.

The SBC boxplots show the performance of the single-best configuration, i.e., the Kissat solver with options `111010` (cf. Table 3). Finally, the VBS boxplots show the performance of the virtually-best-solver baseline including the default configurations of the Kissat and SeqFrost solvers. Note that the SBC and VBS baselines require a grid search of all included parameters. All pairs of baseline performances are significantly different regarding a Wilcoxon signed-rank test with $\alpha = 0.05$. The margin between the VBS baseline and the other baselines is bigger than between the SBS and SBC baselines. This indicates that adding a new solver to the portfolio is more beneficial than configuring a single solver, at least for the given distribution of instances. The default configurations of both employed solvers are already quite good since the dataset contains instances that have been used at previous SAT competitions (in which those solvers have competed and they have been optimized for). Also, note that the VBC baseline is represented by the vertical dashed line with $x = 1$. Selecting configurations on a per-instance basis for the configurations listed in Section 5.5 can theoretically achieve a speedup of up to VBC / SBS = 1/0.737 = 1.357 in comparison to the single-best solver on, both, the train and test sets. The train and test set mean performances are both 0.737 for the single-best solver.

6.3.3 Competitor Performances

Apart from that, the lower half of Figure 11 shows the competitors' performances as discussed in Section 5.3. Surprisingly, all competitors are significantly worse than the

VBS baseline on, both, the train and test sets, although the VBS uses only the default configurations of the Kissat and SeqFrost solvers and the runtime oracle provides access to a total of 80 different configurations rather than just 2. Also, none of the competitors, including our approach, is significantly better than the SBC baseline on the test set indicating that simply tuning the best solver provides equal or better test-set performance than any of the discussed strategies. Those observations differ when no runtime oracle is available, i.e., tuning is required (cf. Section 6.4).

ISAC. Starting with the ISAC approach, we cluster train-set instances regarding instance features and assign each cluster the single-best configuration using our oracle $\mathcal{R}$. We then report the test-set performance by using the configuration of the closest cluster for each test-set instance regarding Euclidean distance. While this strategy works quite well on the train set, it does not generalize well on the test set. On the train set, the ISAC approach with runtime oracle performs significantly better than the SBS and SBC baselines. On the independent test set, however, the upper bound of ISAC approaches, as described in Section 5.2, is on average slightly worse than the SBC baseline on the test set. The difference is not significant, though.

Family Prediction. Regarding the family prediction approach, we use the runtime oracle to assign the single-best configuration to instances of each family that comprises at least 5 % of the dataset. The remaining families are grouped into one meta-family. On the test set, we predict an instance's family and use the previously assigned solver configuration. As mentioned in Section 5.3, instance family prediction has an almost perfect accuracy using a standard random-forest model. On the train set, the family prediction approach with runtime oracle is significantly worse than the ISAC approach. There is, however, no significant difference between both on the test set. This indicates that clustering instances by features and predicting an instance's family yields roughly the same performance on unseen instances.

Evolutionary Trees. The evolutionary trees approach has been described in Section 4. We report its performance using the runtime oracle to tune the partitions of the partitioning trees. Thereby, the train-set performance is significantly better than the SBS and SBC baselines as well as the family prediction approach. There is no significant difference in comparison to the ISAC train-set performance. Regarding the test set, the evolutionary trees approach is significantly better than the ISAC and the family prediction approach. It produces the best test-set performances of all non-baseline approaches. There is, however, no significant difference compared to the SBC baseline. Section 6.4 reveals how SBC and our approach behave when no oracle is provided.

Summary. To sum Figure 11 up, the ISAC approach is best regarding the train-set performance. Regarding the test set, the evolutionary tree approach is significantly better than any of the competitors. However, none of the approaches outperforms the SBC baseline on the test set. The difference between the SBC baseline and the evolutionary tree approach is not significant, though. Furthermore, there is a big gap between the VBS and VBC test-set performance and all competitors, indicating further room for improvement.

6.4 Runtime Cost Trade-Off

There is a naturally arising trade-off as we optimize for, both, performance as well as tuning runtime cost. We want to maximize the performance $perf := r_{vbc}/r_d \in (0, 1]$ of the best individual resulting from a run of the evolutionary algorithm as discussed in Section 6.3. At the same time, we want to minimize the evolutionary algorithm's total runtime *cost* including the cost of all experiments.

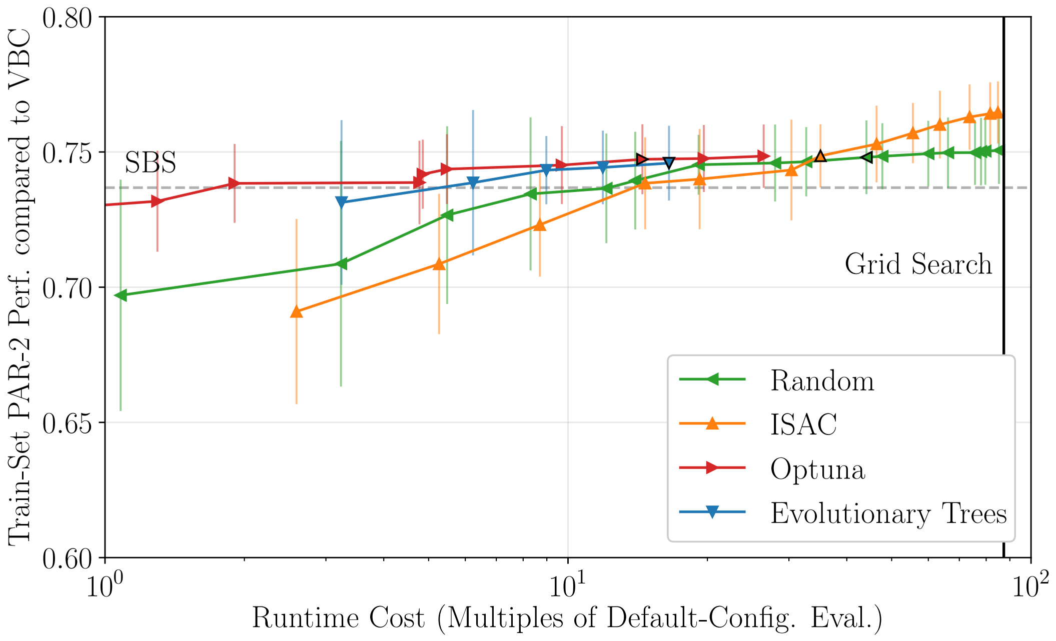
Within this section, we make use of the concept of *Pareto*-optimality. Given a collection of n cost-performance result tuples $(cost_i, perf_i)$ with $i \in \{1, \dots, n\}$, we consider a result tuple $(cost_j, perf_j)$ to be *Pareto*-optimal if and only if there exists *no* $k \in \{1, \dots, n\}$ with $perf_j < perf_k$ and $c_j \geq c_k$. Colloquially speaking, there is no result with better performance for at most the same cost.

