Karlsruher Institut für Technologie

Bachelor thesis

# An Explainable Portfolio of Specialized Kissat Configurations

Raphael Zipperer

Date: 5. Mai 2025

Supervisors: Dr. Markus Iser
Prof. Dr. Peter Sanders

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

# Abstract

The Boolean Satisfiability Problem (SAT) is fundamental in both theoretical and practical applications across the NP domain. Since no single SAT solver excels on all instances, algorithm portfolios composed of complementary configurations have become a promising approach. This thesis presents an end-to-end method for constructing such a portfolio using configurations of the CDCL solver Kissat, the clear winner of the most recent SAT competition. We investigate strategies for labeling instances, selecting diverse configurations, and training classifiers to match instances to solvers. Our results show that labels based on the original NP problem an instance is derived from significantly outperform clustering-based methods. Additionally, we highlight the trade-off between portfolio size and classifier performance, offering insights into effective SAT portfolio design.

# Acknowledgments

Large Language Models (LLMs) were employed in this work for tasks such as grammar checking, rephrasing, code validation, and generating simple code for tasks related to data management and graphics. However, LLMs were not used for directly generating text or conducting research on the topics discussed in this thesis.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, Datum

# Contents

# 1 Introduction

## 1.1 Motivation

The Boolean Satisfiability Problem (SAT) is central when it comes to research on the NP domain. As all NP problems can be encoded into an equivalent SAT representation, it can be applied to a variety of theoretical and practical applications such as hardware verification, scheduling and cryptography. How to efficiently solve SAT instances is thus a significant question in past and contemporary research.
As the field of SAT Solving developed, different approaches emerged, many of which are complementary, meaning that one approach might be very well suited on a subset of SAT instances, but inferior to other approaches on different subsets. As a result, selecting the approach most likely to perform well on a per-instance basis became a hot topic of contemporary research.
The problems of instance grouping has been tackled in a number of ways, ranging from objective oblivious feature clustering as in ISAC [14] to runtime clustering as in SNNAP [9].
We aim to provide a direct comparison of different approaches to build an algorithm portfolio. In the process, we will present an end-to-end approach, starting by acquiring and selecting configurations, then relabeling data and finally training a classifier. The questions we pose regarding our work are as follows:

- How may instances be grouped effectively? Does the original problem that a SAT instance is derived from play a major role for an effective grouping?
- How do the selected configurations complement each other?
- How does the number of configurations affect classification performance? Is there an especially advantageous number of configurations to be used?

## 1.2 Contribution

Currently, the single SAT solving approach with the best average performance is Conflict Driven Clause Learning (CDCL). Kissat, the winner of numerous recent SAT competitions, also belongs to this class of SAT solvers. In this work, we aim to create a portfolio of Kissat configurations as well as a well-suited classification method. We present an end-to-end approach to construct a configuration portfolio and explore different methods of

labeling and classifying instances. We analyze our final portfolio and interpret the resulting performance on a large holdout set.

## 1.3 Structure of Thesis

In chapter 2, we introduce the SAT problem as well as currently successful solving approaches. Additionally, we touch on some of the tools we used in this thesis. In chapter 3, we describe our approach, starting with the acquisition of configurations we may use, then talking about design choices and how they may be combined in an algorithm selector. In chapter 4, we talk about the datasets and metrics used in this thesis, as well as about implementation details. Afterwards, we will discuss evaluation results and compare different approaches in section 5. Additionally, in sections 5.3 and 5.4, we will train a final configuration selector based on the insights we gained from the evaluation. Finally, we aim to give an overview of our insights and an outlook on possible future work in chapter 6.

# 2 Fundamentals

## 2.1 General Definitions

### Boolean Satisfiability Problem

We define the Boolean Satisfiability Problem (SAT) in its Conjunctive Normal Form (CNF) [1]:
Given is a set of **variables** $V = \{x_1, ..., x_n\}$ and their negations $\overline{V} = \{\bar{x}_1, ..., \bar{x}_n\}$. Variables are connected within so called **clauses**, which is a disjunction of a subset of variables and their negation: $c_i = \bigvee_{v \in (V \cup \overline{V})} v$. The clauses are conjoined with each other, yielding the SAT formula $F = \bigwedge_{c \in Clauses} c$. SAT concerns itself with the question if a given formula is solvable, meaning that there exists an instantiation of the variables such that the formula evaluates to true. This problem is NP-complete, meaning that there exists no known algorithm to solve a random problem instance in less than $O(2^{|V|})$. Another defining feature of NP-complete problems is that any instance of one problem can be transformed into an equivalent instance of another NP-complete problem via a polynomial-time reduction function [15].

### Conflict Driven Clause Learning

Conflict Driven Clause Learning (CDCL) is one of the currently most successful approaches to tackling the SAT problem [18]. It relies on recursively assigning variables until a conflict occurs. Conflict in this case means that the current state of assignments implies opposite assignments for another variable, meaning that the formula is unsatisfiable under the current partial assignment. Conflicts are resolved by backtracking ("unassigning" variables) and adding an additional **conflict clause** which does not change the satisfiability of an assignment on the original formula, but prevents the solver from creating the same conflict again.

### Algorithm/Configuration Selection Problem

Algorithm selection [17] is a strategy utilized on problems where different approaches are optimal on different subsets of the problem instances. In practice, this means that an algorithm selector consists of a portfolio of algorithms as well as a classifier component, which ranks or selects algorithms for a given instance at runtime. To gain performance

through an algorithm selector, the algorithms must be complementary, meaning that each algorithm performs strongly on a subset of instances that other approaches struggle with.

### Hyperparameter Optimization

Hyperparameter optimization is the process of selecting the best set of hyperparameters for an algorithm to improve its performance. Hyperparameters are configuration settings that are not learned (e.g., learning rate for machine learning models) but significantly impact the algorithms effectiveness. Optimization techniques, such as Bayesian optimization, are used to systematically explore and identify the optimal combination of hyperparameters for a given task.

### K-means Clustering

Clustering is an unsupervised learning approach, in which a set of objects is divided into multiple clusters. Objects within a cluster are aimed to have a higher similarity to each other than to objects outside a cluster under a chosen metric (e.g. euclidian distance or cosine similarity). K-means Clustering is a clustering algorithm in which the number k of clusters to be extracted is given beforehand [13].

### Kissat

Kissat [8] ("**K**eep **I**t **S**imple and clean bare metal **SAT** solver") is a CDCL SAT solver, based on a reimplementation of the SAT solver CaDiCaL [5] in C. Implementing local search approaches into its CDCL approach, Kissat achieved a spike in solver performance upon its release in 2020. It is currently the best performing SAT solver on many instance categories and the winner of recent SAT competitions. It has fared especially well on the most recent SAT Competition 2024, after the introduction of Clausal Congruence Closure [7] in Kissat [6].

### Global Benchmark Database (GBD)

GBD [12] is a database containing SAT instances and their metadata. All instances of the most recent annual SAT competitions are included. Besides metadata, many instances come with pre-extracted features as well.

### Smac3

SMAC3 [16] is a tool for hyperparameter optimization that leverages Bayesian optimization. It models the cost function probabilistically over the parameter space, using a surrogate model that becomes more accurate as SMAC gathers additional samples. At each

iteration, it maintains an incumbent—the best-performing configuration identified so far. SMAC also supports optimization across various user-defined metrics, including the ability to optimize for multiple objectives simultaneously.

## 2.2 Related Work

### ISAC

Instance Specific Algorithm Configuration (ISAC) [14] is a configuration selection algorithm used in the context of SAT solving. Based on a representative set of instances, it clusters the instances based on instance features. The clustering method used is g-means, which recursively applies 2-means clustering on the current set of clusters until the clusters follow a roughly gaussian distribution. The clustering is objective-oblivious, which is an approach that will be employed in our work as well.
It then uses the parameter tuning algorithm GGA to optimize configurations over the extracted clusters.

### SNNAP

Solver-based Nearest Neighbor for Algorithm Portfolios (SNNAP) [9] is another clustering approach, which additionally takes past performances of solvers on instances as part of the feature vector. It encodes the performance of an algorithm on a given instance in a bitvector, setting the two or three best performing solvers to '1' while the rest is set to '0'. This bitvector is then weighed and used in addition with the other features. For instances that have not been evaluated yet, SNNAP predicts the runtime for each solver to predict the bitvector. While this approach requires more training data than other appoaches like ISAC, it provides a powerful, objective non-oblivious alternative.

### SATzilla

SATzilla [19] is an automated approach to constructing an algorithm portfolio based on a training set and a set of algorithms. It first runs a selection of solvers (pre-solvers) with a low timeout. Then, it extracts the instance features and predicts the best solver based on an empirical hardness model, which is constructed in the training phase and predicts solver runtime. We will not rely on predicting runtime in our work, instead employing decision trees to directly predict configuration from comparably small portfolios.

# 3 Kissat Portfolio Synthesis

The process of crafting an algorithmic portfolio can be split up into a few stages.
For starters, we need to identify a limited number of configurations that should be considered in portfolio creation, as calculating the performance of a single configuration across the whole training set is costly in itself. Initially, we are not aware which instances require complementary configurations. It is thus necessary to split the dataset into groups, on which the hyperparameter optimization may be performed. The resulting configurations may not all necessarily be complementary, but as stated before, it is not possible to know this without acquiring data first. The number of configurations may be reduced in later stages of the algorithm as needed.
Secondly, we need to consider which subset of said configurations to use in the final portfolio, as decreasing the portfolio size may increase classifier performance. As we will see, the question which configurations should be taken into account is closely related to the labeling of the data. Thus, the components of the algorithm are interconnected. In this section, we will investigate various approaches to data labeling and configuration selection, with the aim of evaluating the most effective combination.

