Universiteit Leiden
The Netherlands

Opleiding Informatica

Self-Monitoring Automated Algorithm Configuration

Sam C. Vermeulen

Supervisors:
Koen van der Blom & Holger H. Hoos

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

05/12/2022

Abstract

Boolean Satisfiability Problems, Traveling Salesperson Problems and Mixed Integer Programming Problems are all examples of NP-complete problems. NP-complete problems have many practical applications, but instances of these problems can be quite complex and large, so they are often practically impossible to solve by hand. For that reason we have to use solving algorithms to find answers to them. Some of these solvers can be optimized by carefully choosing their parameter settings so they solve a given set of problems quicker. This is called Algorithm Configuration (AC). However, the performance of one configuration on a given problem set does not guarantee equal or better performance on new, unseen instances. Therefore, blindly using a current configuration on new problem instances might lead to slowdowns. In this thesis we investigate a way to increase the benefits of AC by way of performance monitoring. We use a SAT solver in this thesis to illustrate the principle of our monitoring algorithm. We first configure the SAT solver Spear on a subset of the SWV SAT instance set. We then build a random forest model to predict algorithm running time based on algorithm hyperparameter configuration and problem instance features. Using this model, we establish a performance threshold based on the predicted default algorithm performance that our configured algorithm must outperform. If the performance of the configured algorithm becomes too low compared to the threshold, we notify the user to reconfigure the solving algorithm. Using this predictive algorithm takes very little time and can prevent potentially long solving times on new problem instances by reconfiguring the solver. We configure a solving algorithm for optimal running time so we have a configured algorithm to compare to the default. This configuration process also gives us the running times of many different algorithm configurations and we use these running times for training a random forest. This is a different approach than we see in previous work. Unfortunately the running times in this dataset are biased downwards due to the nature of the AC process and although our results show we can predict whether the default or configured algorithm is faster around 75% of the time, this is less useful in practice than it might seem due to the bias in our dataset. For instance, our running time predictions rarely exceed 10 seconds despite many actual running times being higher than this. We do, however, propose solutions to this problem, along with possible future research. Thus we conclude our monitoring approach is a viable way to increase the benefit and reliability of algorithm configuration.

Contents

1 Introduction **4**

2 Definitions **5**
- 2.1 SAT . . . 5
- 2.2 Algorithm Configuration . . . 6
- 2.3 Running time Prediction . . . 7

3 Related Work **7**
- 3.1 Spear and SWV . . . 8
- 3.2 Sparkle and SMAC . . . 8
- 3.3 Running Time Prediction . . . 8

4 Algorithm Configuration **8**
- 4.1 Experimental Setup . . . 9
- 4.2 Experiment A: Algorithm Configuration . . . 10

5 Running Time Prediction **12**
- 5.1 Methods . . . 13
- 5.2 Experimental Setup . . . 15
- 5.3 Experiment B: Bootstrapping and Censored Value Approximation . . . 17
- 5.4 Experiment C: Excluding Running times Below a Boundary . . . 18
- 5.5 Experiment D: Equal Running Time Buckets . . . 19

6 Performance Monitoring **21**
- 6.1 Experimental Setup . . . 21
- 6.2 Experiment E: Performance Monitoring (0%, 5%, 20%) . . . 22
- 6.3 Experiment F: Performance Monitoring (-5%) . . . 25

7 Conclusion **26**

8 Future Work **28**

References **29**

A split_files.py **29**

B File Server Batch Files **30**

C Using the Random Forest and Monitoring Algorithm **32**

1 Introduction

NP-Complete problems are decision problems in computer science that are very hard to solve, but easy to verify a solution of. There are many examples, such as finding the metric dimension of a graph or finding the longest common sequence in a text document (see, e.g., [Hartmanis, 1982]). Practically relevant instances of these problems can be quite complex and very large. Therefore they can be hard, if not impossible to solve by hand. For that reason we have to use solving algorithms to try to find solutions to these problems. The process of Algorithm Configuration (AC) can help with this. AC is the process of adjusting the parameters of an algorithm for some purpose, like finding higher quality solutions or finding any solution as quickly as possible.
Given a Boolean formula in conjunctive normal form (CNF), the Boolean Satisfiability (SAT) Problem is finding whether it is possible to satisfy the entire formula through some assignment of true and false values. Instances of these problems can be very large and complex, so finding these solutions is often practically impossible by hand. For that reason SAT solving algorithms are often used. SAT problems are useful because they can be used for planning in the field of Artificial Intelligence, finding faulty electrical circuits automatically and haplotyping in Bioinformatics [Marques-Silva, 2008], but also because many other NP-Complete problems can be rewritten to the form of SAT and therefore also be solved using SAT solving algorithms. For these reasons we chose to focus on SAT problems during this thesis, although others like Traveling Salesperson Problems (TSP) and Mixed Integer Programming (MIP) Problems could have been studied during this thesis as well.
Configuring an algorithm in our case means our SAT solver will be made to run as quickly as possible on the given training dataset. However, a configuration of an algorithm is not guaranteed to work as well on new, previously unseen problem instances. This means that when we encounter new problem instances, our algorithm could start performing comparatively much worse than on the set it was trained on. This can be detrimental for performance and as such, when an algorithm starts underperforming, it would need to be reconfigured to maintain adequate performance.
The problem with reconfiguring after running the algorithm first is that running the algorithm can take a long time. For this reason it would be better if we could predict how well our configuration of the solver will perform on the new problem instances and take action based on those predictions. As shown by Hutter et al. 2014, such algorithm running times can be predicted using several methods, among which is the Random Forest (RF) prediction model, which we use in this thesis. Using these predictions we can then warn the user if performance becomes too low based on some established threshold and the user can take action based on these warnings. Fitting the RF and using it to predict algorithm running time is much cheaper than actually running the algorithm on the instances and observing the running time. Therefore it is in our best interest to use these predictions to be able to continually monitor whether or not a certain solver configuration will still perform well on coming problem instances.
In order to test our approach of monitoring the expected performance of a solving algorithm, we need several components such as a solver, a problem instance set and the problem instance features of that set. We use Spear [Babić and Hutter, 2008] as our SAT solver and the SWV instance set [Hutter et al., 2014] since this combination has been used in previous running time prediction literature, which allows us to compare our results. We extract the problem instance features with

the feature extractor used in SATzilla [Xu et al., 2008] [1]. We configure the Spear solver so we have a default and an optimized configuration to compare against each other. We configure Spear using SMAC [Hutter et al., 2011] through the Sparkle platform [van der Blom et al., 2022], which also generates a set of running times of the different configurations SMAC has tried during configuration. We use these running times to build our random forest predictive model. We verify the accuracy of our RF with a different subset of the same data the RF was trained on. The RF is then used in our monitoring algorithm by predicting running times of both the default and the configured algorithm and deciding from that which one is faster. We also establish a performance threshold based on the default running time, which means the configured algorithm must be $X\%$ faster than the default algorithm, since the user might want the configured algorithm to be significantly faster than the default. The monitoring program would warn the user if the configured algorithm performance would be worse than the established threshold. We verify these predictions by running these two configurations of Spear and observing the true running times and comparing them to our predictions.
To explain this process in detail, we first go over useful terms in Section 2. Then, we will name what related work we have made use of during this project and how it relates to our experiments in Section 3. In Section 4, we explain the configuration process of the Spear solver and what results it generates. Using the results from the configuration process, we explain how we predict running times in Section 5. Then using these predictions, we explain our performance monitoring algorithm in Section 6. Then we conclude with what results we have found during this thesis in Section 7 and we discuss possible future work in Section 8.

2 Definitions

In this section we introduce various definitions that we will use throughout this thesis.

2.1 SAT

Since we apply our methods to SAT solving, we briefly explain the basic definitions.

Boolean formulas are formulas that consist of variables that can be assigned either true or false. These variables are also called 'literals.' The three basic Boolean operations are conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$). These can also be described as the logical 'AND,' 'OR' and 'NOT' operators respectively. A Boolean formula is **satisfiable** if the entire formula can be made true by some assignment of its constituent variables. Otherwise it is **unsatisfiable.** For example:

$$(a \vee \neg b) \tag{1}$$

This formula can be rewritten as "a or not b" and it is satisfiable since we can assign either $a = true$ or $b = false$ to satisfy the formula. In CNF (Conjunctive Normal Form), a 'clause' is defined as either a single literal, an empty clause or a disjunction ($\vee$) of two or more literals. All clauses in CNF are connected by conjunction ($\wedge$).

$$(a) \wedge (\neg a) \tag{2}$$

[1] Feature extractor: bitbucket.org/sparkle-ai/sparkle/src/development/.../description_SAT-features-

This formula contains two clauses which can be rewritten as "a and not a" and it is unsatisfiable since a cannot be both true and false at the same time.

Consider a Boolean formula in conjunctive normal form or CNF. A **Boolean satisfiability problem** or **SAT problem** is finding whether this formula can be satisfied through some assignment of true and false values. If this is the case, the problem is considered satisfiable. Each distinct formula forms an **instance** of the SAT problem. For example, take the following SAT problem instance:

$$(a \vee b) \wedge (\neg a) \wedge (\neg b) \tag{3}$$

This formula contains three clauses, connected with $\wedge$ signs. While all three clauses of this formula are satisfiable on their own, namely through $a = true$ or $b = true$ for the first clause and $a = false$ and $b = false$ for the second and third clause respectively, either the literal a or b would need to be both true and false to satisfy the problem, which is impossible. Therefore this SAT problem instance is unsatisfiable.

SAT Solvers are algorithms that can be used to automatically find solutions to SAT problems. Many solvers exist, such as MiniSAT [Eén and Sörensson, 2004], CSCCSat [Luo et al., 2016] and Spear [Babić and Hutter, 2008], each with a different approach to SAT solving. We use Spear in this thesis. Some SAT solvers have configurable parameters, which means they can be configured for purposes like faster running times.