Overview. A run of our evolutionary algorithm (cf. Algorithm 1), with a choice for all hyperparameters, is associated with a total runtime cost as well as the performance of the resulting best individual (cf. Section 5.4). Figure 12 shows the total runtime cost on the x-axis and the PAR-2 performance on the y-axis. We only plot Pareto-optimal hyperparameter choices as discussed earlier. Thereby, runtime costs are measured in multiples of evaluating the default configuration of the Kissat solver on the dataset described in Section 5.1. Note that the runtime costs are shown on a logarithmic scale. The solid lines on the right of Figure 12 represent the costs of a complete grid search. The PAR-2 performance on the y-axis is measured relative to the virtually-best configuration performance similar to Section 6.3. Figure 12 then compares four different strategies.

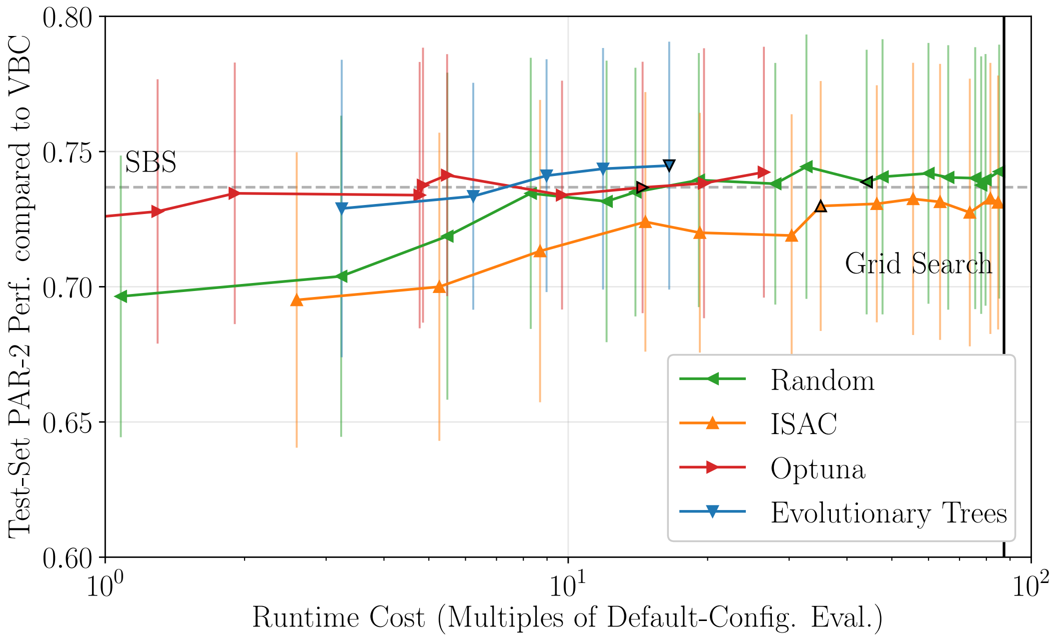
First, we select the single-best configuration regarding a random instance subsample of the dataset of varying fixed sizes. Second, we also show an ISAC implementation using a k-means clustering approach and random sampling for tuning its clusters. Third, we benchmark the performance of the general-purpose tuning library Optuna [5]. We use this tuning framework to find the single-best configuration. It uses a tree-structured Parzen estimator to sample configurations [17] and a random sample of instances to distinguish them. Tree-structured Parzen estimators have been shown to produce significantly better results than random sampling in the hyperparameter optimization domain [17].

Finally, we show different hyperparameter choices of our evolutionary trees algorithm highlighting the best hyperparameter choice of our algorithm with a black border in Figure 12a. We also highlight the hyperparameter choices of the other competitors with similar performances and give absolute numbers of them in Table 7. Figure 12a shows the PAR-2 performances of Pareto-optimal hyperparameter choices on the train set, whereas Figure 12b shows the same hyperparameter choices, that have been Pareto-optimal on the train set, on the test set. The highlighted data points in Figure 12a and 12b show the same hyperparameter choices respectively.

For comparison reasons, we also show the Kissat solver default-configuration runtime (SBS) with its associated standard deviation indicated by the grey area. Test-set standard deviations are generally bigger as the test sets contain 200 instances compared to the 800 instances within the train sets. As a remark, one might note that the single-best solver also precedes heavy experimentation and has already been optimized for a similar instance distribution as ours. That said, it is quite difficult to select a single configuration with significant performance improvement as the default configuration is already the top-7 configuration on our dataset. Throughout our analysis, all tests for significant differences use a Wilcoxon signed-rank test with $\alpha = 0.05$ if not noted otherwise.