## 3.1 Configuration Acquisition

Before starting to train the classifier, we need to determine which subset of Kissat configurations may be a good fit for the portfolio. Optimally, we would find a set of strong configurations which are pairwise complementary. The approach we decided on was to define a set of groups, in which instances should behave roughly similar. In our case, it was possible to exploit preexisting database labels representing the original problem an instance is derived from for this preliminary grouping, though different ways of grouping instances may be explored in future work. We then optimize over each of the groupings using SMAC. Logging the performance of each configuration over individual instances, we may extract multiple powerful configurations from a run over a single group, if they are complementary on the group. This only makes sense if both configurations perform best over a number of instances. Additionally, the instances the configurations perform strongly on should preferably be easily separable, as this makes the task of classification easier. We thus decide to only perform binary splits on groups using a simple splitting point on one of the instance features. Additionally, we apply a threshold to ensure that each group formed after the split contains at least a minimum number of instances. This algorithm may be

recursively called on the two groups the split results in to split further.

**Algorithm 1:** Splitting the groups

```
1  function split(group, minsize, minmargin):
2  bestsplit ← None
3  bestperformance ← PAR2(SBS(group))
4  for distinct c_1, c_2 in configurations do
5      for f in features do
           // For the two configurations, finds the best splitting point on f, such that all
           // instances with f < split get assigned c_1, all others c_2
           // Also returns the PAR2 score of the new assignment
6          split, performance ← optimal_split(f, c_1, c_2, minsize)
7          if performance < bestperformance then
8              bestperformance ← performance
9              bestsplit ← split
10 if bestperformance / PAR2(SBS(group)) < minmargin then
11     return bestsplit
12 else
13     return None
```

## 3.2 Classification

The goal of this thesis is to select a powerful portfolio of configurations and to train a classifier that can predict the optimal configuration for a given instance. Given the configurations acquired using the algorithm of the last section, we have a rough idea which configurations perform well on single groups. There is however no guarantee that these configurations are complementary - two different groups may yield slightly different best configurations in the SMAC optimization process, which still perform similarly across the training instances. Intuitively, more configurations in the portfolio would make the classification task harder. Thus, cutting down on configurations may make sense.
Furthermore, the performance of a configuration is not yet known on the whole training set, as SMAC does not explore the same configurations or each group. We thus compute the performance of each configuration over the whole training set, giving us the ability to eliminate configurations that may behave similarly and ensuring complementarity between configurations in the portfolio. As we will see, the interaction between how the data is grouped and which configurations are chosen is crucial to the performance of the portfolio. This section will examine three labeling methods and three configuration selection strategies, which can be combined in any manner.

### 3.2.1 Label selection

This subsection concerns itself with the grouping of instances. The way that instances are labeled changes the performance of configurations on each label. As such, testing different ways of labeling the data is an important part of this thesis.

#### Database Labels

The database used may include predefined labels indicating which NP-hard problem each SAT instance corresponds to. In our case, the instances were organized based on the original NP problem from which the SAT instances were derived, providing a solid initial labeling.

#### Clustering by features

An alternative approach to relabeling the instances is to cluster them based on their features in the database. We will experiment with K-means using various parameters to identify a clear and easily classifiable labeling of the instances. Clusters tend to group similar instances together, which can simplify classification by highlighting underlying patterns and reducing overlap between classes.

#### Clustering by best configurations

Clustering can also be performed using different metrics. Another idea is to generate a list of strong configurations for each database label. By applying K-means, we extract the most effective cluster of configurations over each label, which can then be encoded as a bitvector. Labels are subsequently clustered based on the similarity of their bitvectors, resulting in groupings that exhibit similar behavior across configurations. This approach is objectively non-oblivious and is inspired by SNNAP [9]. It is important to note that in this method, the bitvector distance operates within the default label space rather than the instance feature space—though alternative initial groupings can also be used.

### 3.2.2 Configuration Selection

This section addresses the question of which configurations should be selected for use in the classifier. Notably, the choice of suitable configurations is closely tied to the labeling of the data, creating a cross-product relationship between labeling and configuration selection. A key trade-off arises in determining the number of configurations to include: increasing the number of configurations generally improves the performance of the Virtual Best Solver, potentially allowing the classifier to achieve significantly better performance on certain instances. However, this also increases the complexity of the classification task, which

may in turn reduce the overall performance of the algorithm selector. The objective of this section is therefore to propose methods for identifying a "reasonable" number of strong and complementary configurations, where the precise interpretation of "reasonable" is guided by the characteristics of the available data.

### Label-best configuration

A naive way of selecting good configurations is simply assigning each label the best configuration from the pool of computed configurations over all of its instances. While this method is intuitive, it does not tackle some of the issues discussed above. For instance, complementarity is not ensured by this approach. Furthermore, the number of configurations used can be relatively high, leading to bad classifier performance. Still, having a higher number of configurations might also increase performance, making this approach worth looking at, be it only as a metric for more advanced approaches.

### Beam search

When considering the entire pool of configurations, some may offer only marginal improvements in virtual performance while potentially hindering classifier effectiveness. To address this, configurations can be added iteratively, with the aim of enhancing solver performance by selecting complementary configurations. Although a greedy search strategy could be employed for this purpose, it is prone to suboptimal selections due to its limited scope. Therefore, we opt for a beam search approach, which maintains multiple candidate sets during the search process and is thus better suited to identifying more globally effective configuration subsets. Beam search works quite well for the algorithm selection problem in the algorithm selection context, yielding near-optimal solutions [2].

**Algorithm 2:** Beam search

```
1  function beamsearch(n, beam):
2  newbeam ← []
3  for each L in beam do
4      for each configuration c not in L do
5          Create new list Subset_c = L with configuration c added to set
6          Append Subset_c to newbeam
   // Sort by VBS PAR2 score
7  newbeam ← sort(newbeam)
   // Only keep the n best results
8  beam ← newbeam[:n]
9  print(beam)
   // Recursively call beamsearch again
10 beamsearch(n, beam)
```

In this implementation, the function sort sorts the beam by scoring the dataset by VBS PAR2 score 4.5. A significant increase in VBS performance upon adding a configuration to the portfolio suggests that the configuration performs better on a subset of instances than the existing configurations, indicating that it is complementary to them. The algorithm is called recursively and outputs the beam at every depth. We later analyze classifier performance given the computed portfolios to determine the best tradeoff between classifier accuracy and portfolio size.

### Minimum hitting set

Looking at a given label, there might be multiple configurations that score fairly well, or there might be one or a few configurations that strongly outperform the rest. We use k-means clustering over runtime performance to determine which configurations are close in performance over a given label. Increasing k potentially yields smaller top clusters, thus making the configuration selection more strict. This leads to the question which minimal selection of configurations would be needed so that at least one of the best performing configurations of each label is contained. This is a minimum hitting set problem, which is in itself a known NP-hard task. As we are dealing with a relatively small problem size, it is sufficient to solve the minimum hitting set problem by enumerating all possible sets in order of size in a brute force manner. To enumerate hitting sets of a given size, we use the combinations function of the itertools python library.

**Algorithm 3:** Minimum hitting set

```
1  function minhitset(labels, k, configurations):
   // Extract top cluster for each label
2  bestconfs ← []
3  for each label in labels do
4      topcluster ← kmeans(runtimes(label), k)[0]
5      Append topcluster to bestconfs
   // Collect configurations that have to be contained in any hitting set
6  backbone ← []
7  for each confs in bestconfs do
8      if |confs| == 1 then
9          Append confs[0] to backbone
10     else
11         continue
12 minimumHittingSet = Null
   // Attain the smallest hitting set by brute force
13 remainingConfs = configurations \ backbone
14 for each n in |remainingConfs| do
       // All combinations containing n elements
15     for each combination in Combinations(remainingConfs, n) do
16         if isHittingSet(backbone ∪ combination) then
17             minimumHittingSet = backbone ∪ combination
18             break
19 return minimumHittingSet
```

### 3.2.3 Classifier

We use two different classification approaches, depending on the labeling method used. For the feature independent labeling methods, which are runtime clustering and the default database labels, we use random forests. Random forests have proven useful in SAT instance classification and are more suited to the amount of data at hand than more verbose classification methods. For the k-means clustering approach, choosing a classification approach is even simpler, as we can simply evaluate over the extracted centroids on the training data. We always first label the data using one of the methods described above. Then, we select the configurations to be used in the portfolio. We then select the LBS 4.5 from the portfolio for each label. Finally, we train the classifier to directly predict the configuration.

# 4 Experimental Design

In this chapter, we provide a concise overview of the design of our experiments. We present the datasets used, highlight key implementation details, and outline the metrics employed in our evaluation.

## 4.1 Dataset

There are two datasets used in this thesis. The first one comprises the main tracks of the SAT competitions of the years 2023 [3] and 2024 [10] and provides the train/test set, which we will call the training set in the following for simplicity's sake. Furthermore, we use the anniversary track of the 2022 SAT competition [4] (excluding instances that are also part of the before mentioned datasets) as the holdout set. This dataset will be called the holdout set or the anniversary track in the following. The features used are provided by GBD. For classification, we used the base database features, including information about the instances like how many horn clauses are included in its conjunctive normal form. A comprehensive list of the features used can be found here A.1.

## 4.2 Implementation

The entire implementation is written in Python, an overview as well as the github repository can be found here A.5. For a full list of the relevant libraries used, see the appendix A.4.