Problem-specific instance features are computable values that say something about a problem instance, for example problem hardness and structure information in the case of SAT problems. Some examples of instance features used during this thesis are the number of variables, the number of clauses and the number of unary, binary and ternary clauses in the problem instance. This information allows us to distinguish between different problem instances and can, for example, be used to make better predictions for a promising configurations of an algorithm during algorithm configuration. These features can be extracted from problem instances using a **feature extractor**.

2.2 Algorithm Configuration

We configure a solving algorithm to collect runtime data and to find an optimized configuration to compare to the default. Here we explain the relevant definitions.

Algorithm parameters are settings of an algorithm that control how it searches for solutions. These settings range from smaller things like weights for decision making to large things like what type of variables to prioritize during searching. These parameters can be configured for optimal performance using some metric such as running time.

Algorithm configuration or **AC** is the tuning of algorithm parameters to optimize its behavior in some way. In this thesis, we use solving algorithms and optimize the search strategy of the algorithm to minimize the running times, but in other contexts AC can also be used to, for example, optimize solution quality.

Sequential Model-based Algorithm Configuration or **SMAC** is a specific approach to automatic algorithm parameter tuning to make algorithms perform better on a given set of problem instances [Hutter et al., 2011]. Problem-specific instance features can also be used by SMAC to improve its configuration quality.

---

competition2012_revised_without_SatELite_sparkle.txt

Sparkle [van der Blom et al., 2022] is a platform made to facilitate easy automatic algorithm selection and configuration. It can be used for various purposes, such as automated algorithm selection, automated algorithm configuration, ablation analysis and detailed result reports. During this thesis, we use Sparkle to configure our SAT solver algorithm for the fastest running time.

2.3 Running time Prediction

We compare the true and predicted running times of two algorithms during this thesis to see which is faster. We go over the relevant definitions here.

Default algorithm performance refers to the performance of an algorithm that has not been configured, so it uses the parameters that it came with out of the box. Often these are carefully chosen by the developer. For example, in the case of the Spear solver, the 'phase selection heuristic' is set to always select the phase which satisfies the most watched clauses, since that seems to be optimal in the most frequently encountered cases [Babić and Hutter, 2008]. **Configured algorithm performance**, refers to the performance of algorithms that have been configured for optimal performance.

Random forests or RF [Breiman, 2001] are prediction models that can be used for classification and regression. Classification means that for any given input, the RF outputs a classification (e.g. cat or dog). For regression, it outputs a numeric value instead. We use an RF for regression, so our RF is built out of several regression trees, which are models that can predict a continuous value when given an input. For these regression trees we use the DecisionTreeRegressor class from the SK-Learn Python library[2]. An RF returns the average of the output of all its contained regression trees.

Running time prediction models are models that predict the running time of, in our case, a SAT solver on a certain set of problem instances. As stated before, in this thesis we use an RF as our prediction model.

Censored data is data that is not fully known. In our case, SMAC can cut off unpromising runs early during configuration to save on resources. The observed running times are therefore lower than the actual running times would have been. This is referred to as right-censored data, since the actual data points lie to the right on a time distribution from the observed, capped running time. We will refer to these right-censored data points simply as censored running times in this thesis. Running times that came to a natural end, for example by finding the satisfiability of the problem, are conversely referred to as **uncensored**.

3 Related Work

In this thesis we make use of various sources which were already proven to be effective in literature. We go over those sources in this section.

[2]SK-Learn DecisionTreeRegressor: scikit-learn.org/.../sklearn.tree.DecisionTreeRegressor.html

3.1 Spear and SWV

We use the Spear SAT solver [Babić and Hutter, 2008], which is described as a "modular arithmetic theorem prover designed for proving software verification conditions." [Babić and Hutter, 2008]. The instance set we use during this thesis is called SWV. This instance set consists of 604 files and was created for use in software verification. The set was generated using Calysto [Babić and Hu, 2007] and further details describing this set can be found in [Hutter et al., 2014].

3.2 Sparkle and SMAC

In this thesis we use Sparkle [van der Blom et al., 2022] since it is an easy-to-use platform which simplifies configuration through SMAC, automatically collects useful data and provides several other key functions. Through this platform we configure our SAT solver for optimal running time. For our work we also extended Sparkle so that problem instance features can be used to configure solving algorithms. The specifics of these extensions are further detailed in Section 4. Sparkle uses SMAC to configure its algorithms. SMAC is a state-of-the-art configurator that can configure not only numerical, but also categorical parameters of an algorithm [Hutter et al., 2011].

3.3 Running Time Prediction

After configuring our solver, we want to be able to predict an algorithm's running time on unseen problem instances. Hutter et al. 2014 describe running time prediction based on random forests, which is the same approach we use in this thesis. From this paper, wherever possible, we borrowed the approach of running time prediction and the specific settings for e.g. setting up our random forest. When certain settings are not mentioned, we choose the most fitting option instead, such as the default value 'None' for the maximum depth of a regression tree, or Sparkle default settings for unspecified settings during configuration. More details on this can be found in Section 5.

4 Algorithm Configuration

With our goal in mind of monitoring the performance of an optimized algorithm configuration compared to the default, we need to configure an algorithm. We use a SAT solver as our configurable algorithm in this thesis. Conveniently, the configuration process also creates a dataset consisting of various algorithm configurations and their running times on specific problem instances, which we can use as data to train our monitoring method.

The algorithm configuration process needs several components: (a) A configurable algorithm, (b) an algorithm configurator and (c) a set of problem instances to run the algorithm on. A configurable algorithm is an algorithm that has adjustable parameters, often specified in a PCS file, that influence how the algorithm works. A configurator can then optimize its performance on a given set of problem instances. During this thesis we minimize the running time of a SAT solver. The performance of algorithm configuration can be improved even further by using instance features [Hutter et al., 2011]. For this we need an additional component: a feature extractor (d) that can compute features for the problem instances. For this purpose we use a modified version of the the feature extractor used in SATzilla [Xu et al., 2008] for the 2012 SAT competition, which is included

in Sparkle [3] .

The algorithm configurator we use in this thesis is SMAC [Hutter et al., 2011]. This tool can be used to optimize algorithms for various performance metrics, such as algorithm running time. The Sparkle platform [van der Blom et al., 2022] automates parts of the configuration process and uses SMAC internally. Although SMAC can use problem instance features during configuration, this functionality was not yet supported by Sparkle, so we extended the platform to be able to call SMAC with these problem instance features.

Since the next step in our process considers running time prediction, we chose a solver and instance set previously used in the running time prediction literature [Hutter et al., 2014]. Specifically the Spear solver [Babić and Hutter, 2008] and the SWV problem instance set [Hutter et al., 2014], both of which are available from ACLib[4]. To use Spear with Sparkle, we implemented a wrapper that ensured the inputs and outputs to Spear and SMAC were handled properly. Sparkle requires the train and test sets to be in separate folders, so we wrote a program called `split_files.py` that splits the SWV dataset into separate folders based on the `train.txt` and `test.txt` provided by ACLib. This program can be found in Appendix A.

Using these components, we configured Spear on the SWV training set through Sparkle and extracted the results. To run the configuration on the file server we ran the batch scripts `spear_with_features.sh` and `spear_without_features.sh` to run the configurations and then we ran `collect_spear_with[out]_features_data.sh` to validate and export the results. These files and further details on their use can be found in Appendix B.

The procedure described above results in both the best found configuration, which we describe in the following subsections, and the running times of all the other algorithm configurations tried by the configurator, which we will later use in our monitoring process. Additionally, we validate both the default and the best configuration by running them on all instances of the `SWV_Test` set and observing their running times so we can verify our predictions later on.

4.1 Experimental Setup

We ran our experiments on the Grace computer cluster of the ADA research group at Leiden University. The cluster uses the CentOS Linux 7.9.2009 operating system. It has 34 total nodes, 26 of which are CPU nodes, with the other 8 being GPU nodes. We ran all of our experiments on the CPU nodes of the cluster. All of the CPU nodes are identical and have the following specifications: Their CPU model is the Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz, they have an advertised max. speed of 2.1 GHz. They also have a cache size of 40 MB and a RAM of approximately 94 GB. To allow comparison to previous work [Hutter et al., 2014] we use their settings as much as possible, which are also documented in the scenario file that belongs to the Spear-SWV dataset [4]: A total time budget of 172800 seconds and a cutoff time per individual algorithm run of 300 seconds. The used performance measure is running time, because run time minimization is desirable in SAT

[3] Feature extractor: bitbucket.org/sparkle-ai/sparkle/src/development/.../description_SAT-features-competition2012_revised_without_SatELite_sparkle.txt