(a) Pareto-optimal hyperparameter choices on the *train set*.



(b) Train-set hyperparameter choices on the *test set*.

Figure 12: The trade-off between runtime cost and PAR-2 performance. The plot shows the PAR-2 performance of random sampling, an ISAC implementation, the Optuna tuning framework, as well as our evolutionary-trees algorithm. The error bars indicate the standard deviation regarding the 5 times repeated 5-fold cross-validation. Runtime costs are measured in multiples of evaluating the default Kissat configuration on the full train-set. The performance is given relative to the VBC PAR-2 performance. Results are based on the dataset described in Section 5.1.

Table 7: PAR-2 performances and runtime costs of the baselines, random sampling, an ISAC implementation, the Optuna tuning framework, and our evolutionary algorithm. For all approaches, we show the hyperparameter choice with similar performance to the best evolutionary algorithm hyperparameter choice (highlighted points in Figure 12). We give both, train-set and test-set performances. The last column shows a combined score: For cost c and test-set runtime performance r, we compute $(r_{vbc}/r)\,/\,(c/c_{grid})$. Both, $r_{vbc}/r \in [0,1]$ and $c/c_{grid} \in [0,1]$.

Approach	Mean Runtime Cost (s)	Mean Train-Set PAR-2 Perf. (s)	Mean Test-Set PAR-2 Perf. (s)	Comb. Score
VBC	–	2330	2330	–
SBC (Grid Search)	152 324	3105	3108	0.75
SBS	–	3159	3162	–
Random Sampling	76 939	3115	3154	1.46
ISAC	61 220	3112	3192	1.82
Optuna	25 249	3118	3162	4.45
Evolutionary Trees	28 820	3124	3128	3.94

Table 7 also shows a combined score of each of the aforementioned approaches. Given a tuning cost c and test-set PAR-2 performance r, we compute the combined score with $(r_{vbc}/r)\,/\,(c/c_{grid})$. Both, $r_{vbc}/r \in [0,1]$ and $c/c_{grid} \in [0,1]$. Thereby, r_{vbc} is the PAR-2 runtime of the VBC baseline, i.e., 2330 s, and c_{grid} is the average cost of a complete grid search per each problem instance, i.e., 152 324 s.

Approaches. For the random sampling approach, we randomly sample 10, 20, 30, ..., or all 800 train-set instances and evaluate *all* configurations on them. Then, we plot all Pareto-optimal approaches in Figure 12a. Sampling all 800 train-set instances and evaluating all configurations on them is equivalent to a full grid search. One might also think of sampling both, configurations and instances, but for now, we show the simpler case of only varying the fraction of sampled instances.

For the ISAC implementation, we cluster all train-set instances into 2, 3, 4, or 5 clusters using k-means. Then again, we use random sampling with a fixed sample size to determine the SBC for each cluster. The Pareto-optimal approaches regarding a grid search of k and the fixed sample sizes are shown in Figure 12.

For the Optuna tuning framework, we sample 10, 20, 30, ..., or all 800 train-set instances and give the tree-structured Parzen estimator 5, 10, 15, ..., or 80 iterations to refine its model. Within each iteration, Optuna picks a configuration and evaluates it on the instance subsample. Optuna provides an ask-and-tell interface in order to do so. We show the Pareto-optimal approaches regarding a grid search of those two parameters.

For our evolutionary algorithm, we show the Pareto-optimal approaches regarding a grid search of only the runtime hyperparameters, i.e., the number of minimum observations and the runtime capping factor T_a (cf. Section 5.6). All other hyperparameters are fixed to the best combination regarding the train-set PAR-2 performance using a runtime oracle (cf. Section 6.3).

Random Sampling. The random-sampling performance grows slowly with an increasing amount of tuning runtime regarding, both, the train and test sets (see Figure 12a and 12b respectively). This is quite intuitive: The more runtimes we sample, the more accurate is the choice of the single-best configuration. For this reason, random sampling provides an informative baseline. For higher tuning costs, the performance improvements are diminishing, however. Also, performances do not grow past a certain plateau on the test set, which is slightly above the SBS baseline. In both cases, we do not surpass the single-best solver baseline by much. On the train set, we only achieve a significant difference compared to the SBS baseline using tuning runtime within the *same* magnitude of a full grid search. On the test set, none of the Pareto-optimal random-sampling approaches achieves a significant difference. In Table 7, we show that random sampling takes between double to triple the tuning cost to achieve a similar performance compared to our evolutionary algorithm on, both, the train and test set.

ISAC Implementation. The performance of our ISAC implementation differs greatly regarding the train and test sets (see Figure 12a and 12b respectively). For this reason, we analyze them separately. Regarding the train set (see Figure 12a), we achieve a significantly better performance in comparison to the SBS baseline. We do so, however, at a relatively high runtime cost. In Table 7, we show that ISAC needs more than double the tuning cost to achieve similar results in comparison to our evolutionary algorithm on the train set. With tuning cost close to a complete grid search, ISAC significantly outperforms any other approach on the train set though.

Regarding the test set (see Figure 12b), there is a different picture though. We are not able, not for any runtime cost, to achieve a significantly better ISAC result than the SBS performance. We attribute this to overfitting and the curse of dimensionality [29, 4]: Common clustering approaches, as we have used one, do not perform well in high dimensions as feature space becomes sparser and instances seem more similar. Our ISAC implementation is, therefore, not capable of producing a clustering that generalizes well to the test-set instances. On the training set, ISAC performs quite well since we partition instance space and tune each partition individually. So, even if the partitioning itself is not useful, we get a performance improvement: The more instance partitions we have, the more overfitting we can get. In Table 7, there is no ISAC hyperparameter choice with similar performance compared to our evolutionary algorithm. The best test-set ISAC approach still takes almost four times the tuning cost in comparison to our evolutionary algorithm.