We use two machines on our cluster, which are named after Turing Price winners.

| **Name** | **Sockets** | **CPUs/Socket** | **Threads/CPU** | RAM (MB) |
|---|---|---|---|---|
| Liskov | 2 | 64 | 2 | 2064052 |
| Floyd | 2 | 32 | 2 | 3096095 |

**Table 4.1:** Specs of the machines used in this thesis

Liskov was used for the entirety of the SMAC optimization stage as well as all evaluations on the holdoutset, both with 32 parallel threads.

After we decided on the configurations we wanted to use for the portfolio, we evaluated all of said configurations on the entire training set, using Floyd with 64 parallel threads.
For optimization, we use all Kissat top level parameters (meaning that they are not dependent on any other parameter). A comprehensive overview of the parameters can again be found in the appendix A.2.

## 4.3 Configuration acquisition

To create the configuration portfolio, we first needed to find configurations which fit the training instances. To create a baseline of powerful instances, we first did a run of 50 iterations of SMAC on each of the groups, capping the amount of instances to optimize over at 32 if the group size exceeded this limit. Additionally, we evaluated the default configuration once on all training instances, resulting in a total of 51 configurations per group. As SMAC evaluated the groups independently, not every configuration learned in a group would also be learned in the other groups. On each of the groups, we then recursively applied 1 with the threshold 8 on minimum size and a 0.9 upper bound on $\frac{PAR2(SPLIT(G))}{PAR2(SBS(G))}$.
The initial label groupings can be found in the appendix A.3.

The resulting 40 configurations A.6 provided our baseline configurations. While more configurations may be acquired from the aforementioned training process, computing the runtimes on one configuration over all instances is expensive.

## 4.4 Model comparison

We always use train-test splits stratified by the database "family" labels. For families that contain less than 5 instances, we create a new clutter label under the name "rest" to make stratification possible. The share of train-test splits is always 80:20. For approach performances (boxplots, averages, etc) we always evaluate over 10 samples, unless stated differently. If no metric is given in a plot, we always report PAR-2 score 4.5.

## 4.5 Evaluation Metrics

### PAR2 score

All classifier results are scored according to the penalized average runtime scheme. PAR2 is a metric commonly used in SAT competitions [11]. The scheme assigns a runtime of two times the timeout to instances that could not be solved in time. PAR2 on one instance is calculated as follows:

$$t_{par2} = \begin{cases} \text{runtime, if timeout was not reached} \\ 2 * \text{timeout, otherwise} \end{cases}$$

For a set of instances $I$, we simply take the average:

$$\text{PAR2}(I) = \frac{1}{|I|} \Sigma_{i \in I} t_{par2}(i)$$

### Virtual Best Solver

The Virtual Best Solver (VBS) performance of a portfolio is the theoretical case in which for any instance, the optimal solver is chosen. While VBS is practically not attainable, it is a valuable metric to understand the potential performance of a portfolio. The PAR2 score of VBS is calculated as follows:

$$\text{VBS}(I) = \frac{1}{|I|} \Sigma_{p_i \in I} \min_{s \in S} Par2(s, p_i)$$

### Single Best Solver

The Single Best Solver (SBS) is the solver with the best performance over an instance set. The PAR2 score of the SBS is calculated as follows:

$$\text{SBS}(I) = \frac{1}{|I|} \min_{s \in S} \Sigma_{p_i \in I} Par2(s, p_i)$$

In this thesis, we will often use the Default configuration of Kissat, rather than the SBS. This is because in all train-test splits, the default configuration was the SBS by a large margin. It will thus provide a baseline for our approaches.

### Label/Cluster Best Solver

For a given labeling, the Label Best Solver (LBS) selects the SBS on each label. It is thus a more coarse approximation of VBS, averaging performance over sets of instances rather than individual instances. This solver may sometimes be called Cluster Best Solver in this work, as some of our labeling methods are based on clustering.

# 5 Experimental Evaluation

In this chapter, we present our evaluation results. In the first section, we evaluate the different methods for labeling and choosing configurations presented in 3.2, trying to find the most successful approach. In the next section, we train a classifier using the dominant approach from the first section and analyze the resulting portfolio. In the last section, we finally evaluate the classifier on the holdout set.

## 5.1 Comparing classification techniques

As stated before, there is a cross product relationship between data labeling and configuration selection. In this section, we evaluate different combinations as well as hyperparameters related to our approaches. Hyperparameters may have direct effects on labeling or configuration selection and thus on performance. For example, we will see how different portfolio sizes affect performance, and that the earlier mentioned trade-off between portfolio size and classifier performance is relevant for our minimum hitting set and beam search approaches. Note that we only tune parameters directly relevant to our approaches and not hyperparameters of the classification methods, which may be explored in future work.

### 5.1.1 Default database labels

We use the default database labels of the main track. Labels covering less than five instances are assigned to the new clutter label "rest" instead to make stratified train/test splitting possible. This is the most intuitive way of labeling the training data, and we will see that it works quite well for our task.

#### Label best solver

We begin by assigning each label its corresponding LBS, followed by training a Random Forest classifier using the instance features as inputs and the assigned labels as targets.

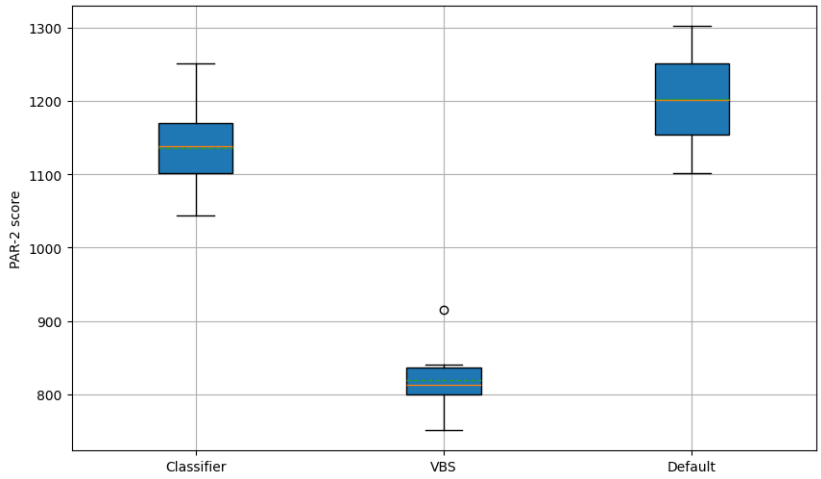



**Figure 5.1:** Default Label LBS based classifier, compared to VBS and Default performance

With this method, we achieved a slight improvement over the default configuration 5.1. However, a considerable gap remains between the performance of our classifier and that of the Virtual Best Solver (VBS). This discrepancy arises from the fact that no effort was made to reduce the number of configurations, potentially leading to a redundant and overly large portfolio.

### Minimum hitting set

We acquire the accumulated runtimes for every configuration over every label. We then use k-means clustering to group configurations by runtime, yielding a minimum hitting set problem we can solve using the algorithm described in 3. As stated before, the value of k influences the "strictness" of the configuration, with a higher value of k yielding a smaller cluster of configurations that are considered good, making it necessary to select more configurations to satisfy the implied minimum hitting set problem. This can be confirmed by looking at a few different values of k (Figure 5.4).

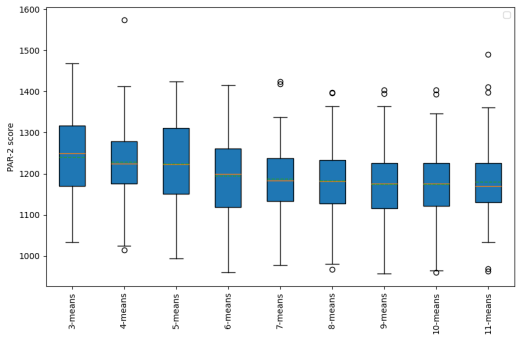



**Figure 5.2:** Boxplot of classifier performance on the test set, given k (50 samples)

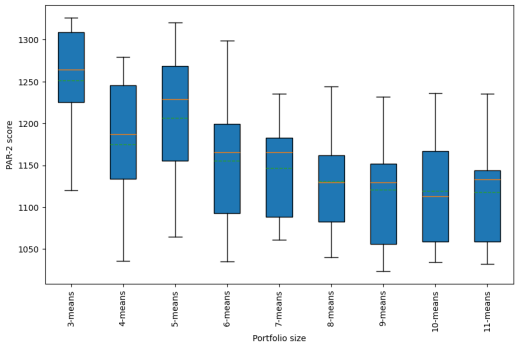



**Figure 5.3:** Boxplot of classifier performance on the test set, given k (10 samples)

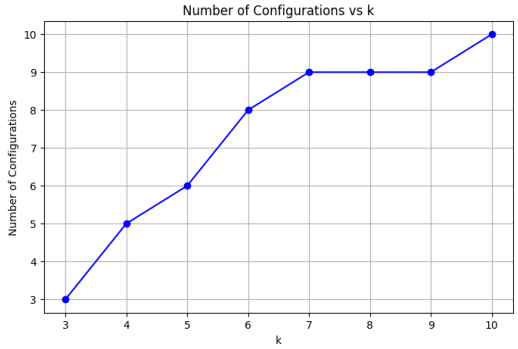



**Figure 5.4:** Exemplary size of minimum hitting set, given k

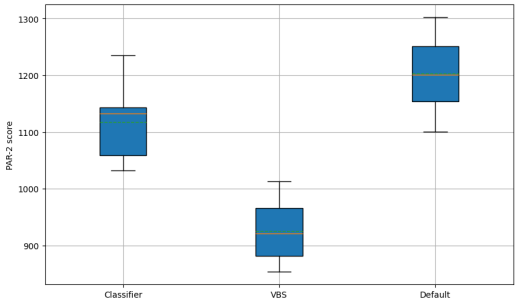



**Figure 5.5:** Minimum hitting set classifier with k=11, compared to Default and VBS performance

Initially, we observe a roughly linear increase in portfolio size as k increases, which then levels off at around k=7. The VBS performance seems to improve monotonously for higher k when looking at our experiment with 10 samples (Figure 5.3), which we will use as reference when comparing this approach to other approaches. As this approach performed quite well however, we reevaluated it with 50 samples (Figure 5.2), revealing a minimum at k=9. This indicates that there is indeed an optimal portfolio size, after which the VBS performance gain achieved by adding more configurations is nullified by an increasing number of misclassifications. The optimal value for k thus seems to be roughly nine in this experiment.

