Bachelor thesis

Cluster Analysis for SAT Instances

Paul Ferdinand Heinen

Date: February 26, 2022

Supervisors: Dr. rer. nat. Markus Iser
M.Sc. Jakob Bach
Reviewer: Prof. Dr. Peter Sanders

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

Abstract

The Boolean Satisfiability Problem (SAT) has been the focus of extensive research when it comes to developing solvers that excel on solving its problem instances. Research shows that selecting solvers based on features of the problem instance can improve the speed of solving these instances. However, as of now, it has been unclear when specific features lead to the selection of specific solvers. In this thesis, we use different clustering algorithms on a given dataset of SAT-Instances and analyze how different clustering algorithms perform, as well as look at selected single clusterings to find connections between the instances and their best solvers.

We will show that *K-Means* and *DBSCAN* excel in comparison to other clustering algorithms when it comes to grouping instances with similar solvers together. Next, we will take a look how selected clusterings of *K-Means* and *DBSCAN* are structured. We will observe, that many clusters only have one or two good solvers that are the fastest and most stable solvers. Furthermore, we will see that many clusters are a subset of SAT-Instance families.

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.

Karlsruhe, den 26.02.2022

Contents

Abstract **3**

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Contribution . . . 10
1.3 Structure of thesis . . . 10

2 Preliminaries **11**
2.1 Fundamentals . . . 11
2.1.1 Boolean satisfiability problem (SAT) . . . 11
2.1.2 SAT-Solvers . . . 11
2.1.3 The algorithm selection problem . . . 13
2.1.4 Clustering . . . 13
2.2 Related work . . . 16
2.2.1 SATzilla . . . 16
2.2.2 ISAC . . . 16
2.2.3 SNNAP . . . 17

3 Experimental design **19**
3.1 Experimental pipeline . . . 19
3.1.1 Feature extraction . . . 20
3.1.2 Preprocessing of features . . . 20
3.1.3 Clustering . . . 21
3.1.4 Evaluation measures . . . 21
3.2 Experimental setup . . . 27
3.2.1 Environment . . . 27
3.2.2 Libraries . . . 28
3.2.3 Filter parameters . . . 28
3.2.4 Dataset characteristics . . . 29

4 Experimental evaluation **33**
4.1 Preprocessing of the data . . . 33
4.1.1 Scaling . . . 33
4.1.2 Clusterings with single features . . . 36

4.2 Comparing feature sets and clustering algorithms . . . 38
4.2.1 Evaluating clustering algorithms . . . 39
4.2.2 Analysis of selected clustering algorithms . . . 41
4.2.3 Evaluation of selected clusterings . . . 46

5 Discussion **61**
5.1 Conclusion . . . 61
5.2 Future Work . . . 62

A Implementation Details **63**
A.1 Project organization . . . 63
A.2 Default settings tables . . . 64
A.2.1 Scaling and feature selection . . . 64
A.2.2 Clustering . . . 64
A.3 Feature sets . . . 66
A.3.1 '*base*' . . . 66
A.3.2 '*gate*' . . . 67
A.3.3 '*runtimes*' . . . 67
A.4 Family distribution of selected clusters . . . 67
A.4.1 K-Means . . . 68
A.4.2 DBSCAN . . . 70

Bibliography **73**

1 Introduction

1.1 Motivation

The Boolean Satisfiability Problem (SAT) is one of the fundamental problems when it comes to the research of problems in the $\mathcal{NP}$ domain. The SAT-Problem has been the focus of extensive research, because it occurs in many fields such as cryptography, scheduling or AI. Furthermore, other $\mathcal{NP}$-Problems can be easily encoded as SAT.
To be able to solve the SAT-Problem, many SAT-Solvers were developed, such as conflict-driven clause learning solvers, lookahead solvers and local search solvers. However, while all solvers have instances of the SAT-Problem they excel at, each of them has instances with very poor runtime behavior. We face the algorithm selection problem (Rice 1976), where we want to choose the best solver for a new, previously unknown, instance of the SAT-Problem. Because of this, the focus of solving SAT-Instances has shifted to construct algorithm selectors and algorithm portfolios such as *SATzilla* (Xu et al. 2008), *ISAC* (Kadioglu et al. 2010) and *SNNAP* (Collautti et al. 2013). The goal of an algorithm selector is to use the features of the given SAT-Instance to identify a solver with fast runtime for it. Both *ISAC* and *SNAAP* use clustering to group the instances based on their features and identify fast solvers for each group. However, there has been a disconnect between the clustering objective and the performance objective of the solvers (Collautti et al. 2013). This means we cannot assume that a clustering algorithm sorts an instance, based on its features, into a cluster with the optimal solver.
Central questions are:

- Q1: How do different sets of instance features influence the clustering quality of different clustering algorithms?
- Q2: What clustering algorithms and parameter settings yield clusterings with high quality?
- Q3: How are the instances in a clustering split into clusters?
- Q4: Do instances in a cluster share solvers, which shows homogeneous runtime behavior?

1.2 Contribution

In this thesis, we want to examine the connection between SAT-Instance features and the performance of different SAT-Solvers on them using a cluster analysis. We will look how preprocessing and different sets of features influence the clusterings, as well as discuss how different clustering algorithms perform. We will select clustering algorithms that perform well and examine how their parameter settings influence the resulting clusterings.
After that, we will analyze selected clusterings of these clustering algorithms and look how these clusterings are structured.

1.3 Structure of thesis

In chapter 2, we will introduce the SAT-Problem, the concept of SAT-Solvers, the algorithm selection problem as well as different clustering algorithms. After that, we will present related work using these concepts in the form of the approaches *SATzilla*, *ISAC* and *SNNAP*. Chapter 3 explains the experimental pipeline and setup we used to generate and evaluate the clusterings. Chapter 4 is split into two parts:
In section 4.1 we will present the preprocessing experiments, where we decide what settings we use for preprocessing. After that, section 4.2 evaluates the clusterings we created using different filtering parameters. Using these filter parameters, we will select good clustering algorithms and clusterings we will examine in more detail.
Lastly, we will discuss our findings in chapter 5 as well as give an outlook on possible future work.
Appendix A shows an overview of the project organization, default settings for the clustering, all features of the instance database as well as more figures of the selected clusterings.

2 Preliminaries

This chapter introduces preliminaries for the later experiments. The chapter is split into two parts. In section 2.1 we introduce the concepts of SAT-Problem, SAT-Solvers, the algorithm selection problem and clustering. In section 2.2 we discuss related work using these concepts.

2.1 Fundamentals

2.1.1 Boolean satisfiability problem (SAT)

One defines the Boolean satisfiability problem as follows:
Each problem instance consists of a set of boolean variables $X = \{x_1, ..., x_m\}$, where each variable x_i can be assigned the value of 0 or 1.
For each variable x_i we can define the literals x_i and $\overline{x}_i$ so that $\forall i \in \{1, ..., m\} : (x_i = 1 \Leftrightarrow \overline{x}_i = 0) \land (x_i = 0 \Leftrightarrow \overline{x}_i = 1)$. Using the literals, we can construct a clause of the form $(y_1 \lor ... \lor y_s)$ where $\forall j \in \{1, .., s\} : y_j \in \{x_1, .., x_m\} \cup \{\overline{x}_1, .., \overline{x}_m\}$
The Boolean satisfiability problem (SAT) asks if, for a given set of boolean variables X and a set of clauses C over X, there exists an assignment of the variables X so that every clause in C is satisfied. We call a clause satisfied if each clause in C contains a literal with value 1. (D. Wagner, Sauer, and Brückner 2019).

2.1.2 SAT-Solvers

A SAT-Solver takes a set of clauses C (often the clauses are given as a conjunctive normal form) as input and determines if there is an assignment of truth values $\{0, 1\}$ that satisfy the set of clauses C. We call a solver an incomplete SAT-Solver if it can find satisfying assignments, but not prove unsatisfiability (Lindauer et al. 2017).