Optuna. Optuna is a general-purpose hyperparameter tuning framework. It is expected that it performs quite well as it contains state-of-the-art tuning algorithms. Indeed, Figure 12b shows that our evolutionary algorithm and Optuna perform quite similarly for more than 10 times the evaluation cost of the default solver. For small tuning costs, however, Optuna performs slightly better (cf. Figure 12b; runtime cost smaller than 10).

Evolutionary Algorithm. Finally, we want to discuss our evolutionary algorithm's performance regarding the train and test sets. The best evolutionary-algorithm hyperparameter choices, in Figure 12a and 12b respectively, are both significantly better than the SBS baseline regarding a Wilcoxon signed-rank test with $\alpha = 0.05$. We have to admit, though, that our best hyperparameter choice on the test set is only barely significant with a p-value of 0.035. We also want to emphasize that our approach has a runtime cost that is magnitudes lower on both data splits than the ISAC approach or random sampling for equivalent performance. Notably, we achieve similar performances and tuning costs as Optuna, a state-of-the-art tuning framework. Table 7 shows that our evolutionary algorithm and Optuna need roughly the same tuning cost for roughly the same performance on both data splits.

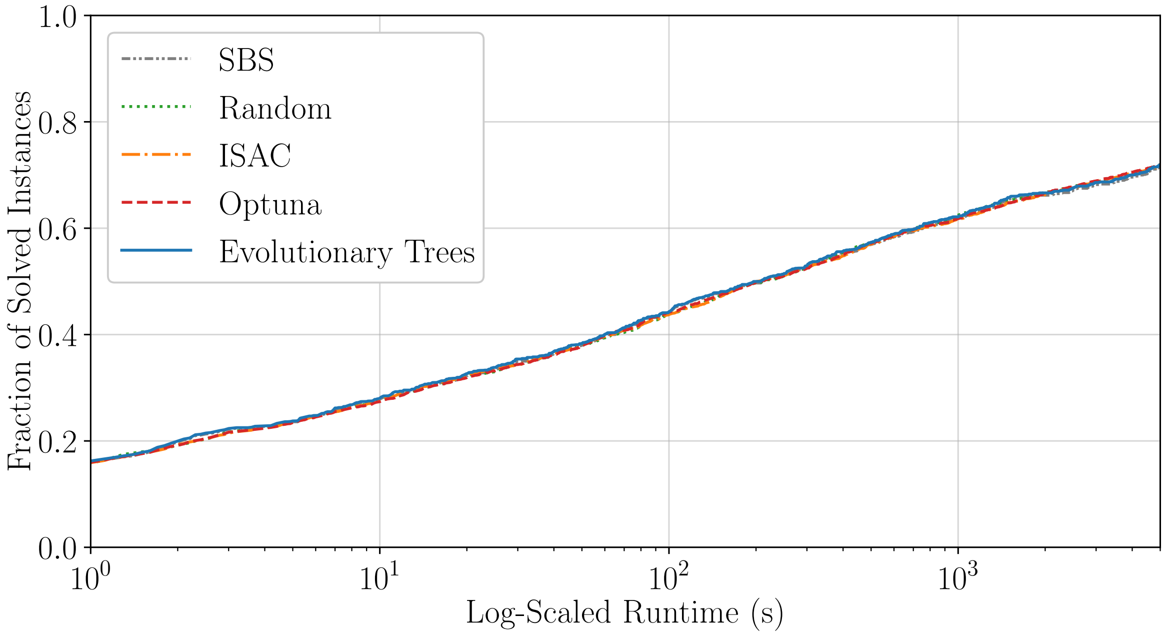
Summary. To sum up this section, we constate that there are more time-efficient tuning strategies than, both, random sampling of configurations and a full grid search. This is consistent with related work in the field of algorithm configuration and hyperparameter optimization [42, 5]. The state-of-the-art tuning framework Optuna, as well as our evolutionary trees algorithm, achieve both comparable results indicating that further refinement of our idea might be beneficial. For small tuning budgets, Optuna is the preferable option though. Moreover, we find that the clustering strategy of the ISAC framework does not generalize well to unseen instances due to the curse of dimensionality in high-dimensional feature space. This conclusion is also supported by the combined scores of each approach in Table 7: Optuna and our evolutionary algorithm achieve the highest scores. In contrast, our ISAC implementation is only slightly better than random sampling and a grid search.