With this approach (Figure 5.5), we achieved a slight improvement compared to the LBS approach (Figure 5.1). The VBS displays noticeably worse performance results than in the LBS approach, which is simply explained by the fact that we reduced the number of configurations significantly in this approach.

**Stratified train-test split without outliers**

As stated before, we work with stratified train-test splits, assigning the rest label to instances that are part of labels with four or less instances. This means that the rest label is inhomogeneous however, as it contains a large variety of problem classes, often only represented by one instance. We thus want to take a quick look at classifier performance without the rest label, i.e. only considering classes that contain at least five instances.

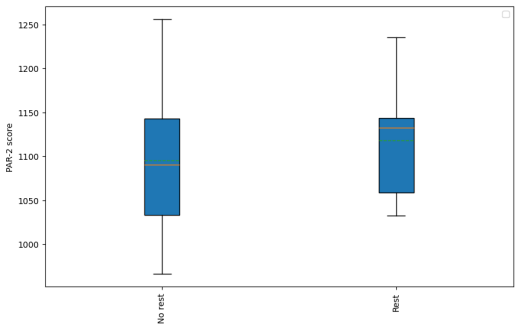



**Figure 5.6:** Minimum hitting set with parameter k=11, with and without rest instances

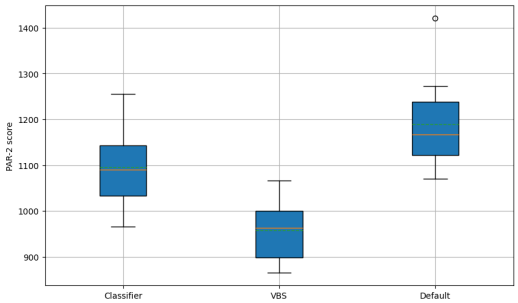



**Figure 5.7:** Minimum hitting set without rest, compared to SBS/VBS

We see an improvement in PAR2 of about 30 seconds (Figure 5.6). While the Default configuration (Figure 5.7) does have a slightly better PAR2 score as well, the improvement is most likely not significant enough to account for this increase in performance. Our classifier thus seems to scale better when all classes contain a sufficient number of representatives.

## Beamsearch

We follow the algorithm 2. As in all other approaches, we use a family stratified train test split. Since we use the same stratified train-test splits for every labeling approach and VBS performance is only dependent on the individual instances contained in the training set, this approach always finds the same configurations independent of labeling. We see an approximately exponential improvement in portfolio VBS performance, starting with large marginal improvements of almost 10% per added solver, then quickly dropping below 1%.

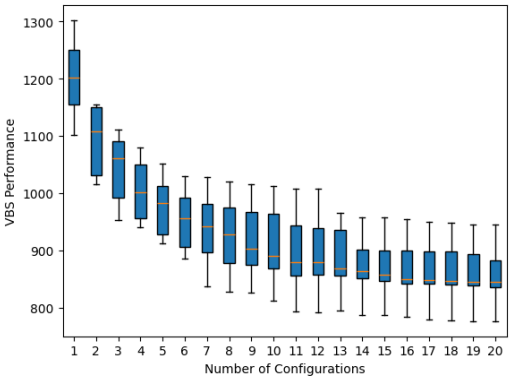



**Figure 5.8:** Boxplot of VBS performance on the test set, given portfolio size

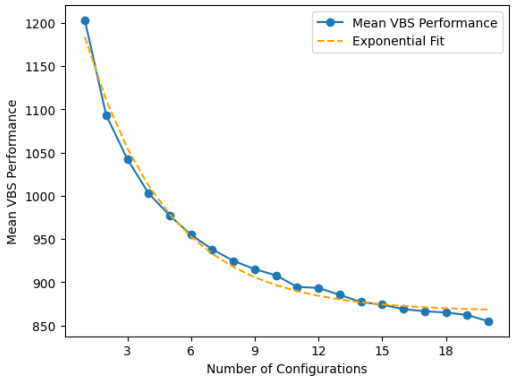



**Figure 5.9:** Mean VBS performance on test set, compared to an exponential function

The labeling approach starts playing a role when training a classifier, i.e. it fixates groups of instances which will always get assigned the same configuration under any classifier trained.
We train Random Forest Classifiers based on the configurations acquired via beamsearch above, using the same train-test splits.

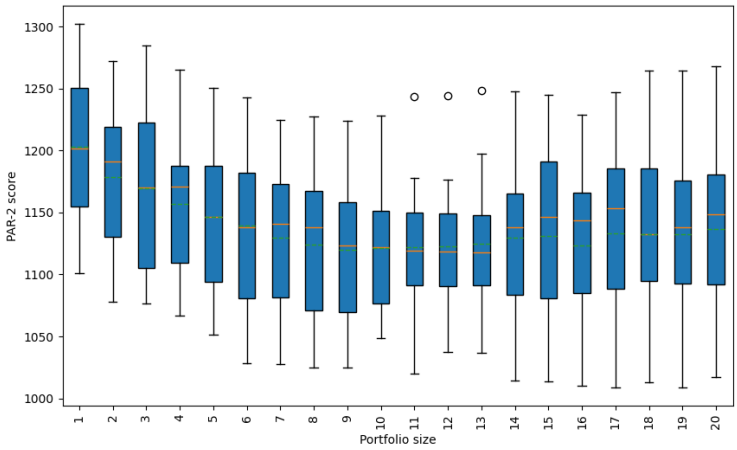



**Figure 5.10:** Beamsearch random forest classifier performance, with increasing portfolio size

As expected, there is a sweet spot for classifier performance when it comes to the portfolio size (Figure 5.10). Again, an increase in configurations initially leads to significant marginal improvements in PAR2 solver score. This trend continues until the portfolio reaches the size of nine configurations, at which point the mean performance slowly decreases again. As adding more configurations always leads to increasing (or at least constant) VBS performance, the loss in performance can only be explained by an increasing

number of misclassifications. This confirms our hypothesis that there is a tradeoff between high VBS performance and high classifier performance.
The performance of the optimal portfolio size scores slightly worse than in the minimum hitting set approach, although not in any statistically significant manner. The best minimum hitting set parameter scores at a mean PAR2 score of 1117, while the best beamsearch run scores at 1119. Ultimately, both approaches seem to perform roughly similarly well. While there is no clear winner, we still decide to use the minimum hitting set approach for future reference.

### 5.1.2 Clustering by features

We normalize the database features using the formula

$$\text{norm(feature)} = \{\tfrac{x-\mu}{\sigma} | x \in \text{feature}\}$$

where $\mu$ is the mean across the feature and $\sigma$ is the standard deviation over the given feature. Then, we apply k-means clustering over the entire training set. The clusters are our new labels, on which we will again evaluate the aforementioned methods of selecting configurations.

#### Cluster best solver

Firstly, we must establish which order of magnitude is best suited as our number of clusters k. Exponentially increasing the parameter k shows that a higher number of clusters generally seems to correlate with a good test performance up to a relatively high number of clusters (fig. 5.13). However, this method is prone to overfitting, as many such clusters only contain a single instance, which is acceptable for a few outliers but not for the majority of instances. We thus decided to select k from the neighborhood of 32, as these cluster sizes still yield a substantial increase in PAR-2 score while most clusters maintain a meaningful size. Linearly increasing k, we however do not see a drastic improvement in PAR2 score (Figure 5.14)

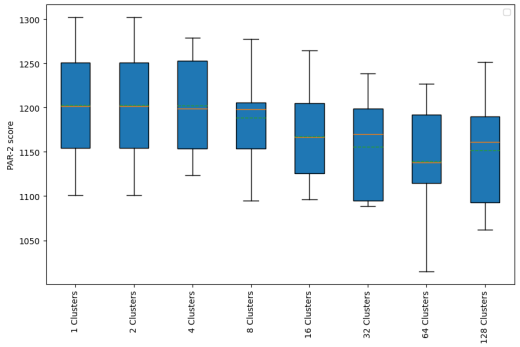



**Figure 5.11:** Exponentially increasing k

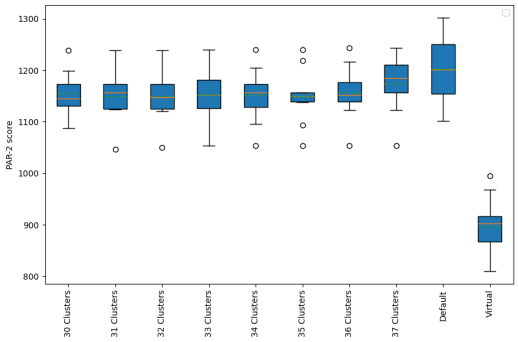



**Figure 5.12:** Linearly increasing k, with Default and exemplary VBS performance distribution

### Minimum hitting set

The idea of computing a minimum hitting set works label independent. As such we can first relabel the data by clustering as before, then reduce the portfolio size by applying the minimum hitting set approach 10. We choose $k_1 \in [20, 30]$ for $k_1$ the number of clusters the instance set is clustered to as well as $k_2 \in \{5, 10\}$ as the parameter for the minimum hitting set problem, as these values of k have been proven to work well in 5.1.2.