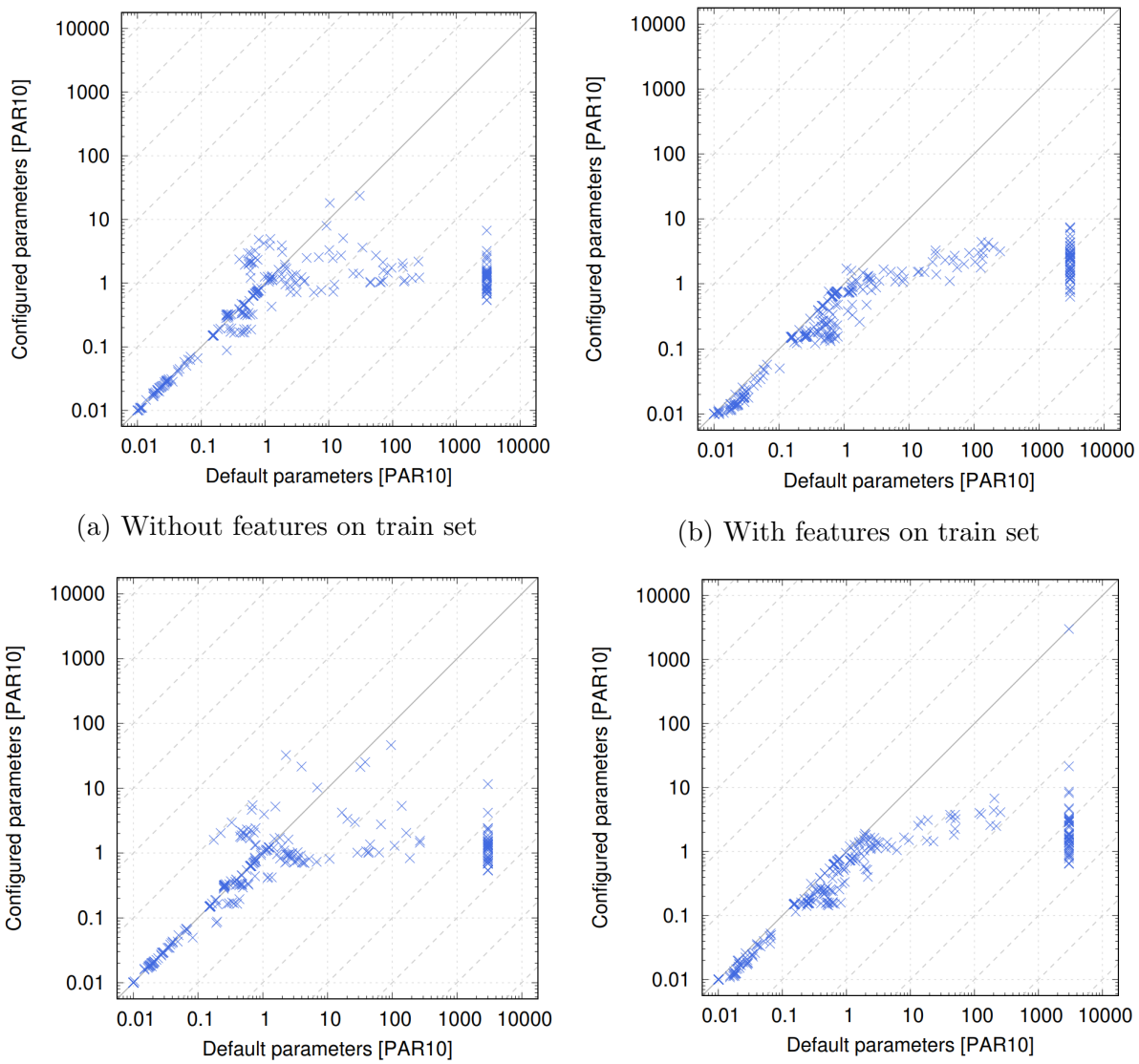
[4] Spear-SWV scenario file: bitbucket.org/mlindauer/aclib2/src/master/scenarios/sat/spear_swv/

solving. Further, we measure running time performance using the PAR10 metric, since it favors low average running times and strongly penalizes time-outs. This results in a configuration that performs well on many instances, in favor of performing very well on some instances and very poorly on others. The PAR10 scores are calculated with running times in CPU seconds. Other settings for which there was no logical default were instead taken from the default settings given in Sparkle.[5] The configurable parameters and the default configuration are given in a PCS file. We run 25 configurations in parallel and the best of the 25 found incumbents gets selected automatically by comparing their PAR10 scores on the entire training dataset. This is the default for Sparkle and it helps reduce the impact of the stochasticity of the configuration process compared to running only one configuration. We set the extractor cutoff time at 60 seconds per instance as it is the Sparkle default and only one second more than the time used for the 'expensive' features in [Hutter et al., 2014]. Note that the true computation time of these features is usually only a few seconds for this set, so the 60 second cap is never actually reached.
We compare the running times of two optimized configurations of the Spear algorithm in Experiment A, with one using problem instance features during the configuration process and the other one not using them. The settings are otherwise identical between the two runs. This experiment generates two configuration reports that compare the found configuration to the default configuration on both the train and test sets. It also generates two sets of running time data we can use later to train our monitoring system.

4.2 Experiment A: Algorithm Configuration

The results we will be looking at here come from the configuration reports of the two configuration runs. The full reports can be found in the supplemental files[6].

[5] Sparkle default settings: bitbucket.org/sparkle-ai/sparkle/src/.../sparkle_settings.ini
[6] The configuration reports can be found in `/Thesis Results/Configuration_Report_[No/With]_Features.pdf`



(a) Without features on train set

(b) With features on train set

(c) Without features on test set

(d) With features on test set

Figure 1: The measured running time plots of the two algorithm configurations on the train and test set of problem instances. The x and y axis show the default and configured algorithm PAR10 scores on the same instances respectively. The left column is for the configuration without features and the right for with features. The top row shows the train set and the bottom row the test set. The PAR10 scores are based on running times in CPU seconds. The cutoff time used is 300 seconds.

	Default	No Features	With Features
Train set	583.911	1.033	0.953
Test set	562.476	1.305	10.878
Test set no time-out	554.378	1.301	0.947

Table 1: PAR10 scores of the three algorithm configurations, namely the default and the ones configured without and with features on the train and test sets. The last row displays the PAR10 score for all configurations on the test set with the single instance which caused a time-out removed. The PAR10 scores are based on running times in CPU seconds. The cutoff time used is 300 seconds.

From this point onward, we will be referring to the default algorithm as DEF, the algorithm configured with features as WF (with features) and the one configured without features as NF (no features). The PAR10 scores for all three algorithm configurations on the train and test instance set are displayed in Table 1. This score is the average of all running times, with time-outs counted as 10 times the running time. Since our maximum running time is 300 seconds, any time-out gets counted as 3000. Even with identical configurations, running times can differ slightly between runs. Therefore we specify that the DEF running times are taken from the configuration without features. We calculated all scores in the table so the 0.001 seconds of difference on the test set between the DEF scores there and the configuration report are probably due to floating point inaccuracies. As is visible in Figure 1d, there is one time-out in the test set of WF, which the configured and default algorithm both performed so poorly on that they timed out at 300 seconds. This makes the PAR10 score a lot higher for the configured algorithm, which makes it hard to compare WF and NF. So, in order to allow proper comparison, we also included the row with the PAR10 scores that had the time-out removed.
If we just look at the PAR10 scores, WF has the highest performance on the train set, while NF has the highest on the test set. However, WF only performs better by an average of about 0.35 seconds on the test set if we remove the time-out, which is negligible in practice. Taking a look at the performance plots in Figure 1, we can also see that NF (see Fig. 1a, Fig. 1c) has more mixed performance in relation to DEF, while WF seems to consistently either perform better or about as good as DEF (see Fig. 1b, Fig. 1d). Given that outliers are very hard to predict and how good the performance of WF is on the rest of the test instances while still having a time-out with a running time over 300 seconds, we cannot guarantee that NF is immune to such exceptions either. With all of this taken into consideration, it does appear to be beneficial to configure an algorithm while using problem instance features for our application.

5 Running Time Prediction

In order to set up a running time monitoring system, we we need to be able to predict the running times of a given algorithm configuration on a given problem instance. Therefore we need to build a prediction model. A previous running time prediction study [Hutter et al., 2014] suggests that random forest models (RF) are a good fit for this purpose. They observe that regression trees are flexible and effective for discrete input data, but prone to overfitting. Random forests overcome this by growing several regression trees and averaging the output and they state that their strong predictions for complex data makes them a natural choice for running time predictions of algorithms

with many parameters.
Now that we have established what model we will be using, we need the data to train the model on. For this we use the data generated during the algorithm configuration. This consists of three essential parts: (1) Various configurations of the considered algorithm (2) their running times on problem instances and (3) the instance features of those problem instances. The running times are needed, since that is what we will use to grow our RF and to test its accuracy on. The algorithm configurations are necessary because we want to investigate the differences in running times of different configurations on identical problem instances. Finally, the problem instance features are important because we want to be able to predict algorithm performance on different, unseen instances. Because instance features give information about the problem structure, we can use this to predict the performance of a configuration on specific instances, since both problem structure and the configuration influence algorithm running time.
In our experiments, we consider censored value approximation and bootstrapping, running scenarios with one or the other, neither and both, resulting in four scenarios we can compare. Censored values refer to the fact that some of the observed running times have been cut off before the solver finished. This is because SMAC terminated it early since it was unpromising and thus its actual expected running time is longer than the observed running time. Using these censored running times as if they were truly that fast would bias our model downwards, so instead we approximate the true running times using our RF. This censored value approximation was based on an approach found in SMAC [Hutter et al., 2013], which is an RF implementation of an Expectation-Maximization method by Schmee and Hahn [Schmee and Hahn, 1979]. The details of this will be explained in Subsection 5.1. Bootstrapping means sampling from the training dataset with repetition, so any one data point could occur zero times, once or multiple times. In our implementation, if it is enabled, we bootstrap a separate set of data points for each regression tree in our RF as detailed in the next subsection. This is to increase diversity in the regression trees, which should increase the accuracy of our RF.