6.5 Runtime Cumulative Disitributions

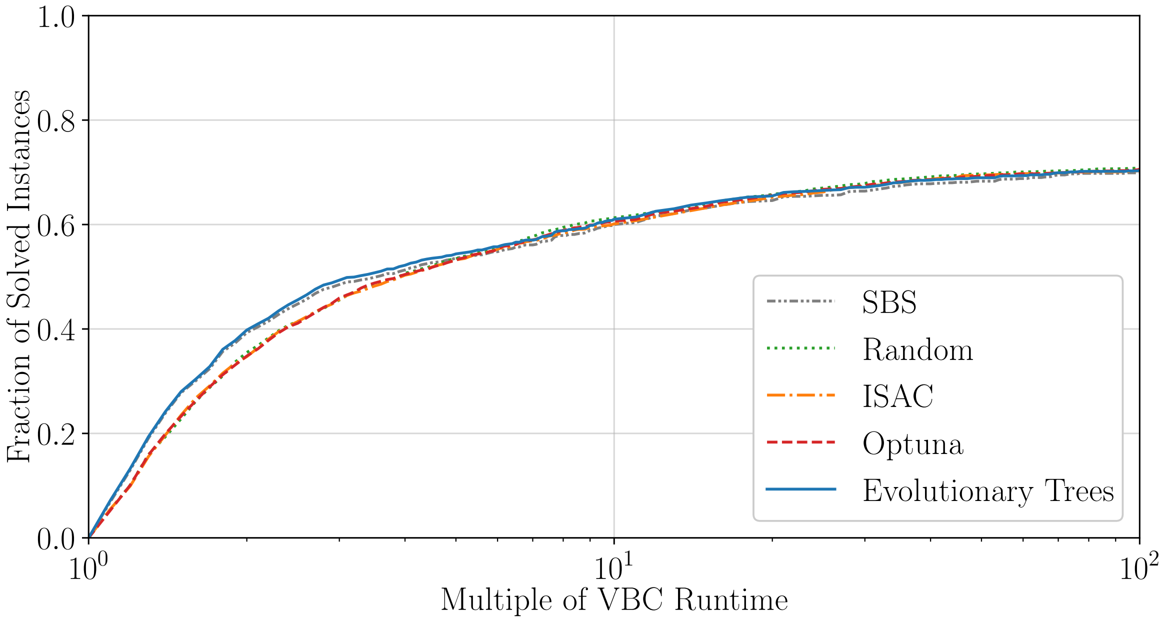
Overview. To further analyze the performance of all aforementioned approaches, we show runtime cumulative-distribution plots in Figure 13. To do so, we use the approaches that are highlighted in Figure 12b and listed in Table 7. As described in Section 5.4, Figure 13a and 13b show par2_rcdf and vbc_rcdf, respectively. In short, given a PAR-2 runtime or a multiple of the VBC runtime, Figure 13 shows the fraction of solved instances within at most the particular timeframe. Thereby, we aggregate the runtime cumulative-distribution plots over all 5-times repeated 5-fold cross-validation folds, i.e., the solid lines indicate the mean performances and the associated areas indicate plus/minus one standard deviation for all shown approaches. Also, note the logarithmic x-axis of both subplots. All results have been gathered on the test sets.

Moreover, Figure 14 shows similar plots. In contrast to Figure 13, we show the differences compared to the SBS runtime cumulative distributions. This allows for a better comparison and for answering questions about where approaches are better or worse than the single-best solver.

PAR-2 Runtime Cumulative-Distributions. Starting with Figure 13a and 14a, we notice that, on average, all listed approaches behave quite similarly. This is somewhat expected since all 25 data folds entail different instances with different instance hardnesses. Also, we selected hyperparameter choices with similar performances within Figure 12b. In Figure 14a, we notice, however, that our evolutionary algorithm performs slightly better than the SBS baseline for instances of all difficulties. In contrast, the other three approaches tend to be slightly worse on easy problem instances, but beneficial on harder ones (runtime greater than 2000 seconds). Differences are not significant, though, as the standard deviations are quite big in comparison to the differences.

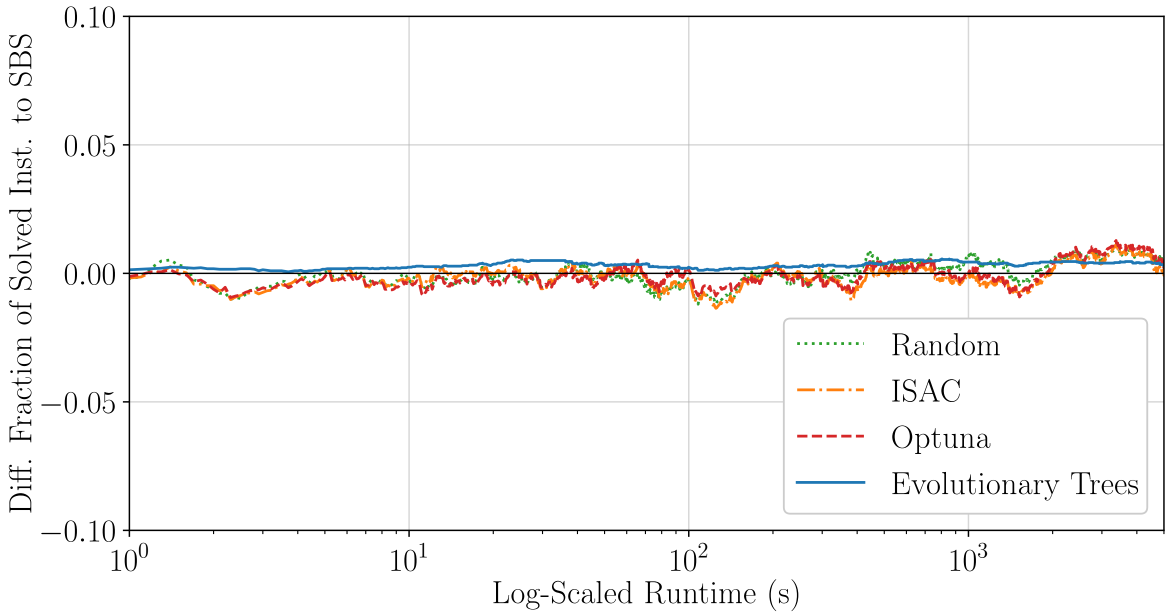


(a) PAR-2 runtime cumulative-distribution plot.