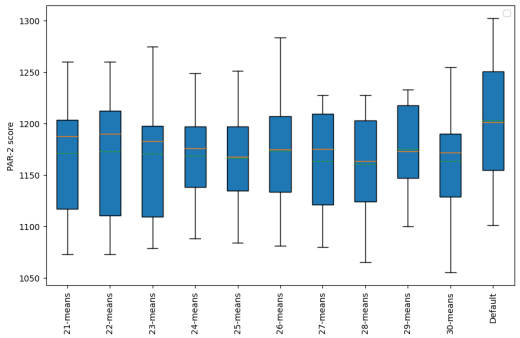



**Figure 5.13:** Different cluster sizes, k=10

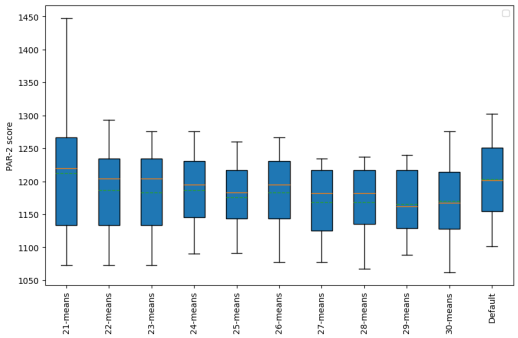



**Figure 5.14:** Different cluster sizes, k=5

We see that there is again no significant difference between different numbers of clusters. While k=10 works significantly better than k=5, even with k=10 the performance is not significantly better than simply using the LBS approach. Ultimately, both approaches perform significantly worse on clusters than they did on the default labeling. We thus conclude that the clustering approach does not work well as labeling, leaving us one more labeling approach to explore.

### 5.1.3 Clustering by best configurations

We again use k-means to cluster over the configurations and select the best performing cluster, then follow the approach laid out in 3.2.1. Before we dive in, it can be helpful to look at how the parameter k affects the bitvector.

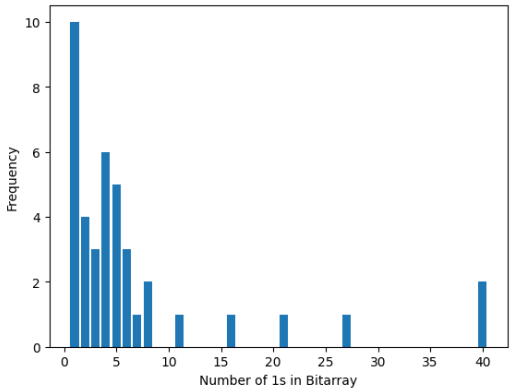



**Figure 5.15:** Exemplary distribution of top cluster sizes for k=10

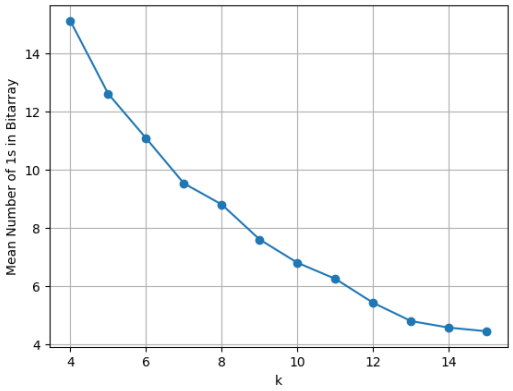



**Figure 5.16:** Examplary mean bitarray size across families, by k

We see that even for $k = 10$, there are still many families with 5 or more top configurations (Figure 5.15). Although we allow more top configurations than SNNAP [9]—which permits at most two or three—our goal remains to identify relatively small and meaningful top clusters. We thus focus on values of k between 8 and 13, as lower values lead to very large cluster sizes and higher values do not change the distribution significantly anymore (Figure 5.16).

#### Label Best Solver

There are two parameters to consider in this approach. Firstly, as mentioned before, we need to consider the strictness of our top cluster selection approach, given by the parameter $k_{config}$. Secondly, we want to fixate an appropriate number of classes $n_{labels}$. We evaluate all combinations for $k_{config} \in [8, 13]$ and $n_{labels} \in [10, 24]$.

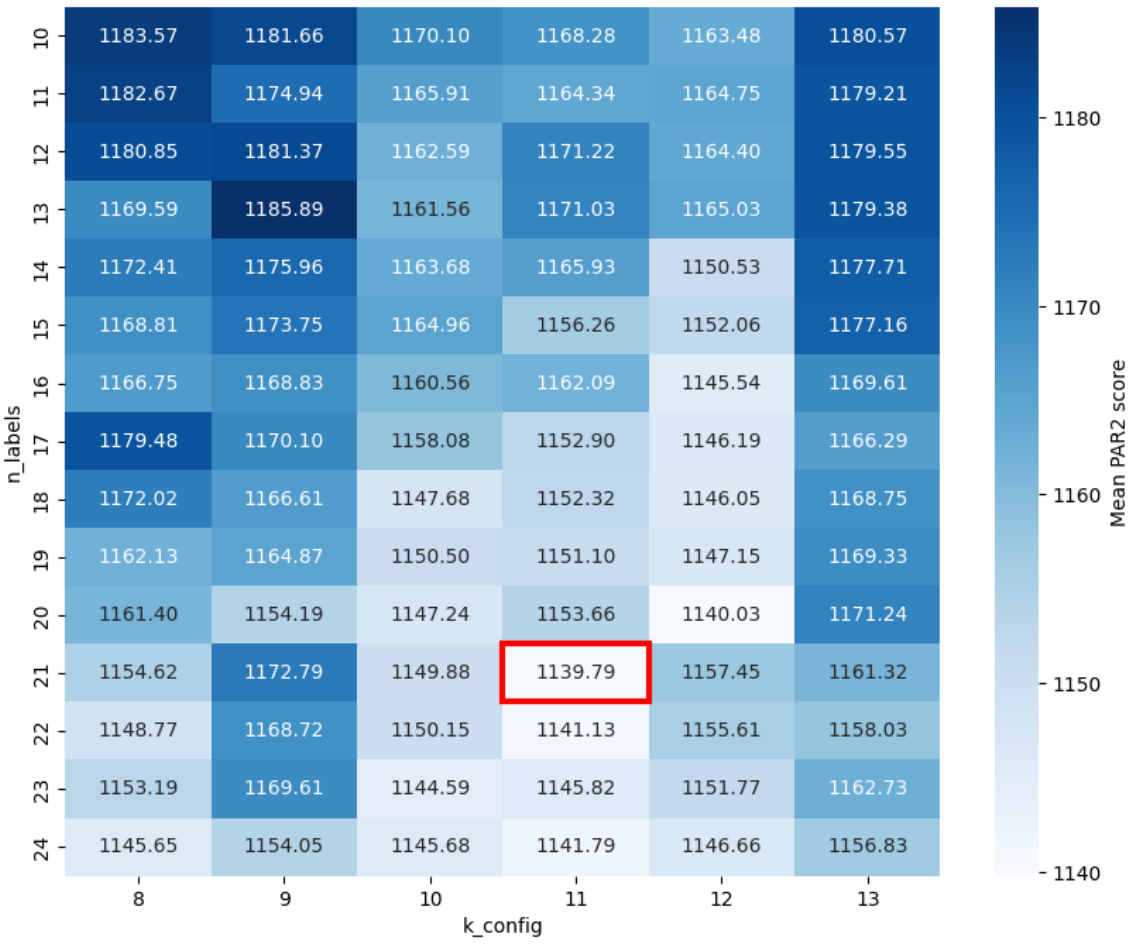



**Figure 5.17:** Heatmap of mean PAR2-score for different values of $k_{config}$ and $n_{labels}$

We can see that an increase in the number of labels is beneficial in most cases. Considering that there are only 40 distinct families and that our new labels are simply groupings of families, this does not technically speak in favor of this approach. Still, there seems to be at least a local minimum around $k_{config} = 11$ and $n_{labels} = 21$. Still, averaging over the same seeds, combining default labeling with the minimum hitting set approach works significantly better.

### Minimum hitting set

We again try the minimum hitting set approach, with the parameter k for runtime clustering set to 10, for different numbers of labels.

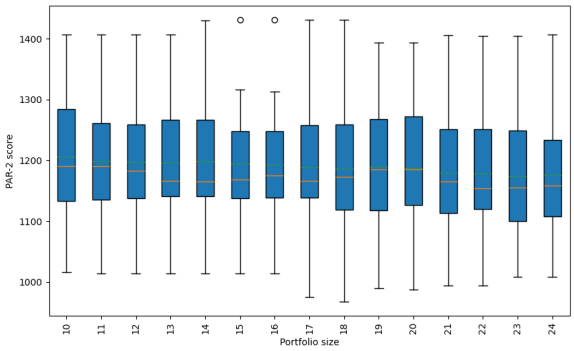



**Figure 5.18:** Comparing different numbers of labels for the minimum hitting set approach, for the value $k = 10$ when calculating top clusters

We again observe that more labels directly correlate with better performance. Seeing as even for 25 labels the performance is still considerably worse than for the 40 default labels, we again conclude that this labeling approach is not performant compared to the database labels.

This leads to the conclusion that among the labeling approaches we evaluated, having prior knowledge about the structure of an instance (i.e. the original problem the instance was derived from) is the most reliable way to label data.

