Universiteit Leiden
The Netherlands

Sampling strategies in automated algorithm configuration

Anastacio, M.I.A.

5

Sampling instances to compare the performance of algorithms

Empirical performance evaluation plays a critical role in algorithm configuration and performance optimisation, be it automated or manual. Whilst for a configurator we would need to compare two configurations of the same algorithm, this chapter[1] focus on approaches to compare two algorithms against each other. This problem arises in competitions and scientific publications aimed at improving the state of the art in solving many automated reasoning problems, such as Boolean satisfiability (SAT), constraint satisfaction problem (CSP) and Bayesian network structure learning (BNSL). We explore the intuition that the decision to keep or discard an algorithm can be taken earlier by carefully selecting on which instances to evaluate its performance. By performing runs on carefully chosen problem instances, we minimise the computational cost of running algorithms, whilst probabilistic tests allow us to control the desired statistical significance of observed performance differences. We describe a set of methods for this purpose and evaluate their efficacy on diverse datasets from SAT, CSP and BNSL. On all these datasets, most of our approaches were able to select the correct algorithm with approximately 95% accuracy, while using less than one-third

[1] Parts of this chapter have been published as Matricon et al. (2021)

of the CPU time required for a comparison on the full instance set. The best methods achieve this level of accuracy within less than 15% of the CPU time needed for a complete comparison. Intuitively, we expect the behaviour of two algorithms to be more distinct than that of two configurations of a single algorithm. The transfer of the evaluated methods to automated algorithm configuration (AAC) will be further explored in the subsequent chapter.

5.1 Introduction

From the evaluation of early algorithms against the human ability to solve given instances by hand (Davis and Putnam, 1960) to extensive competitions requiring CPU years to determine a winner (Heule et al., 2019; Pulina and Seidl, 2019; Sutcliffe, 2020), the amount of computational resources needed to assess empirically whether an algorithm exceeds state-of-the-art performance is growing along with the ability of state-of-the-art solvers to tackle larger instances.

Here, we address this issue by focusing on the instance space and on techniques for identifying instances that help discriminate between the compared algorithms. We argue that carefully selecting instances and avoiding long evaluations that provide only a limited amount of information allows us to decide earlier when to stop running a less promising algorithm. Despite similarities with existing problems, such as active learning and algorithm selection (AS), this problem has a different objective, and it is thus not possible to apply existing methods directly.

5.1.1 Background

To the best of our knowledge, the problem we address has not been previously studied in the literature. However, similar questions appear in other settings. Some early SAT Competitions (see *e.g.* the 2002 competition Simon et al., 2005) have been organised in two stages: first, they ran all the solvers on a subset of hand-picked instances to extract the top performers, which were then run on all problem instances. The subset was selected by experts, requiring extensive knowledge and understanding of the problem instances at hand, which is not readily available to an automated system.

To decide on which instance the algorithm should run, we assign a score to each instance (see Section 5.3) based on existing approaches from the literature. In the context of instance generation for CP problems, Gent et al. (2014) proposed a method for defining the discrimination power of an instance, enabling the generation of problem instances for model selection based on samples of running times. This method does not address our aim to reduce running time, but it could lead to selecting relevant instances. We included this method in our comparison, with minor adjustments to account for our objective of minimising running time. Note that after the publication of the work described in the current chapter, Bossek and Wagner (2021) developed an explicit-ranking method as a fitness function for evolutionary algorithms, in order to generate instances that follow a given ranking. Their ranking maximises the similarity between the ranking of the algorithms and the difference in their running times on

this instance.

Once we know which instances should be solved, the next decision is whether to stop or continue the comparison. This question also arises in other situations. In AAC, comparing the performance of two configurations is a key element (as explained in Section 2.2.2). SMAC and ROAR (Hutter et al., 2011b), as well as irace (López-Ibáñez et al., 2016), pick uniformly at random the instances on which they run it, without considering prior knowledge they gathered. irace is based on earlier work from Maron and Moore (1997), which aimed at comparing many machine learning models on a subset of test points to estimate their accuracy with a certain statistical confidence. In this line of work, irace requires evidence in the form of a statistical test to decide when to stop running a less promising configuration. SMAC and ROAR, on the other hand, compare the raw performance metric. We included the statistical test from irace in our experiments.

Our problem is also related to the per-instance algorithm selection (AS) problem (Kerschke et al., 2019, see *e.g.*) in which one tries to predict on which algorithm a specific instance should be run to be solved with the best possible performance. However, there are key differences that prevent us from using directly the methods developed for AS; typically, their use requires prior knowledge in the form of instance features, which we do not always assume to have, and the running time of the algorithms on other instances, which we do not have available for the new algorithm. Additionally, our primary goal is to reduce the time required to determine which one is the best.

Finally, there is a significant link with problems tackled by active learning methods (Sun and Wang, 2010), particularly the pool-based selective sampling problem, which seeks to choose an instance from a set on which the model should be trained next. The idea is that a relevant instance should have a high impact on the model, (*e.g.* increasing its accuracy or reducing its variance). This is closely related to our problem, but differs in that the chosen instance should also lead to low running times, which is not a common objective in active learning. Those methods are aimed at a machine learning model, and the choice of an instance is based on the impact it may have on the model, (*e.g.* reducing its variance or expected error).

In this chapter, we assume the accessibility to the empirical performance data of the studied algorithms. We focus on situations in which we cannot or do not want to learn a performance model for comparing the two algorithms, *e.g.* in the context of developing a new solver or running a competition. Chapter 6 will delve further into model-based approaches when incorporating these methods into a configurator.

5.1.2 Research question

RQ6 *How can we smartly select on which instances to run our evaluation to lower the time spent evaluating bad algorithms?*

We introduce the per-set efficient algorithm selection problem (PSEAS): Given two algorithms, an incumbent A_{inc} and a challenger A_{ch}, and a set of problem instances $\mathcal{I}$, how can we minimise the computational resources (here: CPU time) required to determine, at a required level of confidence, whether A_{ch} performs better than A_{inc} on $\mathcal{I}$? In the following, we describe five methods for selecting on which instances to run the competing algorithms, and two methods for deciding when to stop the evaluation. We compare the ten resulting approaches on four benchmarks for classic computational problems: SAT, CSP and BNSL. On these datasets, our approaches can determine the better-performing algorithm with up to 98% accuracy, while using less than a third of the CPU time required for a full comparison, and the best methods achieve this level of accuracy within less than 15% of the CPU time for an exhaustive comparison.