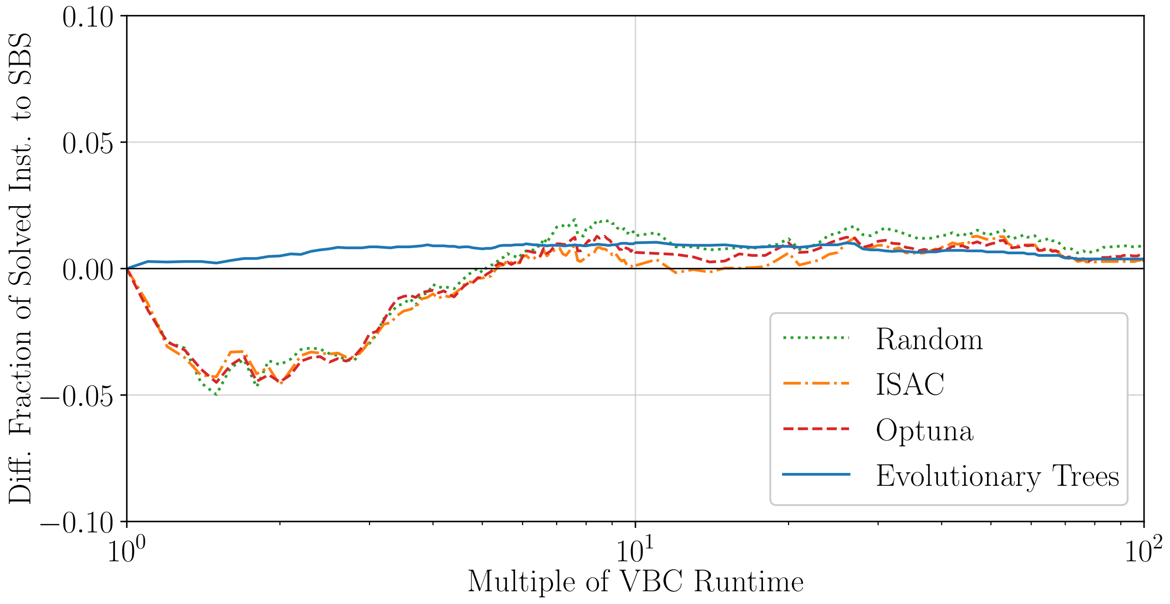


(b) Multiple-of-VBC runtime cumulative-distribution plot.

Figure 13: Runtime cumulative-distribution plots as explained in Section 5.4. We show the mean performances aggregated over all dataset folds and repetitions. The colored areas indicate the standard deviation for each approach. Results are based on the dataset described in Section 5.1.



(a) PAR-2 runtime cumulative-distribution plot showing the difference to the SBS.



(b) Multiple-of-VBC runtime cumulative-distribution plot showing the difference to the SBS.

Figure 14: Runtime cumulative-distribution plots as explained in Section 5.4. We plot the differences in comparison to the SBS baseline. Moreover, we show the mean performances aggregated over all dataset folds and repetitions. The colored areas indicate the standard deviation for each approach. Results are based on the dataset described in Section 5.1.

Multiple-of-VBC Runtime Cumulative-Distributions. Furthermore, Figure 13b and 14b show the same data as before, but instead of showing absolute PAR-2 performance, we express the specific runtimes of problem instances as a multiple of the best-known runtime on each respective instance: In contrast to the absolute runtimes in seconds, we use runtime measured in multiples of the VBC runtime. A value of two, for example, means that we need twice the runtime of the VBC baseline. The advantage of doing so is that we are less sensitive to varying instance difficulties within different test sets, i.e., twice as much runtime is a relative measure. A disadvantage is though that we value easy instances over-proportionally as it is easier to get a bigger improvement regarding multiples of the VBC runtime.

In Figure 13b and 14b, we notice significant differences between our evolutionary algorithm and all competitors for the range of 1.5 to 3 times the VBC runtime. Our approach solves the highest fraction of problem instances within the aforementioned timeframe.

That our evolutionary algorithm is consistently slightly better than the SBS baseline on all four subplots, whereas there is more fluctuation regarding the competitors, is no surprise. The selected hyperparameters of our evolutionary algorithm involve a fitness metric that aggregates performance on a per-instance level. Therefore, our approach favors improving the runtime of the highest fraction of instances, even if it is only a slight improvement. By contrast, the other approaches optimize for the resulting PAR-2 runtime leading to the improvement of only a few hard instances (cf. Figure 14a). Hyperparameter analysis shows that the former is beneficial for our purposes (cf. Section 6.3.1).

Summary. To sum up this section, we looked at the runtime cumulative distributions of random sampling, an ISAC implementation, the Optuna framework, and our evolutionary trees algorithm. Overall, the performances of the approaches are quite similar since we picked hyperparameter choices with similar performances for comparison purposes and average results over 25 datasets. There is only one major difference in Figure 13a and 14a. Our approach optimizes for the runtime improvement of the highest fraction of problem instances, whereas random sampling, the ISAC implementation, and Optuna favor improving fewer harder problem instances (cf. Figure 14a and 14b). In the end, improving the runtime of many easier or fewer harder problem instances, both, gets the job done.

7 Conclusion

The archetypal NP-complete propositional satisfiability problem (SAT) is a versatile tool in modeling various applications ranging from cryptography [66] to planning and scheduling [34]. In this work, we have looked at automatically tuning and selecting propositional satisfiability solvers.

Tuning algorithms by changing their parameters is known as algorithm configuration [42, 5], whereas selecting the best algorithm from a set of known ones is called algorithm selection [41]. Since SAT problem instances are derived from a multitude of domains, the best algorithms differ greatly depending on which instances are in the dataset. Therefore, we consider the notion of per-set algorithm configuration, which specifies an algorithm configuration for each region of problem instances regarding some feature space. Thereby, per-set algorithm configuration is a superset of, both, algorithm selection and configuration (cf. Section 3).