## 5.2 Best configurations

After evaluating different ideas on how to create a configuration portfolio, we will now train one final solver on the whole training set, making it the product of this thesis. The most successful approach in testing has been default database labeling paired with the minimum hitting set approach with parameter k set to 9 5.1.1. We apply this approach to the whole dataset, yielding the following configurations, paired with the amount of instances assigned to them:

| Configuration | Number of instances |
|---|---|
| 0 | 292 |
| 21 | 118 |
| 6 | 79 |
| 7 | 59 |
| 9 | 48 |
| 29 | 45 |
| 27 | 41 |
| 25 | 40 |
| 16 | 37 |

**Table 5.1:** Configurations selected for the final classifier, paired with the number of instances from the training set assigned to them

We see that the default configuration 0 is by far the most commonly selected configuration. There are 759 instances in total, meaning that configuration 0 was selected for roughly 38,5% of training instances.

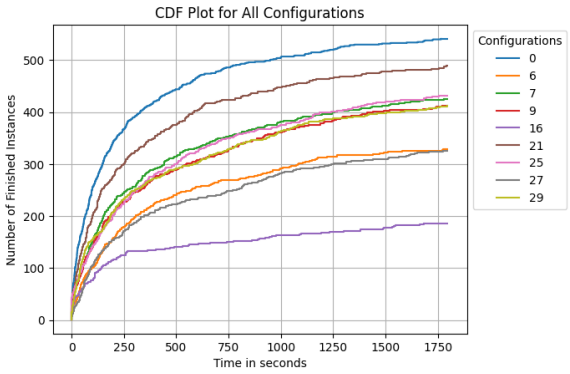



**Figure 5.19:** Accumulated solved instances over time, by configuration

Looking at a CDF plot of each individual configuration over the whole training set (Figure 5.19), we confirm that the default configuration is the best 'allrounder' among the

selected configurations, solving the most instances before the timeout. While some other configurations also perform relatively well, we see some that display awful performance when looking at the whole dataset. This makes sense, as we were aiming to create a complementary portfolio, meaning that some configurations may only work well on a subset of instances rather than on the whole dataset.

| Portfolio VBS | Default | Best 9-portfolio VBS |
|---|---|---|
| 973 | 1224 | 910 |

**Table 5.2:** VBS $PAR_2$ of the final portfolio, compared with the Default configuration and the best 9-portfolio extracted using beam-search

Comparing the VBS PAR2 score of the final portfolio to that Default configuration, we can see a major increase. However, checking for the best 9-portfolio on the whole dataset using beamsearch, we see that there is still a noticable discrepancy. This makes sense, as the metric for acquiring our portfolio was not VBS but LBS on each database label, meaning that each configuration has been averaged over a whole label. If there are e.g. two configurations which are complementary and both work exceptionally well for subsets of a label, only one of them may be selected for said label in our approach.

| Family | Count | 0 | 6 | 7 | 9 | 16 | 21 | 25 | 27 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|
| rest | 86 | **1509** | 2467 | 1835 | 1902 | 2961 | 1937 | 1730 | 1808 | 1716 |
| hardware-verification | 10 | 1214 | 3600 | 946 | **909** | 3241 | 2492 | 1198 | 1574 | 2966 |
| social-golfer | 20 | 2796 | 3477 | 3502 | 3474 | 3253 | **2617** | 3294 | 3499 | 2963 |
| cryptography-simon | 27 | 2266 | 2266 | **2266** | 3600 | 2270 | 2266 | 2266 | 3600 | 2576 |
| miter | 56 | **928** | 1871 | 1260 | 1436 | 2511 | 1665 | 1161 | 1836 | 2283 |
| cryptography | 15 | **734** | 1474 | 2279 | 2511 | 2024 | 850 | 2027 | 2297 | 1196 |
| planning | 8 | 1202 | 2536 | 1156 | 819 | 3150 | 2022 | **741** | 835 | 2256 |
| subgraph-isomorphism | 6 | 1389 | 1342 | 1354 | 1334 | 2400 | 1800 | 934 | **859** | 1424 |
| hamiltonian | 40 | 122 | 433 | 1206 | 809 | 3600 | **109** | 1086 | 1434 | 534 |
| reg-n | 9 | 2001 | 3600 | 2001 | 2001 | 3600 | **2001** | 3600 | 3600 | 3600 |
| software-verification | 15 | **176** | 728 | 390 | 1015 | 3600 | 556 | 195 | 348 | 958 |
| pigeon-hole | 7 | 2272 | 2601 | **2182** | 2353 | 2596 | 2227 | 2629 | 2634 | 2699 |
| minimum-disagreement-parity | 17 | 2625 | 2567 | 2767 | 2689 | 3221 | 2559 | **2352** | 2582 | 2894 |
| heule-nol | 11 | 347 | 3600 | 3600 | 3600 | 3093 | **249** | 3600 | 3600 | 2483 |
| argumentation | 41 | 1005 | **750** | 1179 | 1129 | 3394 | 1063 | 984 | 1447 | 1038 |
| grs-fp-comm | 19 | **1530** | 3083 | 2270 | 1936 | 3600 | 2795 | 2405 | 2291 | 2921 |
| cryptography-ascon | 26 | 217 | 191 | 233 | 209 | **166** | 220 | 223 | 3600 | 2485 |
| tseitin-formulas | 18 | 3216 | 3292 | 3265 | **3077** | 3400 | 3233 | 3257 | 3415 | 3255 |
| scheduling | 51 | **1555** | 2493 | 2056 | 1954 | 2996 | 1611 | 1994 | 2229 | 1604 |
| hashtable-safety | 20 | **111** | 946 | 116 | 473 | 3600 | 732 | 396 | 281 | 577 |
| profitable-robust-production | 20 | 970 | 2179 | **76** | 1878 | 1884 | 1864 | 1819 | 1674 | 1917 |
| school-timetabling | 19 | 643 | 3410 | 3600 | 3600 | 3600 | 805 | 1961 | 3600 | **537** |
| bitvector | 5 | 2281 | 3020 | **1659** | 1779 | 2903 | 2995 | 1826 | 1720 | 2950 |
| maxsat-optimum | 13 | 508 | **366** | 831 | 638 | 2983 | 568 | 586 | 722 | 677 |
| or$_r$$andxor$ | 6 | **8** | 3600 | 10 | 9 | 3600 | 10 | 3600 | 3600 | 3600 |
| subsumptiontest | 5 | 24 | 28 | 25 | 22 | 819 | **18** | 26 | 59 | 28 |
| satcoin | 15 | 905 | **815** | 2271 | 1034 | 3600 | 907 | 1071 | 3600 | 3600 |
| mutilated-chessboard | 15 | 1824 | 2750 | 1062 | 1152 | 3600 | 2473 | **862** | 999 | 1849 |
| relativized-pigeon-hole | 6 | **3600** | **3600** | **3600** | **3600** | **3600** | **3600** | **3600** | **3600** | **3600** |
| register-allocation | 20 | 36 | 3200 | 118 | **33** | 3466 | 416 | 3207 | 3207 | 3600 |
| heule-folkman | 11 | 232 | 3600 | 3600 | 3600 | **90** | 204 | 3600 | 3600 | 137 |
| rbsat | 6 | 1843 | 1316 | 1904 | 1970 | 2169 | 2062 | 1487 | 1433 | **866** |
| random-circuits | 15 | 417 | 2513 | 414 | 554 | 3363 | **395** | 560 | 568 | 1110 |
| interval-matching | 20 | **3600** | **3600** | **3600** | **3600** | **3600** | **3600** | **3600** | **3600** | **3600** |
| brent-equations | 19 | **140** | 3600 | 3130 | 3101 | 947 | 571 | 642 | 3290 | 463 |
| coloring | 12 | 2605 | 2562 | 2654 | 2846 | 2730 | **2301** | 3134 | 3196 | 2493 |
| set-covering | 20 | 314 | 3600 | 3600 | 3600 | 266 | 225 | 3600 | 3600 | **84** |
| cril-misc | 5 | **473** | 530 | 965 | 1191 | 2881 | 537 | 943 | 897 | 1242 |
| independent-set | 15 | 2322 | 3600 | 2251 | 2247 | 3600 | 2881 | 2261 | **2238** | 2577 |
| quantum-kochen-specker | 10 | 149 | **106** | 226 | 176 | 2482 | 140 | 160 | 170 | 134 |

**Table 5.3:** $PAR_2$-score of entire families, over individual configurations. All configurations are highlighted in rows where every configuration timed out on every instance.

Evaluating each configuration on each family over the whole training set, we can see that complementarity is mostly given. Configuration 16 stands out in particular, yielding disastrous performances on most families, but outperforming the other families by a long shot on a few select families (cryptography-ascon, heule-folkman). Configuration 0 (Default) displays an almost consistently strong performance, often ranking first across all configu-

rations. This makes sense as it is the most commonly selected configuration in our training set. Looking at the families, we also see a few interesting results. Firstly, all configurations timed out on all instances of the families interval-matching and relativized pigeon hole. Another interesting family is or_randxor, where a few configurations do really well while the rest times out on every single instance.
Another interesting family is crypthography-simon, where multiple configurations have almost the exact same PAR2-score (all rounded to 2266 in this table). As this is a very noticeable result, we checked all configurations (including ones not listed in the final portfolio). As it turns out, configurations with the "lucky" option turned on solve a subset of instances in a fraction of a second while timing out on the rest. Configurations without the lucky option perform significantly worse, often timing out on all instances.

| Parameter | Parameter Space | 0 | 6 | 7 | 9 | 16 | 21 | 25 | 27 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|
| backbone | [0,2] | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 |
| bump | {0,1} | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| chrono | {0,1} | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| congruence | {0,1} | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| eliminate | {0,1} | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| extract | {0,1} | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| factor | {0,1} | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| fastel | {0,1} | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| forward | {0,1} | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| lucky | {0,1} | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| phase | {0,1} | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| phasesaving | {0,1} | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| preprocess | {0,1} | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| probe | {0,1} | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| randec | {0,1} | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| reluctant | {0,1} | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| reorder | [0,2] | 1 | 2 | 2 | 0 | 2 | 2 | 2 | 0 | 0 |
| rephase | {0,1} | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| restart | {0,1} | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| stable | [0,2] | 1 | 2 | 0 | 0 | 2 | 2 | 0 | 0 | 1 |
| substitute | {0,1} | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| sweep | {0,1} | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| target | [0,2] | 1 | 1 | 0 | 0 | 1 | 2 | 0 | 0 | 2 |
| transitive | {0,1} | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| vivify | {0,1} | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| warmup | {0,1} | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

**Table 5.4:** Individual parameter instantiation of each configuration by parameter

Looking at the configurations in detail, we see that for all parameters every possible value A.2 was taken at least once. Configuration 16 stands out again as being the only configuration to deactivate the 'bump' and 'vivify' options, which may be a possible explanation for its before mentioned solving behavior. The options 'reorder' and 'backbone', which both have three modes (0: turned off, 1: turned on, 2: eager) are both either set to eager mode

or turned off for all configurations except Default.