5.2 Model-free instance selection

To answer our research question RQ6, we place ourselves in the context of comparing the running time performance of two algorithms and define the problem at hand.

Definition of the per-set efficient algorithm selection problem (PSEAS)

We let $\mathcal{I}$ denote the set of instances, $T_{\text{cut}} \in \mathbb{R}^+$ the cutoff threshold, m the performance metric that evaluates an algorithm on an instance, and c the cost function that evaluates the cost of running an algorithm on an instance. We consider two algorithms: the incumbent A_{inc} and the challenger A_{ch}, and assume that the cost $c(A_{\text{inc}}, I)$ and performance $\mathcal{M}(A_{\text{inc}}, I)$ of running A_{inc} on an instance I is known for all instances, whereas these quantities are unknown on all instances for A_{ch}. This assumption is consistent with the fact that A_{inc} represents the state of the art, hence can be assumed to have been evaluated on many problems. The problem is to determine which of the two algorithms performs best according to $\sum_{I \in \mathcal{I}} \mathcal{M}(A_{\text{ch}}, I)$ while running A_{ch} only on a subset $\mathcal{I}_{\text{run}} \subset \mathcal{I}$ that minimises the cost $\sum_{I \in \mathcal{I}_{\text{run}}} c(A_{\text{ch}}, I)$.

Here, we pose $\mathcal{A}$ a set of algorithms, including A_{inc}, regarding which we have prior knowledge in the form of their costs and performances on (at least part of) the instances from $\mathcal{I}$. Unless stated otherwise, we write I for an instance in $\mathcal{I}$ and A for an algorithm in $\mathcal{A}$. For simplicity, we consider the algorithms to be deterministic; hence,

for an algorithm $A \in \mathcal{A} \bigcup \{A_{\text{ch}}\}$, we define the running time as $rt(A, I) \in [0, T_{\text{cut}}]$ for an instance I. We define $\mathcal{M}(A, I) = c(A, I) = rt(A, I)$: the running time of an algorithm is considered as a proxy for the energy cost of running it.

The performance of an algorithm defined as the sum of the running times over all instances, where timed out runs are penalised, following the Penalised Average Running time (PAR) typically used in configuration scenarios.

Each methods we describe relies on a different amount and type of background knowledge about the set of instances. This knowledge is similar to the one used in the AS problem and thus readily accessible. We consider the following ways of specifying the background knowledge:

- *Sample-based*: for each instance I and algorithm A we have the running time $rt(A, I)$ of A on I.
- *Feature-based*: for each instance I we have a feature vector f_I.
- *Statistics-based*: for each instance I, we have a prior in the form of a probability distribution δ_I over $[0, T_{\text{cut}}]$, expressing that $\delta_I(t)$ is the probability that A_{ch} solves the instance I at time t. In practice, we obtain this prior by fitting a distribution to the running times of A.

Note that above, $A \neq A_{\text{ch}}$, *i.e.* the background knowledge does not contain information about the challenger algorithm. The implicit assumption is that running times of algorithms from $\mathcal{A}$ and feature vectors of the instances are both predictive of the running times of A_{ch}: for instance, if all algorithms in $\mathcal{A}$ solve an instance I very quickly, then so should A_{ch}. In other words, A_{ch} is expected to have similar behaviour as the algorithms in $\mathcal{A}$. Similarly, if two feature vectors f_I and $f_{I'}$ are close for two instances I, I', then their running times should be close. These assumptions are prominently made in running-time prediction, such as in Hutter et al. (2014b) and per-instance AAC (see *e.g.* Kerschke et al., 2019). In other words, the key insights and mathematical formalisation of this section are based on the background knowledge described above to evaluate the expected performance of A_{ch}.

5.3 Methods

Our goal is to define a *strategy* that sequentially chooses the instances on which to run our challenger A_{ch} and decides if the evidence so far gives sufficient confidence to stop the comparative evaluation. Algorithm 5.1 formalises this iterative process using a

Algorithm 5.1 PSEAS solving strategy

Input A_{inc}: the incumbent algorithm, A_{ch}: the challenger algorithm, C_{thres}: the target confidence threshold, $\mathcal{I}$: the training instances.
Output A_{inc} or A_{ch}: the best performing algorithm.

1: set $\mathcal{I}_{\text{torun}} = \mathcal{I}$ and $C_{\text{current}} = 0$
2: compute score(I) for all $I \in \mathcal{I}$
3: **while** C_{current} ¡ C_{thres} **do**
4: pick $I^* \in \text{argmax}_{I \in \mathcal{I}_{\text{torun}}}\ score(I)$ and remove I^* from $\mathcal{I}_{\text{torun}}$
5: evaluate $rt(A_{\text{ch}}, I^*)$
6: update C_{current}
7: update score(I) for $I \in \mathcal{I}_{\text{torun}}$
8: **end while**
9: **return** best performing algorithm from (A_{inc}, A_{ch})

score-based approach: each instance is assigned a score, which may be updated along the comparison to – intuitively – reflect the interest in running this instance. C_{current} is the current confidence and depends on A_{inc}, $\mathcal{I}_{\text{torun}}$ are the instances on which A_{ch} has not been run There are two main components in this algorithm: one for score computation (lines 2, 4, 7; see Section 5.3.2) and one for confidence (lines 1, 3, 6; see Section 5.3.1). The score enables choosing the best instance to run, whereas the confidence tells when to stop the comparison. These two components will be explained in more detail later.

Strategy evaluation

We consider two metrics for evaluating strategies: the *cost* and the *accuracy*.