We proposed an evolutionary algorithm that evolves instance partitioning decision trees for per-set SAT solver configuration. We discussed that our approach fulfills several desirable criteria including interpretability and robustness. It is interpretable in the sense that the partitioning decision tree provides insights about what instance features lead to a configuration decision. Moreover, we presented a clustering of instance families based on optimal partitioning decision trees and showed which instance families are most similar. Partitioning instances with the aid of decision trees is also robust as we tune an SAT solver for a region of instances rather than for single problem instances. By doing so, we generalize better to new instances. We showed that the train-set and test-set performances of our approach are quite close together.

Furthermore, we discussed that our per-set algorithm configurator, i.e., the evolutionary trees approach, has similar PAR-2 performances and tuning costs in comparison to the state-of-the-art algorithm configurator Optuna [5] with which we wanted to find the single-best configuration. We introduced a scoring combining, both, performance and tuning costs showing that Optuna and our approach are superior to random sampling and a grid search. While Optuna has the best score, the difference compared to our approach is not big. In contrast, our ISAC implementation was only slightly better than random sampling and a grid search.

Also, we showed that the ISAC approach lacks generalizability because of the curse of dimensionality [4]. The cluster-then-tune ISAC framework achieves good train-set results. However, because the feature space is high dimensional, the learned clusters do not perform well on the test set. Our experiments showed that partitioning instance space with the aid of decision trees is more robust.

We introduced a novel per-set algorithm configuration approach evolving instance partitions using decision trees. We showed comparable performance in comparison to the state-of-the-art algorithm configurator Optuna [5]. Finally, we discussed why the ISAC framework underperforms on the test sets and lacks generalizability.

7.1 Future Work

In the future, we might look into further improving the performance of our evolutionary trees approach. This includes the analysis of more advanced evolutionary selection, mutation, and cross-over strategies as well as a better tuning algorithm. Possible tuning algorithms include, for example, tree-structured Parzen estimators [17]. They can model prior knowledge in the form of a kernel-density estimator. Also, they are the main tuning algorithm within the Optuna framework [5].

Furthermore, it makes sense to expand our analysis to more datasets to get a better overall picture. With more datasets, we also mean domains other than satisfiability solving as tuning frameworks are especially prominent in the machine-learning field. Hansen et al. [37], for example, propose a platform for comparing black-box optimizers on the black-box optimization benchmark [9].

References

[1] Tinus Abell, Yuri Malitsky, and Kevin Tierney. Fitness landscape based features for exploiting black-box optimization problem structure. Technical report, IT-Universitetet i København, 2012.

[2] Steven Adriaensen, André Biedenkapp, Gresa Shala, Noor H. Awad, Theresa Eimer, Marius Lindauer, and Frank Hutter. Automated dynamic algorithm configuration. *CoRR*, abs/2205.13881, 2022.

[3] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3146–3153. AAAI Press, 2020.

[4] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 94–105. ACM Press, 1998.

[5] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 2623–2631. ACM, 2019.

[6] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving csps. In Carla P. Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 316–324. Springer, 2013.

[7] Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney. Model-based genetic algorithms for algorithm configuration. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 733–739. AAAI Press, 2015.

[8] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2009.

[9] Anne Auger, Dimo Brockhoff, Nikolaus Hansen, Dejan Tusar, Tea Tusar, and Tobias Wagner. Gecco'16 black-box optimization benchmarking workshop (BBOB-2016): workshop chairs' welcome message. In Tobias Friedrich, Frank Neumann, and Andrew M. Sutton, editors, *Genetic and Evolutionary Computation Conference, GECCO 2016, Denver, CO, USA, July 20-24, 2016, Companion Material Proceedings*, page 1167. ACM, 2016.

[10] Florent Avellaneda. Efficient inference of optimal decision trees. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3195–3202. AAAI Press, 2020.

[11] Jakob Bach, Markus Iser, and Klemens Böhm. A comprehensive study of k-portfolios of recent SAT solvers. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 2:1–2:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[12] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evol. Comput.*, 1(1):1–23, 1993.

[13] Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2022.

[14] Thomas Bartz-Beielstein, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. Evolutionary algorithms. *WIREs Data Mining Knowl. Discov.*, 4(3):178–195, 2014.

[15] Nacim Belkhir. *Per Instance Algorithm Configuration for Continuous Black Box Optimization. (Configuration Automatisé d'algorithme par instance pour l'optimisation numérique boîte-noire)*. PhD thesis, University of Paris-Saclay, France, 2017.

[16] Nacim Belkhir, Johann Dréo, Pierre Savéant, and Marc Schoenauer. Per instance algorithm configuration of CMA-ES with limited budget. In Peter A. N. Bosman, editor, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, pages 681–688. ACM, 2017.

[17] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, pages 2546–2554, 2011.

[18] Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Mach. Learn.*, 106(7):1039–1082, 2017.

[19] Armin Biere, Katalin Fazekas, Mathis Fleury, and Maximilian Heisinger. CaDiCaL, kissat, paracooba, plingeling and treengeling entering the SAT competition 2020. *SAT Competition 2020*, 2020.

[20] Armin Biere and Daniel Kröning. Sat-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018.

[21] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evol. Comput.*, 4(4):361–394, 1996.

[22] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[23] Tom Carchrae and J. Christopher Beck. Applying machine learning to low-knowledge control of optimization algorithms. *Comput. Intell.*, 21(4):372–387, 2005.

[24] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21(1):6, 2020.

[25] Marco Collautti, Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. SNNAP: solver-based nearest neighbor for algorithm portfolios. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Zelezný, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III*, volume 8190 of *Lecture Notes in Computer Science*, pages 435–450. Springer, 2013.