5.1 Methods

During the fitting process, we want to be able to fit each regression tree on a different dataset. This is something which is not possible with the Python `RandomForestRegressor` class from the SK-Learn library. For that reason we build our own RF using their `DecisionTreeRegressor` class. To use Sparkle's configuration output to train our RF, we process it to create datapoints including the running times, parameter configurations and instance features (see Appendix C). For each of the data points we also save whether it was censored or not. We do this for each of the 25 configuration runs and for each of their algorithm runs and we put all of those points into a single dataset. We use the same approach for fitting an RF on a dataset that contains censored data as [Hutter et al., 2014]. Our implementation of an RF is the class called `RegressionForest` (See Appendix C). We first fit our RF on the uncensored data points. Then using the constructed RF, we can approximate the censored values by using a truncated Gaussian distribution. This distribution is described by the probability density function in the following equation:

$$p(y) = \begin{cases} 0 & y < K_i \\ \frac{1}{\sigma_i}\varphi(\frac{y-\mu_i}{\sigma_i})/(1-\Phi(\frac{\mu_i-K_i}{\sigma_i})) & y \geq K_i \end{cases} \tag{4}$$

In this equation, φ stands for the probability density function and Φ stands for the cumulative distribution of a normal distribution. $p(y)$ stands for the new predicted running time, y stands for the prediction of the running time by the current forest, μ_i and σ_i stand for the mean and standard deviation of this prediction respectively and K_i stands for the time the run was capped at. Examples of a left and right-truncated normal distribution that result from this formula can be seen in figure 2. Using such a left-truncated distribution where the truncation point is the capped running

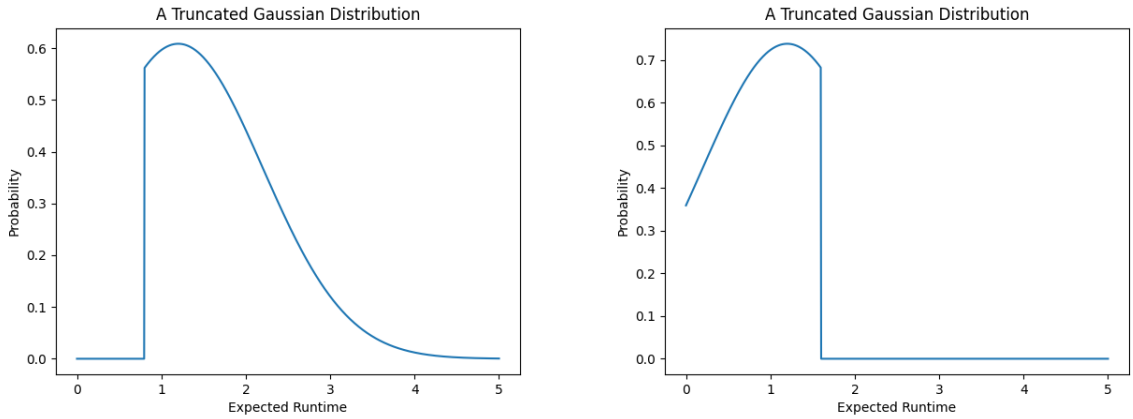


Figure 2: Two Gaussian distributions. The left figure is left-truncated at 0.8 seconds and the right figure is right-truncated at 1.6 seconds. Both have the same original normal distribution with a mean of 1.2 seconds. Take note that the probability of randomly pulling a value from this distribution is 0 for running times that have been truncated.

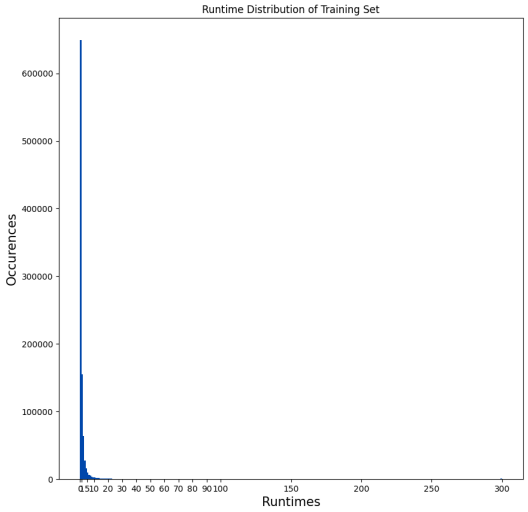
time, we can approximate the true running times of an algorithm. To do this we iterate between two steps: The first step is iterating over every censored data point of every tree and randomly sampling a value from the distribution to replace it. This means that every tree gets its own set of training data points, even if the initial dataset was identical for all trees in the RF, such as when bootstrapping is disabled. The second step is refitting each tree on its new dataset. These two steps are repeated until the maximum amount of iterations is reached or the new values on average increased less than 10^{-10} compared to the old values, which means the values have converged. After this we have an RF that can be used for predicting running times on specific problem instances.

To start the experiments, we find all running times and their respective algorithm configurations and problem instance features and convert them into a set of data points. All running times are then 10-log-transformed since the running times can vary greatly in these problems and this should increase our prediction accuracy [Hutter et al., 2014]. Our dataset contains 961,700 total instances. To keep memory usage and computation time manageable, we sample 20,000 points randomly from this dataset. These 20,000 points then become the dataset we grow all regression trees on, since we leave the rest of the data points out to save memory. Each `RegressionForest` is made up of 10 `DecisionTreeRegressors` from the SK-Learn library. The `min_samples_split` of each `DecisionTreeRegressor` is 5, which means a leaf node must have at least five samples if it wants to be split further. The `max_features` setting specifies how many features can be chosen as a split parameter in a leaf node and is set to 0.5, which means a random subset of half of all possible features are evaluated instead of all of them. This ensures that all regression trees grow differently,

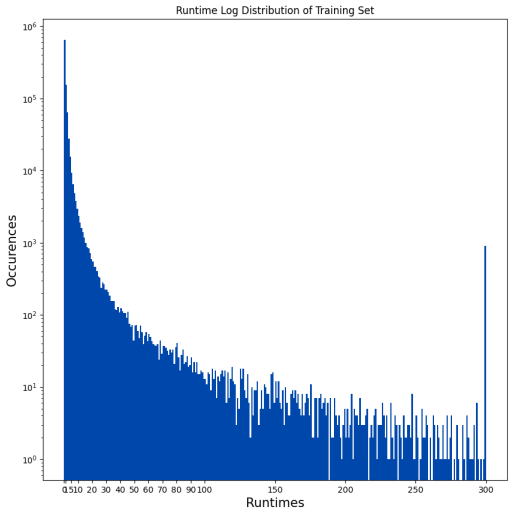
even if they were to be grown on the exact same training dataset. Using the 20,000 points and our `RegressionForest` class, we can start our 10-fold cross-validation.

5.2 Experimental Setup

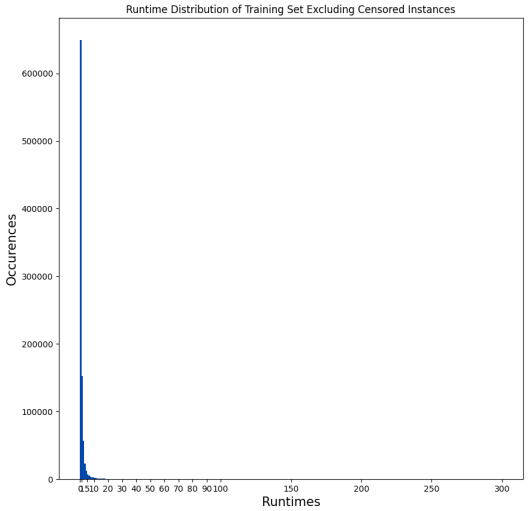
In our experiments, we compare various methods of growing our RF and what effect they have on our results. Experiment B (Section 5.3) focuses on bootstrapping and censored value approximation. In this experiment we build four sets of ten RFs; One with bootstrapping enabled, one with censored value approximation enabled, one with neither and one with both. We then perform 10-fold cross-validation for each of the four sets and report their average results.
For Experiment C (Section 5.4), we did this same experiment, but we limit which points can be picked from the complete set based on running time, where the two running time boundaries we chose are 1 second and 10 seconds. This means the program can only sample points with a running time higher than 1 or 10 seconds from the dataset. The reason for this can be seen in Figure 3: Lower running times are much more prevalent than higher running times. Running times below or around one second are especially common. For this reason we investigate the quality of our RF if we exclude these lower running times. We implement this by collecting all data points that are higher than or equal to the given minimum running time and then sampling 20,000 unique points. Then we perform 10-fold cross-validation in the same way as Experiment B. Since Experiment B is the same as this experiment, but with a minimum running time of 0, we can also compare the results of this experiment to those of Experiment B.
As mentioned, low running times are overrepresented. However, if we exclude lower running times (as in Experiment C), our RF will be unable to predict running times that would fall lower than that exclusion boundary. Besides that, this exclusion does not influence the spread of the running times that would be usable, causing the random sampling to still favor lower running times. For this reason, in Experiment D (Section 5.5), we represent running times more equally to see how it influences our predictions. To achieve this, we create three running time 'buckets' with each bucket containing all data points between two running times. The first bucket is for running times between 0 and 1, the second for running times between 1 and 10 and the last is for all running times above 10 seconds. We run this experiment twice, the first time using only the second and last bucket and the second time using all buckets. When using only the two highest running time buckets, we sample 10,000 data points from each of these buckets and build our RF on that. If we use all three buckets, we sample 6,666 points instead. This is because two times 10,000 is 20,000 and three times 6,666 is 19,998 which is as close as possible to the 20,000 points we used in Experiment B and C. The experiment then proceeds the same as the other experiments, cross-validating with bootstrapping, censored value approximation, neither and both.



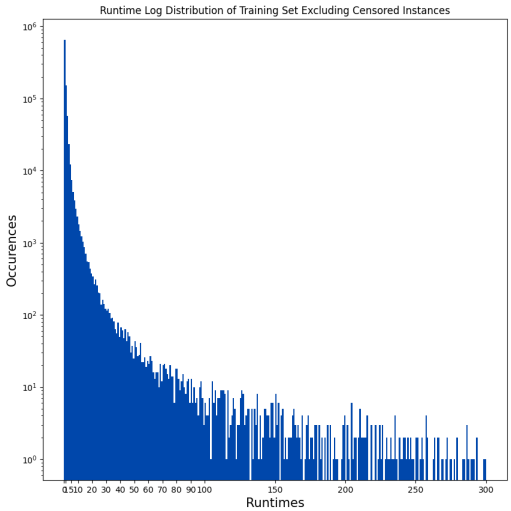
(a) The distribution of running times



(b) The log-scaled distribution of running times



(c) The distribution of running times, excluding censored running times



(d) The log-scaled distribution of running times, excluding censored running times

Figure 3: The distribution of running times in our training dataset. The images on the right are logarithmically scaled to show how the higher running times are distributed. It is clearly visible that running times around or below one second make up the largest part of our dataset by far. Excluding the censored running times also shows that little to no censored running times are below one second.