When running a SAT-Solver on a given problem instance, a variable can be either *assigned* with a boolean value of 0 or 1, or it may be *unassigned*.
A clause of a SAT-Problem instance can be *unsatisfied*, *satisfied*, *unit*, and *unresolved*. If the algorithm assigns the value 0 to all literals of the clause, a clause is *unsatisfied*. If the algorithm assigns the value 1 to at least one literal, the clause is *satisfied*. If the algorithm

assigns the value 0 to all but one literal of the clause and leaves the remaining literal unassigned, the clause is *unit*. A clause is *unresolved* if it is neither *satisfied*, *unsatisfied* nor *unit*.
A key procedure for SAT-Solvers is the *unit clause rule*, which states that if a clause is *unit*, then the last unassigned literal must be assigned a value of 1 for the clause to be *satisfied*. Using the unit clause rule iteratively is called *unit propagation* or *Boolean constraint propagation* (BCP) (Silva, Lynce, and Malik 2009).

There have been different approaches for constructing SAT-Solvers. The three most common solver types are conflict-driven clause learning (Silva, Lynce, and Malik 2009), local search (Hoos and Stützle 2000) and look-ahead (Heule and Maaren 2009) solvers.

Conflict-Driven clause learning (CDCL) Solvers CDCL solvers use unit propagation to derive logical consequences. To do this, the solver applies unit propagation after each branching step to identify variables that need to have specific boolean values (Silva, Lynce, and Malik 2009).

The dataset used in this thesis contains the following CDCL Solvers:

- **CaDiCal**: Has the two configurations *cadical_elimfalse* (variable elimination deactivated) and *cadical* (default configuration) as well as the two hacks *cadical_pripro* (prioritized propagation) and *cadical_sability* (increases probability of solving crypto instances)
- **Glucose**: Has the five configurations *glucose* (default configuration), *candy* (simplified, modular version of glucose), *glucose_chanseok* (has strengths on existing instances), *glucose_syrup* (runs on four parallel threads), *glucose_var_decay099* (glucose configuration)
- **Kissat**: Best CDCL SAT-Solver in the main track of the SAT-Competition 20/21 (Heule, Jarvisalo, Suda, et al. 2021)
- **Lingeling**
- **Minisat**
- **Relaxed Maple**

Local Search Algorithms Local Search examines a search space given by the problem instance to solve, by initializing the problem at some point and then iteratively moving from one search space position to a neighboring search space position. The decision of what neighboring position to choose is only based on the information about the local neighborhood. To score these neighbors, an objective function $f : S_\pi \to \mathbb{R}$ maps each search space position to a value. In SAT, f often describes the number of unsatisfied clauses, which we try to minimize. The 1-flip neighborhood typically determines the neighborhood,

which describes the neighbors of a search space position, by considering two search space positions neighbors if they differ in the truth value of exactly one variable. Local Search Algorithms are typical incomplete, because they can not guarantee to find a solution for unsatisfiable problem instances. (Hoos and Stützle 2000).

The dataset used in this thesis contains the Local Search Solver **YalSat**.

Look-ahead Solver Look-ahead Solvers have a similar approach to CDCL Solvers, however they spend much more time on deciding which variable assignment they check next. This is done by using the look-ahead, which checks the next neighboring state, evaluating its effect and then backtracking and ending the look-ahead (Heule and Maaren 2009).

The dataset used in this thesis contains the Look-ahead Solver **March**.