We measure the computational effort (which we want to minimise) as the ratio of the total running time for instances in $\mathcal{I}_{\text{run}}$, the set of instances on which A_{ch} has been run by the strategy, over the total running time over all instances; this results in a number between 0 and 1. Note that the goal is not to minimise the *number* of instances A_{ch} is run on, but rather the total running time of A_{ch} on these instances. To evaluate our strategy, we determine this cost over many ordered pairs of algorithms $(A_{\text{inc}}, A_{\text{ch}})$ and consider the median. Formally, for a set of ordered pairs $\mathcal{P}$:

$$cost(\mathcal{P}) = median\left[\left(\frac{\sum_{I \in \mathcal{I} \setminus \mathcal{I}_{\text{torun}}} rt(A_{\text{ch}}, I)}{\sum_{I \in \mathcal{I}} rt(A_{\text{ch}}, I)}\right)_{(A_{\text{inc}}, A_{\text{ch}}) \in \mathcal{P}}\right],$$

where $\mathcal{I}_{\text{torun}}$ are the instances that have not been run by the strategy during its execution, as defined in Algorithm 5.1. We note that $cost(\mathcal{P})$ only depends on A_{ch}, since A_{inc} is assumed to have already been run.

We measure the *accuracy* of a strategy (which we want to maximise), as the ratio of correct guesses made by the strategy when deciding which algorithm from an ordered pair of algorithms $(A_{\text{inc}}, A_{\text{ch}})$ performs best. Formally, for a set of ordered pairs $\mathcal{P}$:

$$accuracy(\mathcal{P}) = \frac{\sum_{(A_{\text{inc}}, A_{\text{ch}}) \in \mathcal{P}} 1_{\{\hat{A}_{\text{best}} = A_{\text{best}}\}}}{|\mathcal{P}|},$$

where A_{best} is the true best performing algorithm in $(A_{\text{ch}}, A_{\text{inc}})$, and $\hat{A}_{\text{best}}$ is the best algorithm given by the strategy. Our definition of *accuracy* uses the mean, since the median over the results of the indicator function would produce too limited a range of results to be useful for comparing strategies.

We note that the choice made in line 4 of Algorithm 5.1 aims at optimising two goals. The *instance selection component* tries to minimise the computational effort by deciding on which instances to run A_{ch}, based on a score given to each instance. The *discrimination component* decides, based on the data gathered so far, whether the expected accuracy, or confidence, is high enough to stop the comparison.

5.3.1 The discrimination component

The discrimination component aims at estimating the accuracy of the current decision of which among A_{inc} and A_{ch} performs best. However, this measure can never be accessed, since the complete data is not available. Hence, we look at the expected accuracy, or *confidence*, as a proxy for accuracy. The confidence is computed differently for each discrimination method and is thus not comparable among them. It provides a measure of the current state of the strategy. When the confidence reaches a threshold C_{thres} (line 3 of Algorithm 5.1), the strategy stops and returns the algorithm evaluated as being the best.

Baseline: Subset method

As a baseline, we use a fixed-size subset of instances: we fix $\gamma \in [0; 1]$ and decide to stop when A_{ch} has been run on $\lfloor \gamma |\mathcal{I}| \rfloor$ instances. Note that this does not ensure a ratio of γ for the total running time, as the total running time of A_{ch} over all instances is not available and therefore cannot be used for discrimination. The confidence for this method is 0 until all instances of the subset have been executed; then the confidence is 1.

Wilcoxon test

There is a large body of literature on statistical tests, and many of them can be used in the context of racing algorithms (Birattari et al., 2002). For instance, the F-Race (Birattari, 2009) algorithm uses a Friedman two-way analysis of variance by ranks. However, this test concerns a family of candidates, while here, we are interested in an ordered pair of algorithms. When only two configurations remain, the F-Race algorithm switches to a *Wilcoxon matched-pairs signed-ranks test* (Conover, 1998), because it is more powerful and data-efficient than the Friedman test in that scenario (Siegel and Castellan Jr, 1988).

The test we want to apply should satisfy the following requirements: it should be nonparametric, and it should apply to paired data. Such a test would *not need any background knowledge*. We chose the Wilcoxon test because it satisfies our requirement while exploiting other properties of our data: data is measured on an interval scale, the differences (between running times) are symmetric, and the magnitudes of the differences between our paired data are exploited. This test assumes that running times are independent and the two samples are mutually independent. While it is not truly the case, we find that assuming independence is a good first approximation. This test is only based on observed data; it does not take into account the remaining instances. Through hypothesis testing, we can find out when there is enough evidence to stop, at which point the best algorithm is the one with the lowest mean running time. In this case, our confidence threshold C_{thres} is compared to the p-value of the alternative two-sided hypothesis. Let us note that other statistical tests, such as the Mann-Whitney U test, the permutation test, the Kolmogorov-Smirnov test, or the paired t-test, do not satisfy our assumptions.

The distribution-based discrimination method

This method requires statistics-based background knowledge. Let us consider the following random variable computing the difference in performances:

$$\Delta_{\text{tot}} = \underbrace{\sum_{J \in \mathcal{I}_{\text{run}}} rt(A_{\text{ch}}, J) - rt(A_{\text{inc}}, J)}_{\text{constant}} + \sum_{I \in \mathcal{I}_{\text{torun}}} \underbrace{rt(A_{\text{ch}}, I)}_{\text{random variable}} - \underbrace{rt(A_{\text{inc}}, I)}_{\text{constant}}.$$

We are interested in determining the sign of Δ_{tot}, meaning which of the two algorithms performs best. For a fixed confidence threshold $C_{\text{thres}} = 1 - \varepsilon$, we estimate $\mathbb{P}(\Delta_{\text{tot}} > 0)$ and stop if:

- $\mathbb{P}(\Delta_{\text{tot}} > 0) \geq 1 - \varepsilon$, in which case A_{ch} performs worse than A_{inc},
- or $\mathbb{P}(\Delta_{\text{tot}} > 0) \leq \varepsilon$, meaning $\mathbb{P}(\Delta_{\text{tot}} \leq 0) \geq 1 - \varepsilon$, *i.e.* A_{ch} performs better than A_{inc}.

The confidence is $\mathbb{P}(\Delta_{\text{tot}} > 0)$ for the former case and $1 - \mathbb{P}(\Delta_{\text{tot}} > 0)$ for the latter. Looking at the definition of the random variable Δ_{tot}, its probability law can be described using translations and convolutions of the distributions $(\delta_I)_{I \in \mathcal{I}_{\text{torun}}}$. In practice, many natural classes of distributions (such as Gaussian and Cauchy distributions) are closed under translations and convolutions, so $\mathbb{P}(\Delta_{\text{tot}} > 0)$ can be effectively computed or approximated.