## 5.3 Testing our approach on the holdout set

Finally, we want to take a look at the classifier performance on the holdout set. We use the classifier trained in the last section 5.2 and run it on the holdout set. Additionally, we run all configurations in the portfolio on the whole holdoutset.
Firstly, we notice that the holdout set seems to be much more difficult than the training set. From our portfolio, the default configuration is the SBS, with a PAR2 score of **1608**. This is a comparatively high score, considering that the SBS PAR2 score on the training set (also the Default configuration) is 1224.
The classifier does work slightly better than the SBS with a PAR2 score of **1604**. Wondering about this minimal improvement, we looked at the configurations selected.

| Configuration | Number of instances |
|---|---|
| 0 | 2543 |
| 9 | 288 |
| 27 | 210 |
| 21 | 206 |
| 25 | 91 |
| 29 | 67 |
| 6 | 28 |
| 7 | 17 |
| 16 | 0 |

**Table 5.5:** Configurations of the portfolio and how many instances were assigned to them

We notice that the default configuration is assigned much more frequently in the holdout set than it was during training, with 73.7% of instances being assigned to it. This could be due to the training set not being fully representative. Upon examining the family labels of the holdout set, we find that several families are represented by more than 20 instances, whereas in the training set, these families are represented by very few instances—sometimes even just one. The result combined with the insights from 5.1.1 seem to indicate that the training data has insufficient instances on some labels, meaning that the classifier does not scale well for instance sets that contain unknown or lesser known problem classes.

| Family | Training Instances | Holdout Instances | Classifier | Default |
|---|---|---|---|---|
| miter | 56 | 171 | 912 | 912 |
| scheduling | 51 | 137 | 1412 | 1412 |
| tseitin-formulas | 18 | 114 | **1910** | 2112 |
| cryptography | 15 | 264 | 1733 | **1715** |
| software-verification | 15 | 63 | 492 | 492 |
| coloring | 12 | 173 | **2333** | 2499 |
| hardware-verification | 10 | 158 | **685** | 772 |
| planning | 8 | 126 | 1322 | **1074** |
| pigeon-hole | 7 | 88 | 2892 | **2865** |
| relativized-pigeon-hole | 6 | 32 | 2604 | 2604 |
| subgraph-isomorphism | 6 | 146 | **2075** | 2128 |
| rbsat | 6 | 95 | **2383** | 2609 |
| bitvector | 5 | 119 | **1481** | 1486 |
| cril-misc | 5 | 53 | 933 | 933 |
| rooks | 4 | 24 | 1903 | 1903 |
| polynomial-multiplication | 4 | 40 | 1643 | 1643 |
| cellular-automata | 4 | 30 | 575 | 575 |
| sgen | 4 | 52 | 2889 | 2889 |
| graph-isomorphism | 2 | 60 | 747 | 747 |
| hgen | 2 | 42 | 2201 | 2201 |
| stedman-triples | 2 | 37 | 1815 | 1815 |
| quasigroup-completion | 2 | 55 | 665 | 665 |
| tree-decomposition | 1 | 22 | 1995 | 1995 |
| unknown | 1 | 59 | 1990 | **1954** |
| waerden | 1 | 35 | 2509 | 2509 |
| random-modularity | 1 | 38 | 562 | 562 |
| prime-factoring | 1 | 37 | 1547 | 1547 |
| popularity-similarity | 1 | 39 | 3160 | 3160 |
| hypertree-decomposition | 1 | 26 | 1027 | 1027 |
| edge-matching | 1 | 30 | 1832 | 1832 |
| clique-width | 1 | 22 | 3083 | 3083 |
| bioinformatics | 0 | 28 | 1234 | 1234 |
| petrinet-concurrency | 0 | 25 | 1770 | 1770 |
| generic-csp | 0 | 31 | 2064 | **1687** |
| fixed-shape-random | 0 | 27 | 247 | 247 |
| termination-analysis | 0 | 21 | 908 | 908 |
| diagnosis | 0 | 69 | 1014 | 1014 |
| auto-correlation | 0 | 47 | 2023 | 2023 |

**Table 5.6:** Families with at least 20 instances in the holdout set, with classifier and default PAR2 performance. Sorted by training instance count in a descending manner.

Looking at commonly occuring families from the holdout set, we see that the classifier tends to score well if there were enough corresponding training instances, achieving significantly better PAR2 scores than the default configuration on some families. On families that had less than 5 training instances (remember, 5 was the threshold under which the family would be assigned to the rest labels), the classifier almost always selects the default configuration. In the cases where it does not, the classifier scores significantly worse than the default configuration. Still, these families make up a significant part of the holdout set. We can thus conclude that the training set sadly did not contain data comparable to that of the holdout set, which explains why our approach did not scale well.

# 6 Discussion

## 6.1 Conclusion

With the insights gained from our research, we can answer the questions posed in the introduction.

**How may instances be grouped effectively? Does the original problem that a SAT instance is derived from play a major role for an effective grouping?**

Using predefined labels has proven to yield considerably better results than methods like clustering. These labels correspond to the original problems from which the instances were derived, allowing us to confidently answer the second question with a clear "yes." However, it is important to note that such labels may not always be available or desired in every dataset.

**How do the selected configurations complement each other?**

It is clear that configurations must be complementary, with each parameter in the configuration space receiving every possible value at least once in the final portfolio. Additionally, options with an eager mode tend to be either completely turned off or set to eager mode in most configurations. Some configurations, despite their relatively poor overall performance, stand out due to their exceptional performance on a select number of labels. Nevertheless, there appears to be a need for an "allrounder" configuration that performs well across a wide range, as highlighted by the frequent use of the default configuration.

**How does the number of configurations affect classification performance? Is there an especially advantageous number of configurations to be used?**

We found that, with a low number of configurations, increasing the number of configurations led to a notable improvement in VBS performance. However, this improvement begins to level off in an approximately exponential manner, after which misclassifications become more significant than the marginal gains in VBS performance. In our dataset, nine configurations appeared to be the threshold for optimal performance across different approaches.

## 6.2 Future Work

There are a few important questions that are left open by this thesis, which might be explored in further research:

### More representative data

The training set we worked with was comparatively small and sadly not representative of the holdout set. Training our winning approach on a larger amount of more diverse data may yield a classifier that is better equipped to deal with new data.

### Different initial instance grouping

In this thesis, the initial instance grouping used for hyperparameter optimization was based on the database labeling. Within this context, the poor performance of our clustering-based approach may be partially attributed to the fact that a configuration was considered "good" based on its performance within a specific label group. Additionally, some of these initial groups were quite inhomogeneous, encompassing a wide variety of labels. Consequently, it may be worthwhile to explore alternative strategies for initial instance grouping.

### Optimizing Decision Tree Hyperparameters

While we optimized the hyperparameters directly involved in our approaches, we did not optimize the hyperparameters of our classifier models. There are a number of parameters (e.g. minimum leaf size) that may be tweaked to increase performance.

# A Implementation Details

## A.1 Feature Set

This is a list of the features used to classify the instances. We excluded the base features "ccs" and "bytes" as they are not complete across all instances used. We further excluded the feature "runtime", as acquiring information about runtime performance for an instance requires solving the instance first, making runtime useless as a feature to use on new instances. The train/test set can be found using the query "(track=main_2024 or track=main_2023) and minisat1m!=yes", the holdout set using the query "track=anni_2022 and (track!=main_2024 and track!=main_2023) and minisat1m!=yes" in the base feature section of GBD: https://benchmark-database.de/
The full list of base features used is thus:
clauses, variables, cls1, cls2, cls3, cls4, cls5, cls6, cls7, cls8, cls9, cls10p, horn, invhorn, positive, negative, hornvars_mean, hornvars_variance, hornvars_min, hornvars_max, hornvars_entropy, invhornvars_mean, invhornvars_variance, invhornvars_min, invhornvars_max, invhornvars_entropy, balancecls_mean, balancecls_variance, balancecls_min, balancecls_max, balancecls_entropy, balancevars_mean, balancevars_variance, balancevars_min, balancevars_max, balancevars_entropy, vcg_vdegree_mean, vcg_vdegree_variance, vcg_vdegree_min, vcg_vdegree_max, vcg_vdegree_entropy, vcg_cdegree_mean, vcg_cdegree_variance, vcg_cdegree_min, vcg_cdegree_max, vcg_cdegree_entropy, vg_degree_mean, vg_degree_variance, vg_degree_min, vg_degree_max, vg_degree_entropy, cg_degree_mean, cg_degree_variance, cg_degree_min, cg_degree_max, cg_degree_entropy