5.3 Experiment B: Bootstrapping and Censored Value Approximation

To assess the predictive quality of our RF, we use the Root Mean Squared Error or RMSE and the Pearson Correlation score. The RMSE is derived from the Mean Squared Error or MSE. The MSE is the average of the squared errors of all predictions in a dataset and the RMSE is the square root of that value. It is measured in the same value as the data points. For example, in the context of our experiments an RMSE of 1 means an average difference of 1 second between predictions and true values. The Pearson Correlation score or simply correlation indicates the predictive quality of a set of predictions. It is a value between -1 and 1, where 1 means the predictions perfectly describe the true values. Conversely, -1 means the predictions rise as fast as the true values drop and vice-versa. If, for example, all running times in a training set fall below 1 second, the RMSE could not exceed 1 second, but this does not mean the predictions are good. For this reason we use both of these measures to assess our predictions.

RMSE	Without Bootstrapping	With Bootstrapping
Without CVA	10.919	9.770
With CVA	12.220	12.166
Correlation	**Without Bootstrapping**	**With Bootstrapping**
Without CVA	0.276	0.260
With CVA	0.369	0.375

Table 2: Root Mean Square Error and Pearson Correlation scores of our Random Forest. Scores 10-cross-validated with a total set of 20,000 data points sampled from the full data set. CVA stands for Censored Value Approximation.

The lowest RMSE is found with bootstrapping enabled and with censored value approximation disable. The highest correlation is found with both enabled. Therefore we cannot directly choose the best configuration. However, since both of these values have bootstrapping enabled, we conclude this must be good for the predictions of our RF. We can also see that CVA increases correlation scores. As explained earlier, CVA uses the RF to approximate the true values of censored running times. These values are always as high as or higher than the original censored value. Therefore it makes sense the RMSE increases, since more high values are present. If we take this into account, we conclude that CVA increases the predictive power of our RF.
The best found RMSE is 9.770 seconds, which means the average running time prediction of our forest is more than 9 seconds off the true running time. The highest Pearson Correlation value is 0.375. When we look at the results in [Hutter et al., 2014], we see their predictive quality is much higher. For example, with 10,000 training data points, they achieved an RMSE of 0.34 seconds (see their Table 2). Although there are no correlation scores for Spear-SWV in the study, their Fig. 5 shows RF correlation scores higher than ours using much fewer data points for each of the data sets. Since our approximation method is the same, namely an RF, we conclude our approach to data collection must have biased our dataset. This can be seen in the average running times of the data points in both sets. Their dataset has an average algorithm running time of 60 seconds (see their Table 3) while ours has an average running time of about 1.28 seconds. This difference is because we used the running time data generated by the configuration process of the Spear solver

which optimizes for running time, while the [Hutter et al., 2014] study randomly sampled 10,000 combinations of training configurations and instances without optimizing for running time. Since our training set is heavily biased downwards (Figure 3) this also biased our RF such that it will be hard, if not impossible for the RF to accurately predict high running times, since regression trees cannot predict outside the boundaries of their given data and many trees in the RF probably contain little, if not no running times above 5 seconds. Even if a few of the trees in the RF do contain higher running times, the RF would still be biased downwards for these predictions, since a regression RF averages the predictions of all its trees. In order to limit the impact of this issue, we tried different sampling methods from our dataset, as can be seen in experiments C and D.

5.4 Experiment C: Excluding Running times Below a Boundary

This experiment is the same as Experiment B, but we sample differently. We exclude all runtimes below a certain running time and sample from the rest of the dataset. To do this, we take the full dataset of 961,700 points and exclude all points with a running time below either 1 second (Table 3) or 10 seconds (Table 4). We then sample 20,000 points from this new set and proceed as in Experiment B. The results of this experiment should show us if this is an effective way to deal with bias or if other methods should be tried. Note that the predictive quality is measured through 10-fold cross-validation, which means it is on the same domain as the training set, only with different points. Since `DecisionTreeRegressors` cannot predict outside of their trained range, we suspect lower scores on these experiments if we test them on a test set sampled from the full distribution.

RMSE	Without Bootstrapping	With Bootstrapping
Without CVA	20.187	20.283
With CVA	21.571	22.418
Correlation	Without Bootstrapping	With Bootstrapping
Without CVA	0.222	0.264
With CVA	0.367	0.342

Table 3: Root Mean Square Error and Pearson Correlation scores of our Random Forest. Scores 10-cross-validated with a total set of 20,000 data points, all with a running time of 1 second or higher, sampled from the full data set. CVA stands for Censored Value Approximation.

RMSE	Without Bootstrapping	With Bootstrapping
Without CVA	63.144	63.749
With CVA	65.385	65.547
Correlation	Without Bootstrapping	With Bootstrapping
Without CVA	0.316	0.316
With CVA	0.514	0.528

Table 4: Root Mean Square Error and Pearson Correlation scores of our Random Forest. Scores 10-cross-validated with a total set of 20,000 data points, all with a running time of 10 seconds or higher, sampled from the full data set. CVA stands for Censored Value Approximation.

These results do not seem better than those of Experiment B. The RMSE scores are higher, though this can be explained by the fact that there are more high running times in the dataset, since it is not mainly filled with running times around or below one second. However, as visible in Figure 3 the distribution is still right-skewed, even if we remove the lower values. This can be seen in the average running times of the reduced datasets. With all values below 1 second removed, the average is 3.539 seconds. The average is 26.029 seconds for all values of 10 or higher. Hence our dataset is still biased and this influences our predictions negatively. This can also be seen in the results. Table 3 shows that excluding values below 1 second does not make our predictions more accurate. The lowest RMSE is 20.187 and the highest correlation is 0.367 as compared to the 9.770 and 0.375 we found in Experiment B. All RMSE scores are higher and all correlation scores are lower than their Experiment B equivalent. This means excluding runtimes below one second does not seem like a good approach. When we look at Table 4 we see similar results. Although the correlation score is improved, with a highest value of 0.528, the RMSE has become so high, with a lowest value of 63.144, that this is not an acceptable trade. It should also be noted that an RF trained on data that is sampled like this is unable to predict any values below the exclusion boundary. Therefore, although we were able to increase correlation values slightly, the approach does not seem viable to use in practice.

5.5 Experiment D: Equal Running Time Buckets

Because the approach from Experiment C produced unsatisfactory results, we try another approach in this experiment. We create three running time 'buckets' with the domains of [0,1), [1,10) and [10, 300] seconds, which we distribute the appropriate data points over from the 961,700 of the full set. Then we sample an equal amount of points from each of these buckets to get a more evenly distributed dataset. We run two experiments with this approach, one using only the two highest running time buckets and the other with all three. When using two buckets, we sample 10,000 points from both for a total of 20,000 points like in Experiments B and C. When using all buckets, we sample 6,666 points for a total of 19,998 to get as close as possible to 20,000. We then proceed with the experiment as in Experiments B and C, performing 10-fold cross-validation with bootstrapping or CVA, neither and both. The results from this experiment will tell us if this is a useful way to improve the predictive strength of our RF with the limitations of our dataset. Note that, as in Experiment C, the predictive quality is measured through 10-fold cross-validation. This means it is tested on the same domain it is trained on, only with a different fold. Since `DecisionTreeRegressors` cannot predict outside of their trained range, we predict lower scores if we were to test on a set sampled from all 961,700 data points, at least on the experiment with the two highest buckets. That experiment might also produce different results, since the running time distribution in the test set would be more biased, however, this difference might be positive or negative. It could be good for the RMSE scores because of the higher amount of low running times, but if the RF would predict high running times for these low values, it could negatively affect the RMSE instead. The same principle applies to the correlation score.

RMSE	Without Bootstrapping	With Bootstrapping
Without CVA	40.423	41.274
With CVA	47.369	48.580
Correlation	Without Bootstrapping	With Bootstrapping
Without CVA	0.282	0.278
With CVA	0.371	0.377

Table 5: Root Mean Square Error and Pearson Correlation scores of our Random Forest. Scores 10-cross-validated with a total set of 20,000 data points sampled from the full data set. 10,000 points each from the domains [1,10) and [10, 300]. CVA stands for Censored Value Approximation.

RMSE	Without Bootstrapping	With Bootstrapping
Without CVA	30.377	31.220
With CVA	33.921	35.017
Correlation	Without Bootstrapping	With Bootstrapping
Without CVA	0.274	0.316
With CVA	0.355	0.322

Table 6: Root Mean Square Error and Pearson Correlation scores of our Random Forest. Scores 10-cross-validated with a total set of 19,998 data points sampled from the full data set. 6,666 points each from the domains [0,1), [1,10) and [10, 300]. CVA stands for Censored Value Approximation.

The results in Tables 5 and 6 are from using the highest two buckets and using all three buckets respectively. The results from both experiments yield similar correlation scores and although the second experiment has lower RMSE values than the first, this can be explained by the higher amount of low values present in that experiment, since it uses the [0,1) bucket. The lowest RMSE is 30.377 seconds and the highest correlation is 0.377. We compare this to Experiment B, where the lowest RMSE is 9.770 and the highest correlation score is 0.375 and conclude the RMSE scores are higher and the correlation scores do not seem to improve meaningfully when compared to Experiment B. To explain this, we examine the average running times of each bucket. The three buckets have an average running time of 0.910, 2.379 and 26.029 seconds respectively. This gives us a weighted average of 9.772 seconds if we sample equally from all three buckets or 14.204 seconds if we only sample from the highest two buckets. When we compare to the average running times in the training dataset of [Hutter et al., 2014] we see their average is 60 seconds. The difference in these scores reveals that our distribution is still more biased downwards, even if we sample like this. This low average means that our `DecisionTreeRegressors` will be biased downwards and even if some of them are not due to sampling higher data points, the predictions from our RF will likely still be heavily biased since it takes the average of all predictions of its trees.
Therefore we conclude that the approach of sampling equally from three running time domains does not increase the predictive power of our RF. However, if we were to use the same method of generating data as in the study mentioned previously, we expect our performance monitoring in the next paragraph should be improved as well.

6 Performance Monitoring

Based on the work in previous sections, we now have the components we need to enable performance monitoring. These parts are (1) a predictive model, here a random forest, (2) the instance features of problem instances and (3) configurations of the Spear solver. These three parts will allow us to predict the running time of given configurations of the Spear solver on previously unseen problem instances. We built our RF in Section 5 and we found the default and an optimized configuration in Section 4. For problem instances to test on, we cannot use the `SWV_Train` set, since that is the dataset our optimized configuration was configured on. Another dataset that we already have running times for is `SWV_Test`. The instance features of this set were also automatically calculated before the configuration process. This dataset is previously unseen by the prediction model and can therefore be used to verify our predictions. It should be noted that we do include the `SWV_Train` set in our experiments below so we can compare our predictions on that set to those on the `SWV_Test` dataset.