Because running times are positive and algorithms are stopped when they reach the cutoff time T_{cut}, the running times are bounded. A distribution matching this behaviour would be a truncated distribution. Still, most are not closed under convolution, which we have stated above as a necessary property, so they cannot be used directly. Nevertheless, the sum of the bounds on individual running times can be used as bounds for Δ_{tot}, which we can model as a truncated distribution. For heavy-tailed distributions, such as the Cauchy distribution, the confidence is higher with a truncated distribution than without, as impossible cases are not taken into account, enabling to stop earlier.

5.3.2 The instance selection component

With the aim of minimising the overall computational effort, our algorithm iteratively chooses the most relevant instance, according to a score (lines 2 and 7 in Algorithm 5.1). Instances with the highest score are expected to be the most relevant ones (*i.e.* intuitively giving the most information at the lowest cost).

Baseline: Uniform random sampling

As a baseline, we use a random sampling approach. In our algorithm, this corresponds to giving the same score to all instances, and thus to a uniform random choice at each iteration.

The discrimination-based selection method

This sample-based method is inspired by Gent et al. (2014); they developed it as a method to find optimal parameters for instances in an instance selection approach for

automated constraint model selection. The intuition is to choose the most discriminating instances first. Let $\rho > 1$ be a constant; an algorithm A is ρ-*dominated* on an instance I if there exists another algorithm A' such that $rt(A', I) \leq \rho \cdot rt(A, I)$. The *discrimination quality* of an instance I, denoted $G(I)$, is the fraction of algorithms that are ρ-dominated on this instance. Using this measure as-is would not take into account our goal of minimising the running time, so we divide the discrimination quality by the mean running time of the instance. The obtained score only needs to be computed once:

$$score(I) = \frac{G(I)}{mean[(rt(A, I))_{A \in \mathcal{A}}]}.$$

The variance-based selection method

This statistics-based method uses the intuition that the most interesting instances are the ones most likely to have very different running times for A_{inc} and A_{ch}. For each instance I we have a prior δ_I, which is the running time distribution of A_{ch}. We want to choose an instance with the highest variance $\text{argmax}_{I \in \mathcal{I}_{\text{torun}}} \mathbb{V}(\delta_I)$. As for the discrimination-based selection method, since we want to minimise the running time, we divide by the mean running time of the instance. The obtained score only needs to be computed once:

$$score(I) = \frac{\mathbb{V}(\delta_I)}{\mathbb{E}[\delta_I]}.$$

The information-based selection method

This statistics-based method is based on a similar intuition to the previous method. We are interested in instances from which we gain as much information as possible; the variance is only one (natural) indicator of this information. Following this approach, we can also estimate the information gained from a specific instance. The concrete information we are after is given by the discrete random variable stating that A_{ch} is better than A_{inc}, formally defined as $sign(\Delta_{\text{tot}})$. Let Δ_I be the random variable defined as $\Delta_I := rt(A_{\text{ch}}, I) - rt(A_{\text{inc}}, I)$, such that

$$\Delta_{\text{tot}} = \sum_{K \in \mathcal{I}} \Delta_K = \sum_{J \in \mathcal{I}_{\text{run}}} \Delta_J + \sum_{I \in \mathcal{I}_{\text{torun}}} \Delta_I.$$

We compute the expected information brought by $I \in \mathcal{I}_{\text{torun}}$; hence the information gain is defined as follows for I:

$$IG\left[\Delta_I\right] := \mathbb{E}_{rt(A_{\text{ch}},I)\sim\delta_I}[D_{\text{KL}}(Q_{+I} \mid\mid Q)] \text{ with}$$
$$Q = P\left(sign(\Delta_{\text{tot}})|\Delta_J = r_J, J \in \mathcal{I}_{\text{run}}\right)$$
$$Q_{+I} = P\left(sign(\Delta_{\text{tot}})|\Delta_J = r_J, J \in \mathcal{I}_{\text{run}}, \Delta_I = r_I\right),$$

where D_{KL} is the Kullback–Leibler divergence, δ_I is the distribution of running time on instance I, and r_J are the realisations of the Δ_J since the difference for the instances in $\mathcal{I}_{\text{run}}$ is known.

As for the previous method, to balance information and running time, we divide by the expected running time, and therefore use the following score function, which we update at each iteration:

$$score(I) = \frac{IG\left[\Delta_I\right]}{\mathbb{E}[\delta_I]}.$$

The feature-based selection method

In this feature-based and statistics-based method, we assume that for each instance I, we have a feature vector $f_I \in \mathbb{R}^n$ in some dimension n. The implicit assumption is that features are predictive of the running times of A_{ch}. We proceed in two steps:

- Constructing a distance metric $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_{\geq 0}$, such that if $d(f, f')$ is small, then two instances with features f and f' have similar running times.
- Assigning a score to each instance $I \in \mathcal{I}_{\text{torun}}$.

Constructing a distance metric. The objective is to define a distance predictive of the running times; to this end, we introduce a weight for instance features, represented by a weight vector $\theta \in \mathbb{R}^n$. Let us consider distances of the form:

$$d_\theta(f_I, f_J) = \sqrt{\sum_{x=1}^{n} \left(\theta(x) \cdot (f_I(x) - f_J(x))\right)^2}.$$

Intuitively, for a feature x, the parameter $\theta(x)$ determines the importance of x in predicting the running times. Let us write m_I for the median time over all algorithms on instance I. We optimise over θ by considering:

$$\theta^* \in \operatorname*{argmin}_{\theta\in\mathbb{R}^n} \sum_{I,J\in\mathcal{I}} \left(d_\theta(f_I, f_J)^2 - |m_I - m_J|\right)^2;$$

i.e., d_{θ^*} is the best distance in this family for predicting differences in median running

time. The parameter vector θ^* is the solution of a non-negative ordinary least square optimisation problem and can therefore be computed efficiently (Lawson and Hanson, 1995). Note that the space complexity is quadratic in the number of instances and linear in the feature space dimension.