2.1.3 The algorithm selection problem

The current research suggests, that different instances of SAT are best solved by different SAT-Solvers (Xu et al. 2008, Collautti et al. 2013, Kadioglu et al. 2010). Therefore, no single SAT-Solver describes the state-of-the-art when it comes to efficiently solve any SAT-Instance. Instead, the focus has shifted to algorithm selection. In the case of SAT the goal is to select the best SAT-Solver for a given instance. Specifically, given a set of instances $I = \{p_i, ..., p_n\}$ of the SAT-Problem, a set of SAT-Solvers $A = \{s_1, ..., s_m\}$ and a metric $m : A \times I \rightarrow \mathbb{R}$ that measures the performance of any solver $s_j \in A$ on the instances I, construct a selector S that maps any problem of instance $p_i \in I$ to a SAT-Solver $S(p_i) \in A$ such that the overall performance of S on I is optimal according to the metric m (Rice 1976) (Kerschke et al. 2019).
We call a perfect algorithm selector that finds the best SAT-Solver for any given problem instances a *Virtual Best Solver* (*VBS*). The *VBS* gives us a lower bound when constructing an algorithm selector with the current state-of-the-art SAT-Solvers. However, we cannot hope to achieve the construction of the *VBS*, instead the goal is to extract a set of easily computable features $F = \{f_1(p_i), ..., f_k(p_i)\}$ from a given instance p_i and use them to determine a solver with high performance for the instance (Kerschke et al. 2019).
The *Single Best Solver* (*SBS*) is the best performing solver over all solvers in S. It describes the upper bound for an algorithm selector using state-of-the-art solvers. The gap between *SBS* and *VBS* gives an indication of the performance gains that can be realized using an algorithm selector (Kerschke et al. 2019).

2.1.4 Clustering

We will use clustering, utilizing the features F of an instance p_i to determine the best solver. This has been proven effective by Kadioglu et al. 2010 and achieved better performance than using the *SBS*. However, it is not clear if the cluster objective is the same as the

performance objective of selecting a fast solver for each instance. Ideally, instances with similar feature sets can all be solved quickly with a common fast solver. However, we can not assume that this is the case.
There are a multitude of different clustering algorithms that can be used to cluster the given data. In the following, we introduce the clustering algorithms used in this thesis:

K-Means The *K-Means* algorithm clusters a set of N samples X into k disjoint clusters $\mathcal{C} = \{C_1, ..., C_k\}$ with equal variance. This is achieved by minimizing the inertia criterion. The inertia (or within-cluster sum-of-squares criterion) is given as

$$\sum_{i=0}^{n} \min_{\mu_j \in C}(\|x_i - \mu_j\|^2)$$

where μ_i is the means of the samples in the cluster C_i. The inertia can be seen as a way to measure how internally coherent the clusters are. However, there are multiple drawbacks with this approach:
Inertia makes the assumption that the clusters are convex and isotropic. This is not always the case, and therefore *K-Means* can lead to bad results for irregular shapes.
Inertia is not a normalized metric. This can lead to problems in very high-dimensional space (as is the case in our problem). This problem can be reduced by running dimensionality-reduction algorithms. (scikit-learn 2021a)

Affinity Propagation *Affinity Propagation* creates its clusters by sending messages between pairs of samples until convergence. This enables *Affinity Propagation* to choose the number of the clusters based on the data provided. The main drawback is its complexity. The time complexity is $O(N^2T)$, where N is the number of samples and T the number of iterations until convergence (scikit-learn 2021a).

Mean Shift The *Mean Shift* algorithm tries to discover so-called “blobs” in a smooth density of samples. It works by updating the centroids to be the mean of the points in a given region. *Mean Shift* automatically sets the number of clusters, depending on the given samples. However, the algorithm is not highly scalable, because for each centroid calculation the algorithm needs multiple nearest-neighbor searches. (scikit-learn 2021a).

Spectral clustering The *Spectral Clustering* algorithm uses other clustering algorithm such as K-Means to cluster the components of the eigenvectors in a low dimensional space. The algorithm creates these eigenvectors with a low-dimension embedding of the affinity matrix of the samples. This can be very efficient if the affinity matrix of the samples is sparse.
Spectral clustering requires the number of clusters to be specified in advanced. It works well on a low number of clusters (scikit-learn 2021a).

Agglomerative clustering *Agglomerative clustering* describes a family of multiple clustering algorithms, which build nested clusters by successively merging and splitting them. Commonly, we use a Dendrogram to describe the hierarchy of clusters.
There are multiple Hierarchical clustering techniques:

- Ward: Minimizes the squared difference within all clusters. This makes it similar to the k-means algorithm, but is in this case used in combination with the hierarchical approach.
- Maximum / Complete linkage: Minimizes the maximum distance between observations of a pair of clusters.
- Average linkage: Minimizes the average of the distances between observations of a pair of clusters.
- Single linkage: Minimizes the distance between the closest observations of a pair of clusters.