## A.2 Configuration parameters

The following table provides a list of the configuration parameters used in this thesis, as well as the configuration space and a short description of each parameter as given by the –help parameter of Kissat version SC-2024. We do not include the default parameter values, as they are 1 for every parameter in this list

| Parameter | Configuration Space | Short decription |
|---|---|---|
| backbone | [0,2] | binary clause backbone (2=eager) |
| bump | {0,1} | enable variable bumping |
| chrono | {0,1} | allow chronological backtracking |
| congruence | {0,1} | congruence closure on extracted gates |
| eliminate | {0,1} | bounded variable elimination BVE |
| extract | {0,1} | extract gates in variable elimination |
| factor | {0,1} | bounded variable addition |
| forward | {0,1} | forward subsumption in BVE |
| lucky | {0,1} | try some lucky assignments |
| phase | {0,1} | initial decision phase |
| phasesaving | {0,1} | enable phase saving |
| preprocess | {0,1} | initial preprocessing |
| probe | {0,1} | enable probing |
| randec | {0,1} | random decisions |
| reluctant | {0,1} | stable reluctant doubling restarting |
| reorder | [0,2] | reorder decisions (1=stable-mode-only) |
| rephase | {0,1} | reinitialization of decision phases |
| restart | {0,1} | enable restarts |
| stable | [0,2] | enable stable search mode |
| substitute | {0,1} | equivalent literal substitution |
| sweep | {0,1} | enable SAT sweeping |
| target | [0,2] | target phases (1=stable,2=focused) |
| transitive | {0,1} | transitive reduction of binary clauses |
| vivify | {0,1} | vivify clauses |
| warmup | {0,1} | initialize phases by unit propagation |

**Table A.1:** Configuration parameters used in this thesis, with short descriptions

## A.3 Family groupings

The following table gives an overview over the initial groupings, which were selected to acquire the configurations used in this portfolio using SMAC.

| Families |
|---|
| cryptography, cryptography-ascon, cryptography-simon |
| miter |
| scheduling |
| tseitin-formulas, random-circuits, modcircuits,<br>circuit-multiplier, or-randxor |
| hamiltonian, hamiltonian-cycle |
| argumentation<br>mutilated-chessboard, pigeon-hole, |
| relativized-pigeon-hole, binary-pigeon-hole |
| summle,influence-maximization,st-connectivity-principle,random-csp,stedman-triples,<br>fpga-routing,unknown,hgen,crafted-cec,test-configuration<br>minimum-disagreement-parity,binary-tree-parity,parity-games,minimal-disagreement-parity<br>maxsat-optimum,rbsat |
| social-golfer |
| set-covering |
| register-allocation |
| profitable-robust-production |
| product-configuration,automata-synchronization,ramsey,<br>ensemble-computation,edge-matching,clique-formulas,long-learned-clauses,<br>hypertree-decomposition,erdos-discrepancy,clique-width,<br>fermat,waerden,subset-cardinality,popularity-similarity,hardware-model-checking,<br>theorem-proving,generic-csp,random-modularity,core-based-generator |
| interval-matching |
| hashtable-safety |
| subsumptiontest,equivalence-chain-principle,risc-instruction-removal-subrv,<br>risc-instruction-removal-golcrest,sgen |
| school-timetabling |
| quasigroup-completion,bitvector,cril-misc |
| grs-fp-comm |
| coloring,clique-colouring,pythagorean-triples |
| brent-equations |
| software-verification |
| satcoin |
| independent-set |
| reg-n,cellular-automata |
| hardware-verification,hardware-bmc |
| trigonometric-functions,polynomial-multiplication,prime-factoring,matrix-multiplication |
| planning |
| heule-nol |
| heule-folkman |
| subgraph-isomorphism,graph-isomorphism,tree-decomposition<br>rooks,pebbling,puzzle,battleship,knights-problem,baseball-lineup |
| quantum-kochen-specker |

**Table A.2:** Family groupings

## A.4 Libraries

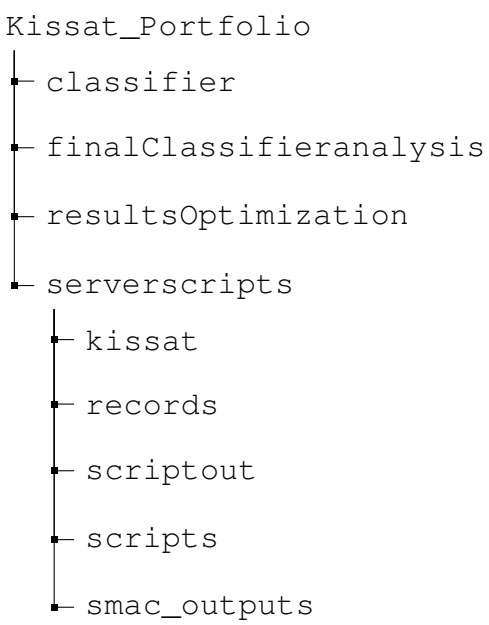
The following list provides an overview over the relevant python libraries used in this thesis.

- smac3 to optimize kissat hyperparameters
- scikit-learn for classification and evaluation of models
- gbd_core API for database management
- itertools for implementing beam search/minimum hitting set
- pickle to store results and models
- matplotlib to visualize results
- pandas to manage data
- pebble for parallelism

## A.5 Structure implementation

The github repository can be found here: https://github.com/Error-coding/Kissat_Portfolio
The structure is as follows:

- **serverscripts** contains all scripts used on the computing cluster. Besides the scripts themselves (scripts), their logs (scriptout) and the output of smac during hyperparameter optimization (smac_outputs), the folder also contains the runtime results of all instances evaluated (records).
- **resultsOptimization** contains an evaluation of the data acquired during hyperparameter optimization. In this folder, the configurations to be used in our final portfolio are selected.
- **classifier** contains a number of jupyter notebooks, evaluating different labeling and configuration selection approaches. Subfolders contain results of evaluations as well as trained clasifiers.
- Finally, **finalClassifieranalysis** contains a jupyter notebook dealing with the analysis of the final classifier on the holdout set


Kissat_Portfolio
classifier
finalClassifieranalysis
resultsOptimization
serverscripts
kissat
records
scriptout
scripts
smac_outputs

## A.6 All configurations

The following tables provide an overview over all configurations acquired in the SMAC optimization stage.

| Parameter | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| backbone | 1 | 0 | 1 | 2 | 0 | 0 | 2 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 1 | 0 |
| bump | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| chrono | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| congruence | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| eliminate | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| extract | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| factor | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| fastel | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| forward | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| lucky | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| phase | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| phasesaving | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| preprocess | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| probe | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| randec | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| reluctant | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| reorder | 1 | 1 | 2 | 0 | 2 | 1 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 2 | 0 | 1 | 0 |
| rephase | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| restart | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| stable | 1 | 0 | 1 | 2 | 0 | 2 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 2 | 2 | 1 | 2 | 0 | 1 | 0 |
| substitute | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| sweep | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| target | 1 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 2 | 2 | 1 | 0 | 0 | 2 | 2 | 1 | 0 | 1 | 0 |
| transitive | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| vivify | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| warmup | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |

**Table A.3:** Individual parameter instantiation of each configuration by parameter, configurations 0-19

| Parameter | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| backbone | 2 | 2 | 2 | 0 | 2 | 0 | 2 | 1 | 0 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| bump | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| chrono | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| congruence | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| eliminate | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| extract | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| factor | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| fastel | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| forward | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| lucky | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| phase | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| phasesaving | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| preprocess | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| probe | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| randec | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| reluctant | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| reorder | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 1 | 0 | 2 | 0 |
| rephase | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| restart | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| stable | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 0 | 1 | 2 | 0 | 2 | 2 | 1 | 2 | 0 | 2 | 1 | 0 | 0 |
| substitute | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| sweep | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| target | 0 | 2 | 1 | 0 | 2 | 0 | 2 | 1 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 2 | 0 | 0 |
| transitive | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| vivify | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| warmup | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |

**Table A.4:** Individual parameter instantiation of each configuration by parameter, configurations 20-39

# Bibliography


[1] Tasniem Nasser Alyahya, Mohamed El Bachir Menai, and Hassan Mathkour. On the structure of the boolean satisfiability problem: A survey. *ACM Comput. Surv.*, 55(3), March 2022.

[2] Jakob Bach, Markus Iser, and Klemens Böhm. A comprehensive study of k-portfolios of recent sat solvers. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022). Hrsg.: Kuldeep S. Meel*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, page 2:1–2:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (LZI), 2022.

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

[4] Tomas Balyo, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2022.

[5] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024.

[6] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.

[7] Armin Biere, Katalin Fazekas, Mathias Fleury, and Nils Froleyks. Clausal Congruence Closure. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:25, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[8] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL,

Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[9] Marco Collautti, Yuri Malitsky, Deepak Mehta, and Barry Sullivan. Snnap: Solver-based nearest neighbor for algorithm portfolios. volume 8190, 09 2013.

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

[11] Matti Jarvisalo Heule, Marijn and Tomas Balyo. Proceedings of SAT Competition 2017.

[12] Markus Iser and Christoph Jabs. Global Benchmark Database. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:10, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[13] Xin Jin and Jiawei Han. *K-Means Clustering*, pages 563–564. Springer US, Boston, MA, 2010.

[14] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac - instance-specific algorithm configuration. volume 215, pages 751–756, 01 2010.

[15] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[16] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization, 2022.

[17] John R. Rice. The algorithm selection problem. volume 15 of *Advances in Computers*, pages 65–118. Elsevier, 1976.

[18] João Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. *Frontiers in Artificial Intelligence and Applications*, 185, 01 2009.

[19] 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, July 2008.