Assigning a score. Given a distance metric d, we now define a score for a given problem instance. Here, it is convenient to minimise rather than maximise the following quantity with respect to d:

$$S(I) = \sum_{J \notin \mathcal{I}_{\text{torun}}} \frac{rt(A_{\text{ch}}, J)}{d(f_I, f_J)} + \sum_{J \in \mathcal{I}_{\text{torun}}} \frac{\mathbb{E}[\delta_J]}{d(f_I, f_J)} \quad \text{and} \quad score(I) = \frac{1}{S(I)}.$$

The score is updated at each iteration. In all previous methods, the score of an instance I only uses the information on I; the strength of this method is to gather and weight information over all instances. Indeed, the score of I is a weighted average over all running time predictions, meaning $\mathbb{E}[\delta_J]$ when $J \in \mathcal{I}_{\text{torun}}$ and $rt(A_{\text{ch}}, J)$ otherwise, and the prediction for J contributes to the prediction of I up to the multiplicative factor $\frac{1}{d(f_I, f_J)}$.

5.4 Setup of experiments

To empirically evaluate our approaches, we implemented it with Python, using Numpy and Scipy, and ran them on all ordered pairs of algorithms from well-known benchmark scenarios.

5.4.1 Datasets

We use ASlib (Bischl et al., 2016), a benchmark library for AS that contains datasets from competitions for various challenging problems, including Boolean satisfiability and constraint programming. It provides very relevant data on which our strategies can be tested, because such problems are the typical use-case scenario that we envisioned.

From ASlib, we use three datasets: the CSP MiniZinc 2016 dataset, which comprises performance data from the 2016 MiniZinc Challenge ('Free Search' Category) (Lindauer et al., 2017; Stuckey et al., 2014); the BNSL 2016 dataset (Malone et al., 2018) from Bayesian Network structure learning; the SAT18 dataset, which consists of performance data from the EXP track of the 2018 SAT Competition (Heule et al., 2019); and, to account for more recent advances in SAT, we created the SAT20 dataset from the results of the main track of the 2020 SAT Competition (Balyo et al., 2020). Those

Table 5.1: Characteristics of the used datasets

Dataset	CSP MiniZinc	BNSL	SAT 18	SAT 20
Algorithms	20	8	37	67
Instances	100	1178	353	400
Features	95	86	54	108
Mean difficulty	59.74	9.363	2458	78.88
Median difficulty	3.28	1.15	9.65	5.51
Top-3 mean difficulty	24.7	77.5	47.7	49.9

datasets were chosen to cover a broad range of prominent problems and instance sets.

For our feature-based approaches, we decided to replace missing features with the mean value, as done by Hutter et al. (2014b). Hence, no information can be extracted from such instances.

To get a sense of how difficult it is to discriminate between the algorithms from each dataset, we introduce a measure of difficulty based on how different the algorithms behave on our set of instances. We propose to use the following ratio:

$$\mathcal{D}_{\text{discr}}(A_{\text{inc}}, A_{\text{ch}}) = \frac{\sum_{I \in \mathcal{I}} median[(rt(A, I))_{A \in \mathcal{A}}]}{\left|\sum_{I \in \mathcal{I}} rt(A_{\text{ch}}, I) - rt(A_{\text{inc}}, I)\right|}.$$

This measure has been chosen because it grows when the two algorithms have similar performance, and it is invariant under scaling, so that the difficulty remains the same if running times are multiplied by a constant factor. It is also symmetric: exchanging A_{inc} and A_{ch} leads to the same result.

In Table 5.1, we report the characteristics of our datasets as well as their mean difficulty, median difficulty over all pairs and mean difficulty of the subset of the best 3 algorithms. Based on this measure, we expect it to be easy to discriminate between algorithms from BNSL, while SAT18 should provide a bigger challenge. The large discrepancy between the mean and median value, seen for SAT18 in particular, is caused by small groups of algorithms with very close performances. Pairs of algorithms from those groups usually have very high difficulty, reaching up to a million for SAT18, which affects the mean.

5.4.2 Implementation details

Our implementation is available on GitHub[2]. To estimate the parameter of running time distributions, we use maximum likelihood estimation, and we use a Cauchy distribution for the distribution-based discrimination method, as motivated in Section 5.4.3. For the timeout correction, the seed was set to 0.

For the random instance selection method, the seed was also set to 0. The parameter ρ for the discrimination-based selection method was set to 1.2. For the information-based method, we use the expression of Δ_{tot} defined for the distribution-based method, and to compute the expected value, which is an integral, we use Simpson's rule. For the Wilcoxon discrimination method, Conover (1998) recommends at least 20 samples; however, this would represent up to 20% of our instances for the CSP Minizinc dataset. Thus, we decided to follow irace (López-Ibáñez et al., 2016), which requires 5 samples in a context similar to ours. We found no significant performance change between different methods for managing zero differences, when paired data from both populations is equal, as such, we report the performance using Pratt's method (Pratt, 1959).

5.4.3 Estimation of the running time distribution

Our approach relies heavily on our ability to estimate the distribution of running times of algorithms on the instances. This distribution is used in 3 out of the 5 instance selection methods and one of our 3 discrimination methods. As such, the choice of the distribution could significantly impact the performance of those strategies. Fitting a distribution to our data requires us to decide how to handle the cutoff time and which distribution to use.

We note that when predicting a running time, a log transformation is typically used (Hurley and O'Sullivan, 2015; Hutter et al., 2014b). This transformation allows better performance for predicting running times, because running times distributions tend to be heavy-tailed as shown in the work of Gomes et al. (2000). Since in our case we are mostly interested in predicting the mean or the sum over instances, we do not apply this log transformation.

Handling censored running times

As explained in the problem definition, after a given cutoff time T_{cut}, the given algorithm is stopped. Running times are thus right-censored, which limits our ability to

[2] `github.com/Theomat/MPSEAS`

estimate the true distribution.

Our method for handling time-outs is based on the one proposed by Hutter et al. (2011a), which itself is based on a prior work from Schmee and Hahn (1979). The resulting algorithm is Algorithm 5.2 for instance I, with parameters $M \in \mathbb{N}$ and $t_{\max} \in \mathbb{R}_+$.