(scikit-learn 2021a)

DBSCAN *DBSCAN* tries to create clusters by viewing them as areas of high density separated by areas with low density. This helps *DBSCAN* to identify clusters, that can have any shape, in contrast to clustering techniques like K-Means.
A cluster in *DBSCAN* is composed of core samples and non-core samples, that are close to the core samples. The two parameters `min_samples` and `eps` define the core samples. A sample is a core sample, if there exist `min_samples` in a distance of `eps` around it (scikit-learn 2021a).

Gaussian clustering *Gaussian clustering* uses a Gaussian mixture model which is a probabilistic model, that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions. Based on these mixture models, we sort the data points into clusters (scikit-learn 2021a).

OPTICS The *OPTICS* clustering algorithm is a relaxation of DBSCAN. Instead of a fixed `eps` value, *OPTICS* uses a value range.
The main difference is, that the *OPTICS* algorithm builds a reachability graph, which assigns each sample a reachability distance and a spot within the cluster ordering attribute. With these two attributes, it is decided to which cluster the sample belongs (scikit-learn 2021a).

BIRCH *BIRCH* builds a Clustering Feature Tree (CFT) for the given data. The given data is then compressed into a set of CF (Clustering Feature) Nodes. These CF Nodes contain multiple CF (Clustering Feature) subclusters, which in turn can contain CF Nodes as children. The CF subclusters contain the information that is needed for clustering. (scikit-learn 2021a).

2.2 Related work

There have been multiple different approaches to solving instances using a selection of given SAT-Solvers. One prominent approach is the one of a *parallel algorithm portfolio*, which runs multiple solvers in parallel on a problem instance. The *parallel algorithm portfolio* terminates the solvers as soon as one of the solvers has solved the given instance. By running a *parallel algorithm portfolio* on parallel hardware, one can achieve runtimes close to the *VBS*. Current research uses the term *algorithm portfolio* for both parallel algorithms and per-instance algorithm selectors. In this thesis, we will use the term *algorithm portfolio* only to describe *parallel algorithm portfolios* (Kerschke et al. 2019).

2.2.1 SATzilla

SATzilla is an automated approach for constructing per-instance *algorithm portfolios* for SAT that uses empirical hardness models to choose among their constituent solvers. It takes a distribution of problem instances and a set of solvers and constructs a portfolio optimizing a given objective function.
Important to note is, that *SATzilla* uses a mixture of a *parallel algorithm portfolio* and a per-instance algorithm selector. It first runs a selection of solvers in parallel on an instance until it reaches a fixed cutoff time, to solve easy instances. If the solver did not solve the instance, *SATzilla* extracts the features of the instances. Using the empirical hardness model, *SATzilla* predicts the best algorithm to run on the instance based on the features.
The effectiveness of *SATzilla* lies on the ability to learn empirical hardness models that can predict a solver's runtime on a given instance using effectively computable features.
Similar to *SATzilla* we aim to identify connections between instance features and the best solver. While *SATzilla* uses an empirical hardness model to predict the best solver it uses, as well as multiple pre-solvers run in parallel to determine the best solver, our goal is to use clustering on the instance features to identify clusters with connections of runtime behavior and instance features (Xu et al. 2008).

2.2.2 ISAC

Instance-Specific Algorithm Configuration (*ISAC*) has the approach to cluster the training instances based on a representative feature set. The assumption of *ISAC* is that, by clustering with these features, instances with similar underlying clusters are in the same cluster and can therefore be solved efficiently with the same solver. However, *ISAC* relies on a pre-specified set of features and an objective-oblivious clustering.
ISAC uses G-Means for the clustering. G-Means iteratively applies 2-Means clustering, accepting the partition only if the two new clusters are more Gaussian than their predecessor. After clustering the instances, an additional step merges all clusters smaller than a predefined threshold into their neighboring clusters. Using G-Means enables ISAC to select fast

solvers for given instances. We want to expand on the number of the clustering algorithms, to see if there are clustering algorithms that perform better.
ISAC uses the resulting clustering to run a parameter tuning algorithm, GGA, to generate the best parameters for a SAT-Solver on the instances in the cluster. We will not use parameter tuning. Instead, we evaluate the performance of different SAT-Solvers based on pre-calculated runtimes of the different solvers. Because of this, it is expected, that our resulting clusterings will not have the same performance as *ISAC* (Kadioglu et al. 2010).