From the prior experiments, we concluded the RF configuration that worked best was with bootstrapping turned off and censored value approximation turned on (Sect. 5.3). To initiate the monitoring process, we build and save an RF trained with these settings on `SWV_Train`. Then we compute the features for the problem instances we want to predict running times on. The set we use is `SWV_Test`, of which the features were already calculated in our case during the configuration process (Sect. 4). The true running times of the default and configured algorithms on these sets can be found in Figure 1. Then we put the configuration parameters and the instance features into the RF to predict the running times of both the default and the optimized configurations on these instances. We compute the running time the optimized configuration needs to outperform by using a threshold and the default configuration running speed. Based on these predictions, we either tell the user to solve with the configured solver or to reconfigure their solver for use on these instances.

To evaluate the accuracy of our monitoring system, we run Experiments E and F. We establish the ground truth as the true running times of both the default and configured algorithms on the `SWV_Test` set, observed during the configuration of the Spear solver (Sect. 4). Then we predict the running times of these configurations. Using these predictions, we calculate whether the configured algorithm would be faster than the default by a certain percentage, which is the threshold we established. We then perform the same calculation with the true values and observe whether the predictions match the truth.

6.1 Experimental Setup

The threshold works as follows: If the running time of the configured algorithm is 1.8 seconds and the default algorithm takes 2 seconds on the same problem instance, the configured algorithm would be counted as quicker without a threshold. However, with a threshold of 15% (0.3 seconds), the configured algorithm would not be fast enough according to the user-given threshold, since it is not faster than 1.7 seconds. We run Experiment E with thresholds of 0%, 5% and 20% and report the accuracy of our predictions below. In case the user does not need the configured algorithm to outperform the default, but only perform similarly, they can use a threshold below 0%. To illustrate this, we run Experiment F with a threshold of -5%. Since the monitoring algorithm can also predict

when the default will perform better, this threshold can be used for very tough instance sets where, possibly due to a high variety between instances or general instance complexity, running times are often high and it can be hard to find a specific configuration that performs well on all instances.. In such a case, the user could choose to use the default instead of the configured to save resources or choose to generate a new configuration.

6.2 Experiment E: Performance Monitoring (0%, 5%, 20%)

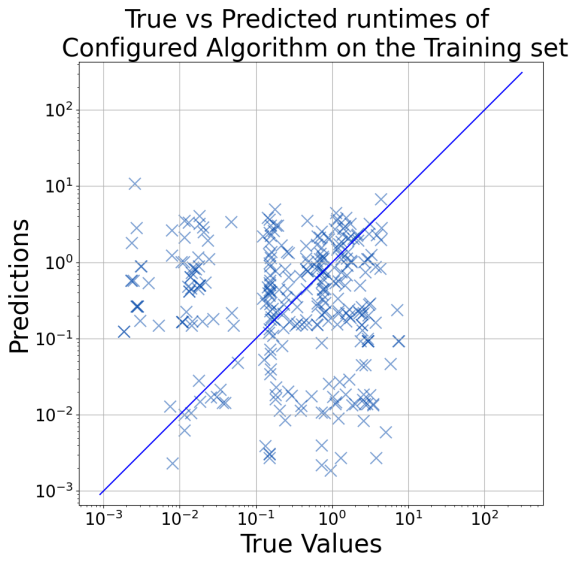
Prediction \ Truth	Configured Faster	Default Faster
Configured Faster	219 (72.52%)	23 (7.62%)
Default Faster	45 (14.90%)	15 (4.97%)

Table 7: The predictions of our monitoring algorithm on `SWV_Train` using a threshold of 0%. Percentage scores indicate what percentage the number before is out of the 302 total instances in either set.

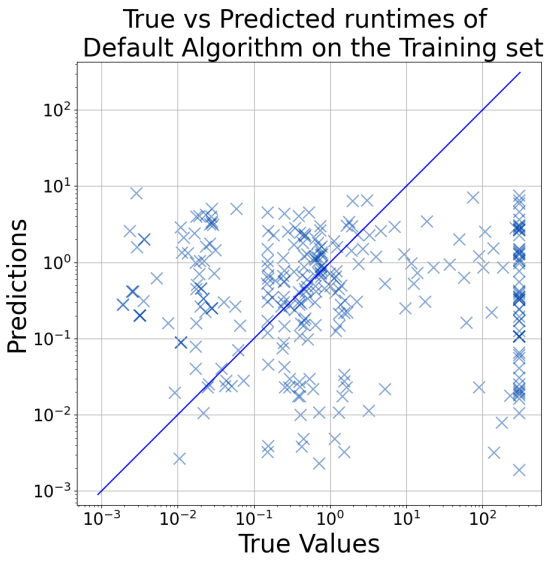
Prediction \ Truth	Configured Faster	Default Faster
Configured Faster	233 (77.15%)	25 (8.28%)
Default Faster	40 (13.25%)	4 (1.32%)

Table 8: The predictions of our monitoring algorithm on `SWV_Test` using a threshold of 0%. Percentage scores indicate what percentage the number before is out of the 302 total instances in either set.

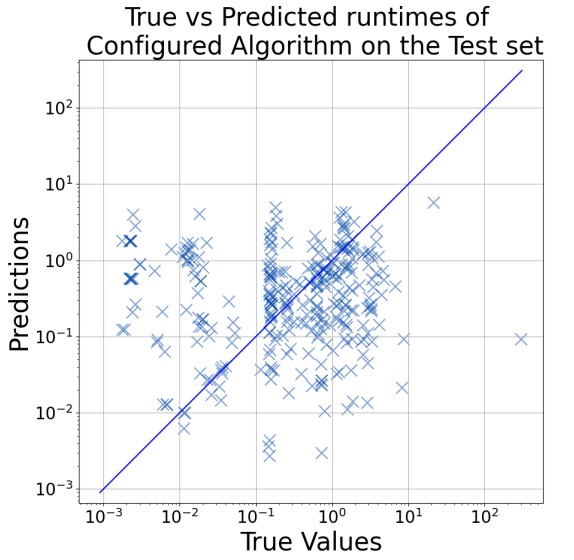
The results, which can be seen in Tables 7 and 8, are of both the `SWV_Train` and `SWV_Test` with a threshold of 0%. They seem usable, with 234 and 237 correct predictions on the train and test set respectively. However, with only a bit more than 75% accuracy, the question is whether these predictions are reliable enough to use in practice. The plots in Figure 4 compare the true and predicted running times of the default and configured algorithms on `SWV_Train` and `SWV_Test` and they reveal undesirable behavior.



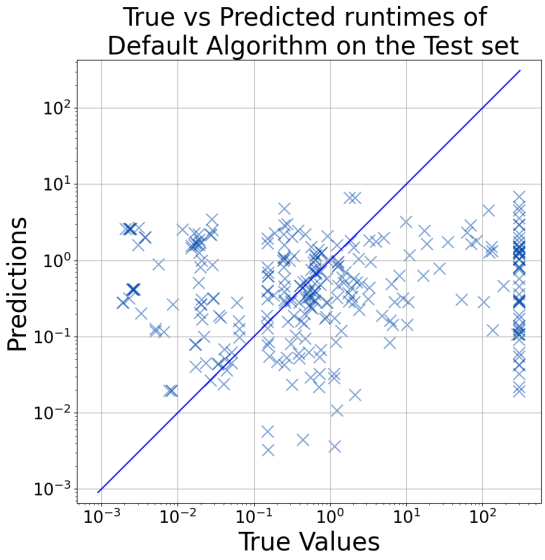
(a) True vs predicted running times of the configured algorithm on the `SWV_Train` set



(b) True vs predicted running times of the default algorithm on the `SWV_Train` set



(c) True vs predicted running times of the configured algorithm on the `SWV_Test` set



(d) True vs predicted running times of the default algorithm on the `SWV_Test` set

Figure 4: The true vs predicted running times of the configured (left) and default (right) configurations of the Spear solver on `SWV_Train` (above) and `SWV_Test` (below). The x axis denotes the true running times and the y axis the prediction of our RF. The blue line indicates where a 'perfect prediction' should be.

As is visible in these plots, our RF is heavily biased downwards, as also seen in the earlier experiments. These plots also explain the bias of our RF when the configuration process of the Spear solver

is taken into account: The default algorithm has quite varied running times, even having several time-outs, while the configured algorithm only has very low running times. Since the configurator focuses on lower running times and cuts off any runs that take too long (see 'censored data' in Section 2) it is likely that the dataset our RF is trained on is saturated with low running times, as also observed in Experiment A in Section 4, thus biasing our RF downwards. This bias is, again, visible in all of its low estimates, since not one of its predictions exceeds even 25 seconds. Looking at Figure 4 we can see that an estimate below 25 seconds would be in the correct range for the configured algorithm (see Figures 4a and 4c), with the exception of the single time-out on the test set, but for the default algorithm, many of the measured running times far exceed 25 seconds (see Figures 4b and 4d).
We also implemented a threshold that the configured algorithm should outperform. For instance, a threshold of 5% would mean the configured algorithm needs to finish a run in 95% of the time in which the default can or faster. In tables 9, 10, 11 and 12 are the results for the same data points as in Tables 7 and 8, but with a threshold of 5% and 20% respectively.

Truth / Prediction	Configured Faster	Default Faster
Configured Faster	186 (61.59%)	36 (11.92%)
Default Faster	54 (17.88%)	26 (8.61%)

Table 9: The predictions of our monitoring algorithm on `SWV_Train` using a threshold of 5%. Percentage scores indicate what percentage the number before is out of the 302 total instances in either set.