Algorithm 5.2 Correcting timeouts for a sample $(t_{I,A})_{A \in \mathcal{A}}$

```
fit Distribution on (t_{I,A})_{A∈A} without the timeouts
set N to the number of timeouts in (t_{I,A})_{A∈A} and n to 0
while not converged do
    set S to M · N + n samples from Distribution then increment n
    for k = 1 to N do
        set q_k to quantile k/(N+1) of S
        replace timeout k with min(q_k, t_max) in (t_{I,A})_{A∈A}
    end for
    fit Distribution on (t_{I,A})_{A∈A}
end while
return Distribution and (t_{I,A})_{A∈A}
```

There is a slight difference from the original algorithm in the use of a loop counter n to increment at each iteration the number of samples used to enable convergence when there is a majority of timeouts on an instance. The parameter M enables reducing the sampling variance; it is most important on instances with many timeouts. The parameter $t_{\max}$ prevents overly large variations of the samples. There are two steps in this algorithm: first, we estimate the parameters of the distribution, and second, we replace the timeouts in the sample. They are repeated until convergence, when the estimated parameters of the distribution are stable. We decided to stop when the squared difference between the parameters between two iterations is less than or equal to 1. Schmee and Hahn (1979) use the mean instead of the quantiles of a sample; however, heavy-tailed distributions such as the Cauchy distribution have an undefined mean. We chose to use the sampling approach used by Hutter et al. (2011a), which enabled them to translate the uncertainty and improve the likelihood for their random forest models.

Choosing a distribution

What is the distribution satisfying the imposed constraints that gives the best performance?

In practice, since only a set of running times is provided, the distribution parameters must be estimated. We explained how the parameters were estimated in practice

Table 5.2: Median log likelihood of Maximum Likelihood Estimation for Levy and Cauchy distributions over the instances of each dataset. The highest likelihood for each dataset is shown in boldface.

	CSP MiniZinc	BNSL	SAT 18	SAT 20
Levy	-129.6	**-58.08**	-299.7	-573.5
Cauchy	**-107.5**	-62.88	**-183.8**	**-364.9**

in Section 5.4, where here, we explain our choice of distribution. This choice can be motivated by choosing the best candidate distribution that has the lowest error on the set of all instances.

Since many running time distributions are heavy-tailed, we tested two heavy-tailed distributions on our four datasets. We report in Table 5.2 the median log likelihood for each distribution; the parameters of these distributions were estimated using maximum likelihood estimation. The Cauchy distribution provides a clear advantage over the Levy distribution. The only case in which the Levy distribution yields a higher likelihood shows a much smaller difference between the two distributions.

5.5 Experiments

We designed and conducted extensive experiments in order to answer our research question RQ6, which we divided into three parts as follows:

RQ 6.1. How much can our strategies reduce the CPU time required for evaluating a new algorithm?

RQ 6.2. How do the selection methods affect the accuracy of the strategies?

RQ 6.3. How well can our strategies discriminate between top-ranking algorithms?

A run consists of selecting an ordered pair $(A_{\text{ch}}, A_{\text{inc}})$ and running the strategy. On each run, all strategies have to compare the same A_{ch} and A_{inc}. In all of our experiments, we ran all of our strategies on each ordered pair of a given dataset.

General Performance Comparison

To answer question 6.1, we plotted our strategies in Figure 5.1, with a target confidence threshold $C_{\text{thres}} = 0.95$ (see Algorithm 5.1). For each of them, the y-axis shows accuracy (in percent) and the x-axis the median time used over all ordered pairs of

algorithms, as defined in Section 5.3. As this corresponds to a multi-objective setup, we highlight the Pareto fronts induced by our results. This does not imply that we can produce a strategy that follows the Pareto front between points; however, by changing the confidence threshold C_{thres}, we can obtain local curves around the performance of each strategy. Note that while we show the performance of our strategies without applying a penalty for timeouts, using penalty coefficients from $[|1; 10|]$ did not affect our findings.

On all datasets, we observe that our random baseline (random sampling a subset of 20% of the instances) shows rather strong performance, with 89% to 100% accuracy for about 20% of the running time. Further investigation (see Section 5.5) shows that the accuracy of the random baseline increases steeply as we add more instances, until reaching about 20% of the instances, after which the increase in accuracy is substantially slower. Thus, increasing the amount of instances does not lead to significantly higher accuracy. Moreover, more than half of the time, this strategy takes 17 to 22% of the running time, which means that the running times of the instances follow a distribution such that there are as many easy instances as hard ones.

We expect that this behaviour is linked to the nature of the competition datasets we are using; instances were gathered by experts to be representative and to show various levels of difficulty. We also note that the BNSL dataset, which is the one that gives the largest advantage to the random baseline, contains very few instances that are not solved within the cutoff time (about 10% of the instances, against about 50% for the other datasets). Choosing unsolved instances incurs a high penalty, because they offer no new information for deciding between the two algorithms while using up a large amount of running time.

On all datasets, we observe that the Wilcoxon method is superior and achieved the desired accuracy in less than 15% of the time; it thus represents the left-hand side of our Pareto front. The subset baseline uses consistently around 20% of the time but hardly reaches 90% accuracy on the hardest dataset; it contributes to the Pareto front only for BNSL. The distribution-based method tends to be more conservative and run longer but often reaches higher accuracies than the desired C_{thres} and thus marks the right-hand side of our Pareto front on our two SAT scenarios; however, it performs very poorly on BNSL, which is the scenario with the least background knowledge due to its low number of algorithms.

The instance selection methods do not show such a clear pattern. We notice, however, that the information-based method lies near or on the Pareto front when combined with Wilcoxon. In contrast, the discrimination-based and variance-based

methods show strong performance when used in combination with distribution-based discrimination.