2.2.3 SNNAP

Solver-Based Nearest Neighbor for Algorithm Portfolios (*SNNAP*) extends the idea of *ISAC* by using two observations.
The first observation is, that the addition of solver performance in the clustering in *ISAC* can be helpful, but disruptive, in combination with the original features.
Secondly, it is enough to just know the best two or three solvers for an instance.
Therefore, *SNNAP*s approach is to use a list of training instances to generate a prediction model for each solver that predicts the runtime of the solver on a given instance. These models are hard to train, as a misclassification can result in selection of the wrong solver. However, *SNNAP* does not aim to find the best solver, but just wants to know which solver behaves well on an instance.
For a new instance *SNNAP* uses the previously trained prediction models to predict the runtimes of the solvers on the new instance. *SNNAP* then uses these runtimes to calculate the distance between the new instance and every training instance and selects the nearest neighbor instances. Then the best solver for the new instances is chosen as the solver that performs best on the neighbors (Collautti et al. 2013).

3 Experimental design

This chapter is split into two sections. In section 3.1 we will introduce the steps of our experimental pipeline. It describes how the features of the instances are structured, the preprocessing done on them, as well as the methods used for clustering and evaluating the instances.
In section 3.2, we will describe the experimental setup, including our steps to filtering the resulting clusterings.

3.1 Experimental pipeline

Algorithm 1 shows the implementation of generating and evaluating the clusterings. The algorithm gets a list of instances I, the features F used in the current experiment and the solvers A. First, we scale the features F. After that, using the instances I and the scaled features $\bar{F}$, we use a clustering algorithm to create k clusters C. We then evaluate the clustering as a whole to get a performance score v of the clustering, as well as evaluating the single clusters, to get their respective Scores B. The scoring of the clustering and clusters aims to evaluate how good the clusters group instances together that have common fast and stable solvers. In section 3.1.4 we will discuss in detail how we will score the clusterings and clusters. We will use this algorithm to generate and evaluate multiple clustering algorithms with different parameter settings. This will give us a set of clusterings and their evaluations we can compare. In the following, we will discuss the important steps of the algorithm in more detail.

Algorithm 1: Algorithm for generating and evaluating a clustering

```
1 function SelectBestSolver(I, F, A)
2     F̄ ← Scale(F)
3     (k, C) ← GenerateClustering(I, F̄)
4     v ← ScoreClustering(k, C, A)
5     for i = 1, ..., k do
6         B_i ←ScoreCluster(A, C_i)
7     end
8     return (k, C, v.B);
9 end
```

3.1.1 Feature extraction

For given instances I, we extract the features using *cnftools*. The feature extraction is not part of the algorithm we presented. Instead, the feature extraction was performed in advanced by Iser, Springer, and Sinz 2020. *cnftools* creates the two feature sets: The '*base*' (56 features) feature set contains the feature cover degree distributions of well-known graph representations of a given instance and many more (Iser 2022). The '*gate*' (57 features) feature set contains the features cover gate distributions over levels of the (potentially recoverable) hierarchical gate structure of an instance (Iser 2022). Furthermore, we use the feature set '*runtimes*' (15 features) that was created by running the selected SAT-Solvers on the given instances I and contains the runtime for each solver on the instance. If the solver takes more than 5000s to solve the instance, we stop the solver and store 5000s as the runtime of the solver. We will therefore use 5000s as the timeout value. The features of '*runtimes*' are named after the solver that was run on the instances.
A feature that is available for each instance, but not used in the clustering, is the discrete feature of the family. It describes from which domain the problem originally stems. We list the features of each feature set in section A.3.