Truth / Prediction	Configured Faster	Default Faster
Configured Faster	187 (61.92%)	47 (15.56%)
Default Faster	56 (18.54%)	12 (3.97%)

Table 10: The predictions of our monitoring algorithm on `SWV_Test` using a threshold of 5%. Percentage scores indicate what percentage the number before is out of the 302 total instances in either set.

Truth / Prediction	Configured Faster	Default Faster
Configured Faster	118 (39.07%)	32 (10.60%)
Default Faster	100 (33.11%)	52 (17.22%)

Table 11: The predictions of our monitoring algorithm on `SWV_Train` using a threshold of 20%. Percentage scores indicate what percentage the number before is out of the 302 total instances in either set.

Prediction \ Truth	Configured Faster	Default Faster
Configured Faster	115 (38.08%)	50 (16.56%)
Default Faster	95 (31.46%)	42 (13.91%)

Table 12: The predictions of our monitoring algorithm on `SWV_Test` using a threshold of 20%. Percentage scores indicate what percentage the number before is out of the 302 total instances in either set.

As might be expected, the predictions get progressively poorer the higher the threshold is, with our RF estimating barely better than chance with a 20% threshold (56.3% accuracy on `SWV_Train` and 52.0% on `SWV_Test`.) This can be explained by the fact that our RF generally predicts very low running times, since this brings all predictions closer to the 0 second mark. This higher density of running time predictions means that, for example, default algorithm running times which should be predicted to be much higher than configured algorithm ones on the same problem instances are only predicted to be marginally higher. This marginal difference in running times results in the prediction of the configured algorithm not being faster than the 20% threshold it should outperform, even though the true running time is. This is why the number of times the default is falsely predicted as faster rises sharply in Tables 11 and 12.
If not for the bias in our dataset, if our RF had an accuracy similar to that seen in [Hutter et al., 2014], this system should easily and quickly be able to predict whether the configured algorithm would still outperform the default algorithm based on some given threshold expectation. However, for use cases similar to our own, where the dataset of the user is biased downwards, we still want to investigate the potential of our performance monitoring method. For this reason, we ran Experiment F.

6.3 Experiment F: Performance Monitoring (-5%)

Experiment F is like Experiment E, but with a threshold of -5%. This threshold is still usable, since a user might not want to reconfigure if their optimized algorithm still performs similarly to the default or if the user wants to pick between either configuration for each instance. In this use case, the user would reconfigure if the configured solver performs significantly worse than the default.

Prediction \ Truth	Configured Faster	Default Faster
Configured Faster	262 (86.75%)	3 (0.99%)
Default Faster	37 (12.25%)	0 (0.00%)

Table 13: The predictions of our monitoring algorithm on `SWV_Train` using a threshold of -5%. Percentage scores indicate what percentage the number before is out of the 302 total instances in either set.

Viewing these results in the context of Experiment E, it makes sense the accuracy of our RF would increase, since the threshold is decreased instead of increased. With this threshold of -5%, we have 262 correct predictions and 40 wrong predictions on `SWV_Train`, giving us an accuracy of 86.75%.

Prediction \ Truth	Configured Faster	Default Faster
Configured Faster	267 (88.41%)	1 (0.33%)
Default Faster	34 (11.26%)	0 (0.00%)

Table 14: The predictions of our monitoring algorithm on `SWV_Test` using a threshold of -5%. Percentage scores indicate what percentage the number before is out of the 302 total instances in either set.

On `SWV_Test`, the RF has a slightly higher accuracy, with 267 correct predictions and 35 wrong ones, for an accuracy of 88.41%. This higher accuracy suggests that a user with data similar to ours can still utilize our performance monitoring approach to determine the performance of their configured solver. However, it should be noted that on `SWV_Train` and `SWV_Test`, the configured algorithm was slower than the threshold a combined amount of only 4 times and in all these 4 cases the prediction was wrong. This means the RF would never suggest to reconfigure the solver in correct cases, only in the combined 71 false negatives. As such, although the results seem better than those of Experiment E, the predictions of our RF built on biased data keep their limitations. Despite our results, we think these thresholds do provide a useful way to approach algorithm configuration. The performance monitoring approach we provide can be used in cases where the configured algorithm must be much faster than the default and cases where it can be a certain amount slower, since it simply scales off the predicted running time of the default algorithm. Our warning system could also be used in a system which aggregates the amount of warnings or how much slower the configured algorithm is predicted to be over a certain amount of time and then takes action based on this aggregated data.

7 Conclusion

In this thesis we introduced a monitoring algorithm to be used in a Self-Monitoring, Automated Algorithm Configuration System that monitors whether or not the user should consider reconfiguring their solving algorithm. To achieve this we configured the SAT solver Spear with SMAC through Sparkle. Using the configuration data, we constructed a Random Forest (RF) to predict performance on new problem instances. Using this predictive model we implemented a thresholding system. With this we aimed to answer two questions: First whether configuration data is useful in constructing a reliable predictive model and second if thresholding is a feasible approach to monitoring.

The configuration process gave us two optimized algorithms, one configured with and the other without problem instance features. The running times of these configurations on the `SWV_Train` and `SWV_Test` sets lead us to conclude that configuring a solver with features is beneficial for our application. This solver optimized with instance features is the one we later compared to the default in our monitoring algorithm. The configurator running many different parameter configurations in this process also gave us a set of algorithm running times along with the configurations of those algorithms and the features of the problem instances they were run on. We used these running times to train our RF.

In building our RF, we considered whether to use bootstrapping and censored value approximation

(CVA). We investigated this by running 10-fold cross-validation and observing which RF configuration performed the best. The results lead us to build and export an RF with bootstrapping and CVA both enabled for use in our monitoring algorithm. Our dataset was biased downwards from the configuration process favoring lower running times and this negatively influenced the accuracy of our RF. To remedy this, we tried different sampling methods to equalize the running time distributions to train our RF on, but the results did not indicate this was a useful way of accounting for the bias.
The performance monitoring algorithm imports the RF exported in the previous step. When given an algorithm configuration and a set of problem instance features, it gives a predicted running time. We used this to make predictions for both the default and the configured algorithm to see which is faster. We also implemented a 'threshold' where, for example, the configured algorithm would need to have a running time of 80% of the default or less if the threshold is 20%. We made predictions for all instances in `SWV_Train` and `SWV_Test` using thresholds of 0%, 5%, 20% and -5% and compared them to the true running times found during configuration. We evaluated the accuracy of the predictions by finding the amount of correct predictions our system made over both sets. We found that, with our biased RF, these predictions were probably not suitable for practical use. However, previous research has shown that RFs are very suited for running time prediction [Hutter et al., 2014]. Therefore, if a different approach to building an RF is used and the accuracy improves, we think the system we set up is a useful way to approach algorithm reconfiguration.

The monitoring algorithm is fast, making almost instant predictions and the random forest that it uses can be generated within minutes. Because of the versatility of RFs, this system could very easily be used for other problems than the SAT problems we explored, such as Traveling Salesperson problems and Mixed Integer Programming problems. Since generally solving algorithms can have potential time-outs that take several minutes before being cut off, it is much quicker to use this algorithm than to run a solver on an instance and observe its running time after it is done. Since the cost of running our monitoring algorithm is so low, it can also still be used without much of a detriment even if the actual algorithm running time turns out to be low as well. Additionally, our algorithm can be easily implemented in an automated system which runs a certain configuration of a solving algorithm on new problem instances continuously. Using our algorithm, the system can notice underperformance of the current configuration and automatically alert the user that reconfiguration may be needed.

The monitoring approach we introduced in this thesis is a lightweight and easy to implement solution to the problem of preventing algorithm underperformance over time. Although we have found potential flaws with the algorithm, such as a biased dataset decreasing prediction accuracy, we have also posited solutions to these problems, like carefully constructing the training dataset as in [Hutter et al., 2014]. The thresholding mechanism we introduced can be adjusted to fit various applications depending on the needs of the user. It allows the user to determine the amount by which the configured algorithm must outperform the default algorithm. The measure in which performance can differ varies greatly depending on context, so this flexibility adds to the versatility of our monitoring algorithm. During this thesis we have also found possible improvements and further research to improve our approach, which can be found in Section 8. We think this monitoring algorithm, possibly with our noted extensions and improvements, could improve the usage of solving algorithms and, as a result, that it can benefit the field of algorithm configuration as a whole.

8 Future Work

In this thesis we evaluated the basic implementation of a monitoring algorithm and it can be improved in several ways. We discuss the most promising ideas here.
If this monitoring algorithm is used within a Self-Monitoring, Automated Algorithm Configuration System, the system could use this program to automatically decide whether it should reconfigure its solver or not. This is a step towards fully automating solving algorithms which we think is not only desirable, but also feasible in the short term. This Configuration System could even be used for other problems like TSP or MIP, greatly expanding its applicability.

In this thesis, we based our performance threshold that the configured algorithm should outperform on the predicted running time of the default algorithm. However, since the default algorithm can have wildly varying running times and frequent time-outs, this metric can be nonoptimal. For example, the default could have a really low running time, like 1 second, while the optimized configuration might have a running time of two seconds on this same problem instance. In this case the monitoring algorithm would warn the user, even if the performance of both algorithms is really good. For that reason we suggest investigation into whether we can base our threshold on maximum performance. If this is calculable with relative ease, this approach would be a logical next step to our approach in this thesis. Nevertheless, depending on how this maximum performance data turns out, it might be difficult to directly base a threshold on it, but we suggest it could be an improvement.

As we noted, our prediction quality was not as good as in, for example, [Hutter et al., 2014]. Therefore we suggest a study similar to this thesis which implements a monitoring algorithm, but finds a dataset with less bias to build their RF on. This study could be set up similar to ours, which would make comparing results easier and in this way we could provide a stronger basis for our monitoring approach.

Lastly, we mentioned previously that this monitoring algorithm is very quick to use. We add to this that algorithm configurations are often single strings, so they take up little space if saved on a storage medium in, for example, a text or csv file. For these two reasons we suggest a user might want to save their unused optimized configurations to a file. Our monitoring algorithm could then be run on one problem instance with the current optimized algorithm, all previous ones and the default configuration to see which one is the fastest. Even if several hundreds of configurations have been saved, this action should take a few seconds at most based on execution times observed in this thesis. In this way the system can pick which of their saved configurations would be optimal to use on the problem instance that is being investigated. This approach should prevent unnecessary reconfiguration if a good enough alternative has already been found previously.