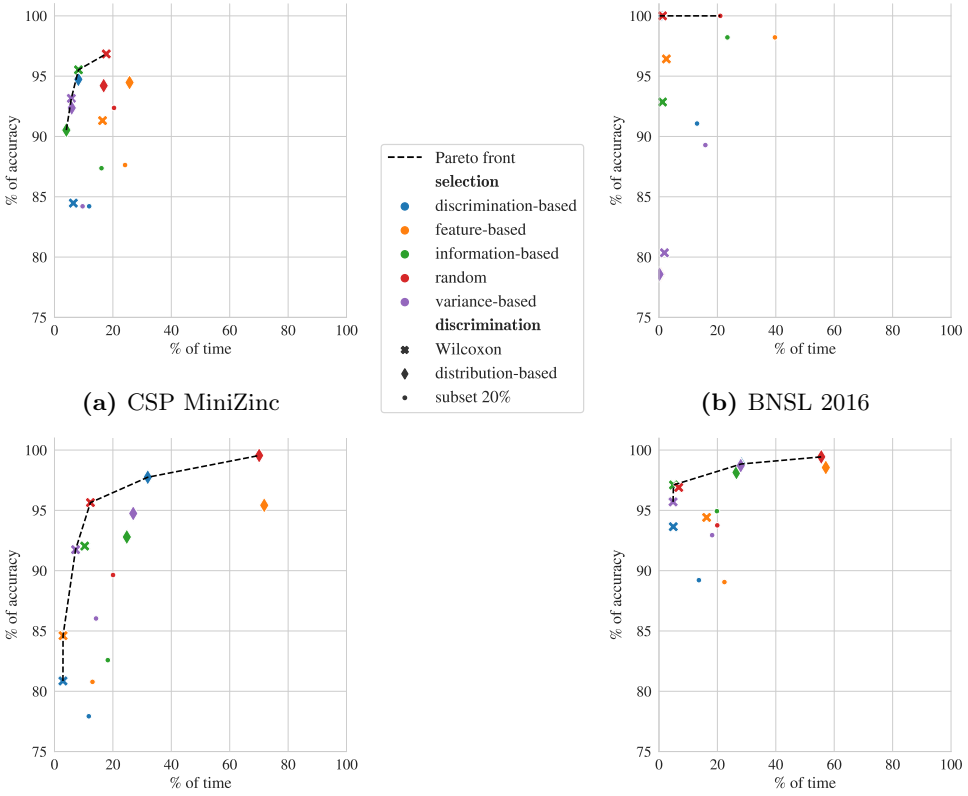
The evaluated strategies achieved up to 95.5% accuracy using 8.21% of the time on the MiniZinc dataset, 95.6% accuracy using 12.3% of the time on SAT18, and 97.1% accuracy using 4.96% of the time on SAT20. For the BNSL dataset, we observed a surprising 100% accuracy while using only 0.0001% of the time using the discrimination-based selection with Wilcoxon discrimination that is hidden behind on Figure 5.1b, running a median number of 6 instances. The observed performance of our strategies is consistent with the ranking of the datasets according to our difficulty metric (see Table 5.1 in Section 5.4) for the distribution-based methods, but not for Wilcoxon, where SAT20 should have been harder than MiniZinc. Overall, in the worst case, we managed to save 87.6% of CPU time while being 95.6% accurate, and in the best case, we saved 95.0% of CPU time while being 97.1% accurate.

Accuracy over time

To answer question 6.2, we ran our strategies without a stopping criterion, measuring regularly the percentage of accuracy and the time spent running A_{ch}. Figure 5.2 shows the accuracy (in percent) of the Wilcoxon and distribution-based discrimination methods on all our datasets.

Unlike Figure 5.1, which did not show any clear pattern regarding the instance selection methods, this analysis reveals two groups of methods. On all but the BNSL dataset, the information-based, variance-based and discrimination-based selection methods lead to a very high accuracy after 55 to 60% running time. This is consistent with the ratio of instances for which most algorithms time out, thus providing little discriminatory power. The feature-based method shows the lowest accuracy, and the random sampling comes in second to last after 40% of the running time.

The BNSL dataset is different, due to a low number of timeouts and large performance differences between the algorithms. In this case, randomly sampling instances offers high accuracy after a few instances. None of the selection methods offers a clear advantage, because all instances provide evidence towards the algorithm performing best. This suggests that the random method is a good choice for easy datasets, while more complex datasets containing instances that cannot be solved within the given cutoff time benefit from more sophisticated selection methods to save running time.



(a) CSP MiniZinc

(b) BNSL 2016

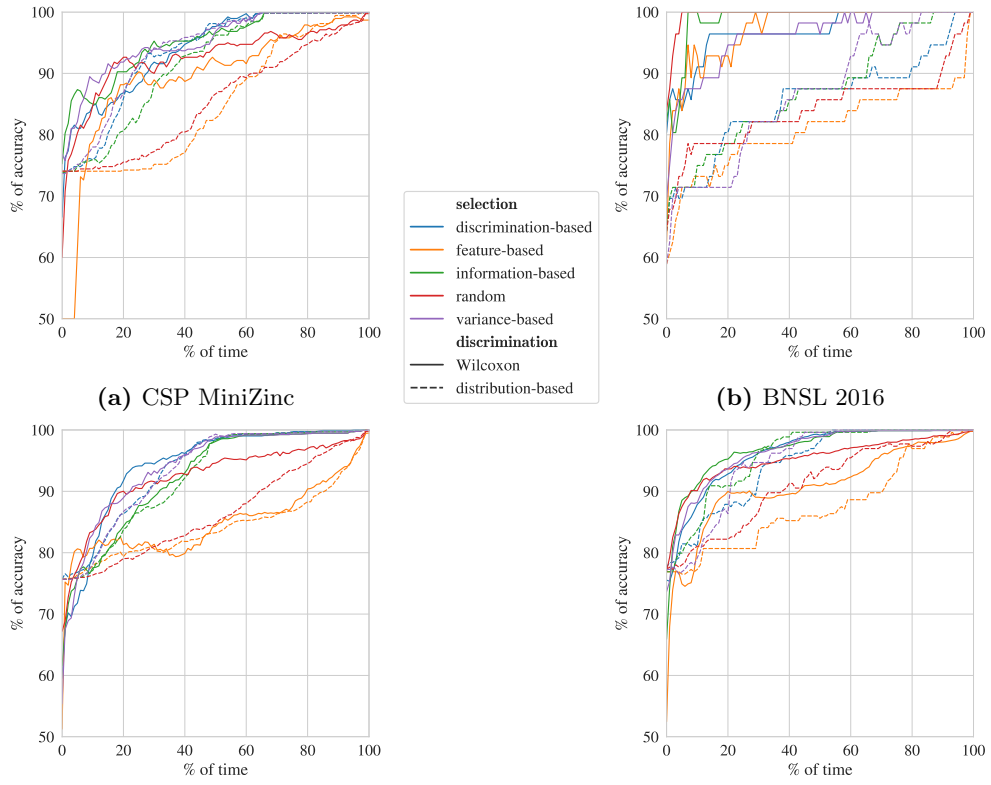
(c) SAT 18

(d) SAT 20

Figure 5.1: Accuracy over median running time. *y-axis*: percentage over all ordered pairs of algorithms in the dataset. *x-axis*: the time spent running the new algorithm.