3.1.2 Preprocessing of features

The SAT-Instance provided by Iser 2022 contains a total of 2527 instances. Some of these instances have `empty` entries in '*base*', '*gate*' or '*runtimes*'. We therefore aim to remove all instances that contain `empty` entries from the dataset. However, for the solvers features *glucose_syrup* and *yalsat* there are many missing runtimes entries for the instances, which would decrease the number of instances in our dataset significantly. Based on that, we decided to exclude the features *glucose_syrup* and *yalsat* from the clustering and evaluation, leaving 13 runtime features. Using the remaining 13 '*runtimes*' features and '*base*' and '*gate*', there are three instances remaining, that contain `empty` entries, resulting in a dataset of 2524 instances.
Other than `empty` and numerical entries, there can be `memout`, `timeout` and `failed` entries. For '*base*' and '*gate*' we replace these values with the minimum integer value of the system. For '*runtimes*', we replace these entries with the timeout value (5000s).

For our experiments the feature sets '*base*', '*gate*' and '*runtimes*' have a total of 126 features. Because many clustering algorithms show deteriorated performance for high-dimensional data, we select different combinations of features for the clustering. These combinations were arbitrary, chosen based on the already given feature sets '*base*', '*gate*' and '*runtimes*'. We will discuss the options of feature selection in section 4.1.2 and why we decided against it.
The most common combinations we will use in the experiments are '*base*' (56 features), '*gate*' (57 features), '*runtimes*' (13 features), '*base gate*' (113 features), '*base runtimes*'

(69 features), '*gate runtimes*' (70 features), '*base gate runtimes*' (126 features).
Clustering algorithms that use distance metrics to calculate the clusters will give different weight to different features. To avoid that, the features will be scaled before the clustering. We discuss the selection of the used scaling technique in detail in section 4.1.1.

3.1.3 Clustering

Using scaled feature combinations of '*base*', '*gate*' and '*runtimes*', we apply different clustering algorithms with different parameters.
In most experiments, we use the same range of parameter values for the algorithms on our data. For the initial experiment to compare all presented clustering algorithms, we orientated our parameter settings for each clustering algorithm on the sklearn documentation (scikit-learn 2021a). We list the standard parameter ranges for each of the settings in section A.2. If we change the parameters for the algorithm in an experiment, the section of the experiment will describe these changes.

3.1.4 Evaluation measures

To evaluate the resulting clusterings, we will use two approaches of measuring. The first measure uses the runtime of the solvers on the dataset and clustering. For this, we will define the *Par2* and *SPar2* scores. The goal of the *Par2* score will be to identify solvers with fast runtime behavior on an instance set, while the *SPar2* score will measure how stable a solver is on an instance set. We will use these scores to define multiple solvers to evaluate the dataset as well as clusterings and clusters based on the dataset.
The second measure compares the generated clusters with existing groupings of our data and measures how similar they are.
Furthermore, we will add a method that determine solvers of similar performance on a cluster.

Scores

Par2-Score The Par2-Score is a score used in SAT competitions (Heule, Jarvisalo, and Balyo 2017).
Given a set of instances $I = \{p_1, ..., p_n\}$, a set of solvers $A = \{s_1, ..., s_m\}$, and a metric $m : A \times I \to \mathbb{R}$ that measures the time it takes any solver $s_j \in A$ to solve the instance $p_i \in I$, we can define the helper function $\mathsf{t}_{\mathsf{par2}}$ as

$$\mathsf{t}_{\mathsf{par2}}(s_j, p_i) = \begin{cases} m(s_j, p_i) & \text{if } m(s_j, p_i) < T \\ T \cdot 2 & \text{otherwise} \end{cases}$$

where T is the timeout value. Using t_{par2}, we define the Par2 score of a solver s_j for a set of instances I as

$$\mathsf{Par2}(s_j, I) = \frac{1}{|I|} \sum_{i=1}^{n} \mathsf{t_{par2}}(s_j, p_i)$$

Stability-adjusted Par2 (SPar2) The *Par2* score only measures the performance of a solver s_j on a set of instances I. It makes no statement about the stability of a solver s_j when run on the instances I. When designing a score to evaluate the stability of a solver, we need to consider the following:
For a solver to be considered stable, we want the runtimes for a set of instances I to be in a similar range.
A problem we face is, that a solver that has timeouts on many or all instances of a set of instances could be considered to have similar runtimes on all instance. But when a timeout occurs, we do not know how long the solver would have taken to solve the instances. It might have finished quickly after the timeout, but it could also have taken considerably longer. Because we do not know this, we can not consider a solver that has timeouts on many or all instances stable. Our proposed solution is to penalize timeouts in the stability score.