References

D. Babić and A. J. Hu. Structural abstraction of software verification conditions. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, pages 366–378, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73368-3.

D. Babić and F. Hutter. Spear theorem prover. In *SAT'08: Proceedings of the SAT 2008 Race*, 2008.

L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24605-3.

J. Hartmanis. Computers and Intractability: A Guide to the Theory of NP-Completeness (Michael R. Garey and David S. Johnson). *SIAM Review*, 24(1):90–91, 1982. doi: 10.1137/1024022. URL https://doi.org/10.1137/1024022.

F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In C. A. C. Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-25566-3.

F. Hutter, H. Hoos, and K. Leyton-Brown. Bayesian optimization with censored response data. *arXiv preprint arXiv:1310.1947*, 10 2013.

F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm Runtime Prediction: Methods & Evaluation. *Artificial Intelligence*, 206:79–111, 2014.

C. Luo, S. Cai, W. Wu, and K. Su. Proceedings of sat competition 2016 : Solver and benchmark descriptions. In *Proceedings of the 2016 SAT Competition*, pages 10–11. University of Helsinki, 2016. ISBN 978-951-51-2345-9.

J. Marques-Silva. Practical applications of Boolean Satisfiability. In *2008 9th International Workshop on Discrete Event Systems*, pages 74–80, 2008. doi: 10.1109/WODES.2008.4605925.

J. Schmee and G. J. Hahn. A simple method for regression analysis with censored data. *Technometrics*, 21(4):417, 1979. ISSN 0040-1706. doi: 10.2307/1268280.

K. van der Blom, H. H. Hoos, C. Luo, and J. G. Rook. Sparkle: Towards Accessible Meta-Algorithmics for Improving the State of the Art in Solving Challenging Problems, 2022. URL https://doi.org/10.1109/TEVC.2022.3215013.

L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

A split_files.py

```
import os
import shutil

sources = ['test.txt', 'training.txt']
destinations = ["./SWV_Test/", "./SWV_Train/"]
# creates the destination folders if not present
for destination in destinations:
    if not os.path.isdir(destination):
        os.mkdir(destination)
```

```
# takes all instances in the text files and moves them from
#   the 'instances' folder to two separate folders
sourc_dest = zip(sources, destinations)
for pair in sourc_dest:
    with open(pair[0]) as f:
        for line in f:
            shutil.copy(f"./{line}".strip('\n'), pair[1])
```

Listing 1: The full code of `split_files.py` which splits the single `instances` folder into `SWV_Train` and `SWV_Test`.

B File Server Batch Files

Below are the server batch files that run the configuration of the Spear SAT solver with SMAC through Sparkle. They can be used as follows:

`spear_with_features.sh` runs the configuration process with features and `spear_without_features.sh` runs it without them.

1. place both files in the root folder of Sparkle and run them
   Note: Since two configuration processes cannot be run in the same folder at the same time and the process takes two days, it is recommended to duplicate the Sparkle directory and run two processes parallel to each other.

2. the files will automatically find the SWV instance set and extract and compute its instance features. Then they will find the Spear solver, import it and configure it using SMAC.

3. when the configuration is done, run `collect_spear_[with/no]_features_data.sh` to run the validation process, generate the report and export it into a folder named `Spear_SWV_Data_with_without_feat` located in the same folder as the Sparkle directory. Other useful results are also automatically copied to the same folder so they can be extracted easily.

```
#!/bin/bash
Commands/add_instances.py --run-solver-later --run-extractor-later Examples/
    Resources/Instances/SWV_Train/
Commands/add_instances.py --run-solver-later --run-extractor-later Examples/
    Resources/Instances/SWV_Test/
Commands/add_solver.py --run-solver-later --deterministic 0 Examples/Resources/
    Solvers/spear/
Commands/add_feature_extractor.py --run-extractor-later Examples/Resources/
    Extractors/SAT-features-competition2012_revised_without_SatELite_sparkle/
Commands/compute_features.py --parallel
Commands/sparkle_wait.py
Commands/configure_solver.py --solver Solvers/spear/ --instance-set-train
    Instances/SWV_Train/ --use-features --settings-file Solvers/spear/
    swv_with_without_feat_settings.txt
```

Listing 2: The Server Batch File `spear_with_features.sh`. It runs SMAC through the Sparkle platform to configure the Spear solver on the SWV problem instance set using problem instance features.

```
#!/bin/bash
Commands/add_instances.py --run-solver-later --run-extractor-later Examples/
    Resources/Instances/SWV_Train/
Commands/add_instances.py --run-solver-later --run-extractor-later Examples/
    Resources/Instances/SWV_Test/
Commands/add_solver.py --run-solver-later --deterministic 0 Examples/Resources/
    Solvers/spear/
Commands/configure_solver.py --solver Solvers/spear/ --instance-set-train
    Instances/SWV_Train/ --settings-file Solvers/spear/
    swv_with_without_feat_settings.txt
```

Listing 3: The Server Batch File `spear_without_features.sh`. It runs SMAC through the Sparkle platform to configure the Spear solver on the SWV problem instance set without using problem instance features.

```
#!/bin/bash
Commands/sparkle_wait.py
Commands/validate_configured_vs_default.py --solver Solvers/spear/ --instance-
    set-train Instances/SWV_Train/ --instance-set-test Instances/SWV_Test/
Commands/sparkle_wait.py
Commands/generate_report.py
mkdir ../Spear_SWV_Data_with_without_feat/with_features/
cp -dr ./Components/smac-v2.10.03-master-778/example_scenarios/spear_SWV_Train/
    outdir_train_configuration/spear_SWV_Train_scenario/state-run* ../
    Spear_SWV_Data_with_without_feat/with_features/
cp Configuration_Reports/spear_SWV_Train_SWV_Test/Sparkle-latex-generator-for-
    configuration/Sparkle_Report_for_Configuration.pdf ../
    Spear_SWV_Data_with_without_feat/with_features/
    Sparkle_Report_for_Configuration_With_Features.pdf
```

Listing 4: The Server Batch File `collect_spear_SWV_with_features_data.sh`. It runs the validation of the found configuration then generates the report and exports the necessary data to the `/Spear_SWV_Data_with_without_feat/with_features/` folder. To be used with `spear_with_features.sh`.

```
#!/bin/bash
Commands/sparkle_wait.py
Commands/validate_configured_vs_default.py --solver Solvers/spear/ --instance-
    set-train Instances/SWV_Train/ --instance-set-test Instances/SWV_Test/
Commands/sparkle_wait.py
Commands/generate_report.py
mkdir ../Spear_SWV_Data_with_without_feat/
mkdir ../Spear_SWV_Data_with_without_feat/without_features/
cp -dr ./Components/smac-v2.10.03-master-778/example_scenarios/spear_SWV_Train/
    outdir_train_configuration/spear_SWV_Train_scenario/state-run* ../
    Spear_SWV_Data_with_without_feat/without_features/
cp Configuration_Reports/spear_SWV_Train_SWV_Test/Sparkle-latex-generator-for-
    configuration/Sparkle_Report_for_Configuration.pdf ../
    Spear_SWV_Data_with_without_feat/without_features/
```

```
    Sparkle_Report_for_Configuration_No_Features.pdf
```

Listing 5: The Server Batch File `collect_spear_SWV_no_features_data.sh`. It runs the validation of the found configuration then generates the report and exports the necessary data to the `/Spear_SWV_Data_with_without_feat/without_features/` folder. To be used with `spear_without_features.sh`.

C Using the Random Forest and Monitoring Algorithm

To construct the RF, we need to use several files, which we explain here.
Data processing is done in `prepare_rf_data.py` in the function `prepare_data`. The data is collected and matched from 25 folders, one for each configuration run, and each of these folders contain three files that we use. The files that start with `runs_and_results` contain the algorithm running times, the files that start with `paramstrings` contain the unique algorithm configurations and the files called `instance-features.txt` contain all problem instances together with their features. There are multiple of each of these files, so we use the one with the highest iteration number. This is because the file with the lowest number is repeated in its entirety in the next higher file, so the file with the highest number contains all the data we need to collect. All running times are then 10-log-transformed since the running times can vary greatly in these problems and this should increase our prediction accuracy [Hutter et al., 2014].
`runtime_predictor.py` contains our custom `RegressionForest` class. If we run the `cross_val_all_configs` function, it automatically runs all configurations of the RF and evaluates their performance for Experiments B, C and D (see Section 5). It collects all data using the `prepare_data` function mentioned above. If we call the `build_and_export_forest` function, it builds an RF based on the optimal settings discussed in Section 5 and exports it to the `/Saved_Data/` folder which is created in the same directory as the file if it is not found. We build the RF and export it using Pickle. So that the object does not get corrupted, Pickle requires the user to use the same or close versions of Pickle and Scikit-Learn. The version of Pickle we used is 4.0 and our Scikit-Learn version is 1.1.1.
Using this exported RF, we build `performance_monitor.py`. This program can read the saved RF and use it to predict running times of previously unseen problem instances. It does this by taking the parameters from an algorithm configuration, as the configuration is presented in the report (`--parameter 'value'`). It then appends these parameters to the problem instance features in the way the RF expects by using the `construct_data` function from `create_test_set.py`. This also finds the running times on those algorithms from the `sparkle_performance_data_sorted.csv` files stored in the `/performance_data_[configured/default/configured_no_features]` folders. Since not all performance data is saved in the same order when it is extracted, please run `order_performance_csvs.py` before this. The program then generates the data used in Experiments E and F.
Run `plot_runtimes.py` to generate the plots of the set used to train the RF and the set used to test the monitoring algorithm (see Sections 4 and 6 respectively), as well as the average running times of DEF, WF and NF (Sect. 4) on the full `SWV_Train` and `SWV_Test` sets.