Top ranking

To answer our last question 6.3, we decided to keep the top 10 algorithms according to their performance on the SAT20 dataset and use our strategies on this new dataset; this reflects the fact that often, the primary interest is in discriminating between top-ranking algorithms, be it to compare a new algorithm to the state of the art or to distinguish between the winners of a competition. As per our difficulty measure introduced in Section 5.4, the mean difficulty of the dataset thus obtained is 163, and the median is 22, which is higher than for any of our other datasets. Furthermore, the number of algorithms is reduced, which should reduce the performance of our methods based on prior knowledge. We report the results in Figure 5.3 analogous to what was



(a) CSP MiniZinc

(b) BNSL 2016

(c) SAT 18

(d) SAT 20

Figure 5.2: Accuracy over running time used. *y-axis*: percentage over all ordered pairs of algorithms in the dataset. *x-axis*: time spent running the new algorithm.

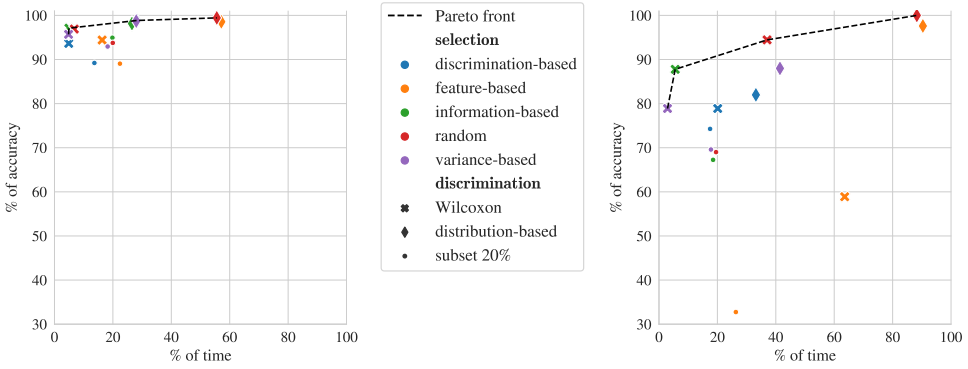
done in Section 5.5; for comparative purposes, we also plot the performances on the full SAT20 dataset. The performance of the subset method decreases by more than 10% in accuracy. The distribution-based discrimination method requires more time for this subset, and the discrimination-based selection method drops out of the Pareto front. Because they require prior knowledge, these methods encounter difficulties with this more challenging dataset. The Wilcoxon method is least affected, as it does not depend on prior knowledge; consequently, 3 out of the 4 strategies on the Pareto front use this method. The selection methods in combination with the Wilcoxon test are affected in different ways. The information-based and variance-based approaches yielded a quick but less accurate decision, while random sampling resulted in a slower decision, achieving 94.4% accuracy for 37.0% of running time.

In this experiment, which compares algorithms with similarly good performance,

the information-based method using the Wilcoxon test suffers less than the other strategies, both in terms of cost and accuracy. All other methods lead to either high cost or poor accuracy.

5.6 Conclusions and future work

In this chapter, we have investigated methods for reducing the computational effort required for comparing the performance of two automated reasoning algorithms, while gathering sufficient statistical evidence to correctly identify the solver that performs better on a given set of problem instances. We defined the per-set efficient algorithm selection problem (PSEAS) in Section 5.2. We studied the case in which the performance of a given algorithm is evaluated based on its running time on a set of instances. We described a set of strategies in Section 5.3, inspired by related problems from the literature and by novel considerations, and tested these on four datasets covering SAT, CSP and BNSL. Our experimental evaluation in Section 5.5 shows that on these datasets, some of our strategies consistently return the correct answer with at least 95% accuracy, while using less than 15% of the CPU time it would take to run the full comparison. In particular, using a Wilcoxon test to decide when to stop, while deciding the next instance to run based on the expected amount of information it can provide, is consistently near or among the best-performing approaches.



(a) SAT 20, full Dataset, 67 algorithms

(b) SAT 20, top-10 algorithms

Figure 5.3: Accuracy over running time used for the full and reduced SAT20 datasets. *y-axis*: percentage over all ordered algorithms' pairs in the dataset. *x-axis*: time spent running the new algorithm.

A finer-grained analysis of our instance selection methods (see Section 5.5) provides additional insights. We found that deciding on which instance to run based on its discrimination power, following the work of Gent et al. (2014), or simply on a notion of running time variance, has the potential to reduce the time required to make a decision when a significant fraction of the given instances are difficult.

Furthermore, we tested our methods on a smaller but more challenging set of algorithms, comprising the top 10 algorithms from the SAT20 competition. While the overall performance is lower than on the full dataset, the Wilcoxon method still reaches an accuracy of 94.4% in 37.0% of the overall running time. Overall, we found that for easy datasets, which discriminate between very different algorithms on instances that can be solved quickly, random sampling offers good performance. However, when facing hard instances or comparing well-performing algorithms, it is beneficial to use more sophisticated methods.

While the scope of our work presented here has been limited to comparing two algorithms, one interesting area of future work would be to extend it to many algorithms, to devise principled mechanisms for running competitions and other large-scale performance comparisons more efficiently.

Moreover, those methods can in theory be applied to comparing pairs of configurations of a single algorithm. This is the avenue studied in the following chapter.