[26] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[27] Katharina Eggensperger, Marius Lindauer, and Frank Hutter. Pitfalls and best practices in algorithm configuration. *J. Artif. Intell. Res.*, 64:861–893, 2019.

[28] Halima Elaidi, Zahra Benabbou, and Hassan Abbar. A comparative study of algorithms constructing decision trees: ID3 and C4.5. In Saaid Amzazi, Abdellatif El Afia, and Mohammed Essaaidi, editors, *Proceedings of the International Conference on Learning and Optimization Algorithms: Theory and Applications, LOPAL 2018, Rabat, Morocco, May 2-5, 2018*, pages 26:1–26:5. ACM, 2018.

[29] Levent Ertöz, Michael S. Steinbach, and Vipin Kumar. Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. In Daniel Barbará and Chandrika Kamath, editors, *Proceedings of the Third SIAM International Conference on Data Mining, San Francisco, CA, USA, May 1-3, 2003*, pages 47–58. SIAM, 2003.

[30] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2962–2970, 2015.

[31] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2020. *Artif. Intell.*, 301, 2021.

[32] Tobias Fuchs, Jakob Bach, and Markus 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. Springer Nature, 2023.

[33] Fei Geng, Lei Yan, and ShuCheng Zhang. Kissat-ELS and its friends at the SAT Competition 2022. *SAT Competition 2022*, 2022.

[34] Stephan Gocht and Tomás Balyo. Accelerating SAT based planning with incremental SAT solving. In Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith, editors, *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, pages 135–139. AAAI Press, 2017.

[35] Evguenii I. Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. In *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, pages 142–149. IEEE Computer Society, 2002.

[36] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.

[37] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tusar, and Dimo Brockhoff. COCO: a platform for comparing continuous optimizers in a black-box setting. *Optim. Methods Softw.*, 36(1):114–144, 2021.

[38] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evol. Comput.*, 9(2):159–195, 2001.

[39] Marijn Heule, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2018.

[40] Holger H. Hoos. Programming by optimization. *Commun. ACM*, 55(2):70–80, 2012.

[41] Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Automated configuration and selection of SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 481–507. IOS Press, 2021.

[42] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. LION*, pages 507–523, 2011.

[43] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. *J. Artif. Intell. Res.*, 36:267–306, 2009.

[44] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1152–1157. AAAI Press, 2007.

[45] Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger H. Hoos, and Kevin Leyton-Brown. The configurable SAT solver challenge (CSSC). *Artif. Intell.*, 243:1–25, 2017.

[46] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Inf. Process. Lett.*, 5(1):15–17, 1976.

[47] Markus Iser and Carsten Sinz. A problem meta-data library for research in SAT. In *Proc. PoS*, pages 144–152, 2018.

[48] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - instance-specific algorithm configuration. In Helder Coelho, Rudi Studer, and Michael J. Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 751–756. IOS Press, 2010.

[49] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evol. Comput.*, 27(1):3–45, 2019.

[50] Donald E. Knuth. Satisfiability and the art of computer programming. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, page 15. Springer, 2012.

[51] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in WEKA. *J. Mach. Learn. Res.*, 18:25:1–25:5, 2017.

[52] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3:

A versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, 23:54:1–54:9, 2022.

[53] Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *J. Artif. Intell. Res.*, 53:745–778, 2015.

[54] C. Luo and H. H. Hoos. Sparkle sat challenge 2018. `http://ada.liacs.nl/events/sparkle-sat-18`, 2018. Last accessed on 14 December 2022.

[55] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 127–149. IOS Press, 2021.

[56] John Mingers. An empirical comparison of pruning methods for decision tree induction. *Mach. Learn.*, 4:227–243, 1989.

[57] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[58] Mario A. Muñoz, Michael Kirley, and Saman K. Halgamuge. A meta-learning prediction model of algorithm performance for continuous optimization problems. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII - 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I*, volume 7491 of *Lecture Notes in Computer Science*, pages 226–235. Springer, 2012.

[59] Yanik Ngoko, Christophe Cérin, and Denis Trystram. Solving sat in a distributed cloud: A portfolio approach. *Int. J. Appl. Math. Comput. Sci.*, 29(2):261–274, 2019.

[60] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random SAT: beyond the clauses-to-variables ratio. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 438–452. Springer, 2004.

[61] Muhammad Osama and Anton Wijs. SEQFROST at the sat race 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2022*, 2022.

[62] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.*, 12(85):2825–2830, 2011.

[63] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *CoRR*, abs/1811.12808, 2018.

[64] John R. Rice. The algorithm selection problem. *Adv. Comput.*, 15:65–118, 1976.

[65] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD*

1996, San Jose, CA, USA, November 10-14, 1996, pages 220–227. IEEE Computer Society / ACM, 1996.

[66] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[67] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-weka: combined selection and hyperparameter optimization of classification algorithms. In Inderjit S. Dhillon, Yehuda Koren, Rayid Ghani, Ted E. Senator, Paul Bradley, Rajesh Parekh, Jingrui He, Robert L. Grossman, and Ramasamy Uthurusamy, editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 847–855. ACM, 2013.

[68] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Evaluating component solver contributions in portfolio-based algorithm selectors. In *Proc. SAT*, pages 228–241, 2012.

[69] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.

[70] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Features for SAT. Technical report, University of British Columbia, 2012.

[71] Lin Xu, Frank Hutter, Jonathan Shen, Holger Hoos, and Kevin Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.

[72] Ye Yuan, Liji Wu, and Xiangmin Zhang. Gini-impurity index analysis. *IEEE Trans. Inf. Forensics Secur.*, 16:3154–3169, 2021.