Given the instances $I = \{p_1, ..., p_n\}$, we split I into the sets M and N, where $M = \{p_i | p_i \in I \land m(s_j, p_i) < T\}$ and $N = \{p_i | p_i \in I \land m(s_j, p_i) \geq T\}$. T is the timeout value and the metric $m : A \times I \rightarrow \mathbb{R}$ gives the time it takes the solver s_j to solve the instance $p_i \in I$. This means M contains all instances that the solver s_j can solve before a timeout T occurs and N all instances where the solver s_j has a timeout. It applies $M \cup N = I$. We define the *SPar2* score as

$$\mathsf{SPar2}(s_j, I) = \frac{|M| \cdot \mathsf{mad}(s_j, M) + |N| \cdot T \cdot 2}{|M| + |N|}$$

where mad is the mean absolute deviation of the instances in M given as

$$\mathsf{mad}(s_j, M) = \frac{1}{|M|} \sum_{p_i \in M} |m(s_j, p_i) - \mathsf{mean}(s_j, M)|$$

with

$$\mathsf{mean}(s_j, M) = \frac{1}{|M|} \sum_{p_i \in M} m(s_j, p_i)$$

Solvers

Using the *Par2* score and *SPar2* score, we can define multiple solvers with different characteristics on our dataset and clusterings:

Virtual Best Solver The *Virtual Best Solver* (*VBS*) chooses the best solver for each instance in the set of instances $I = \{p_1, ..., p_n\}$. The *VBS* is calculated as

$$\mathsf{VBS}(I) = \frac{1}{|I|} \sum_{p_i \in I} \min_{j \in \{1,...,m\}} \mathsf{Par2}(s_j, \{p_i\})$$

Single Best Solver The *Single Best Solver* (*SBS*) chooses the best solver over all instances in the set of instances $I = \{p_1, ..., p_n\}$. The *SBS* is calculated as

$$\mathsf{SBS}(I) = \min_{j \in \{1,...,m\}} \mathsf{Par2}(s_j, I)$$

Clustering Best Solver To evaluate a clustering, we introduce the *Clustering Best Solver* (*CBS*) that uses the *Par2* score. The *CBS* describes the *Par2* score if we choose the best solver in each cluster to solve the instances of I. This enables us to measure how well a clustering algorithm created clusters, that group instances with common fast solvers together.

The *CBS* is the solver induced by a clustering $\mathcal{C} = \{C_1, ..., C_k\}$, where $\forall i \in \{1, ..., k\} : C_i \subseteq I$, $I = \bigcup_{i=1}^{k} C_i$ and $\forall i, j \in \{1, ..., k\} \wedge i \neq j : C_i \cap C_j = \emptyset$. This means each instances in I is in a cluster C_i.
The *CBS* of clustering $\mathcal{C}$ is calculated as

$$\mathsf{CBS}_{\mathcal{C}}(I) = \frac{1}{|I|} \sum_{i=1}^{k} |C_i| \min_{j \in \{1,...,m\}} \mathsf{Par2}(s_j, C_i)$$

where $|C_i|$ is the number of instances in the cluster C_i.

Single Best Stability-adjusted Solver The *Single Best Stability-adjusted Solver* (*SBSS*) is given by the solver s_j, that has the lowest *SPar2* score over all instances in I. It is calculated as

$$\mathsf{SBSS}(I) = \min_{j \in \{1,...,m\}} \mathsf{SPar2}(s_j, I)$$

Clustering Best Stability-adjusted Solver Analogous to the *CBS*, we can define a *Clustering Best Stability-adjusted Solver* (*CBSS*) based on the *SPar2* score. The *CBSS* of clustering $\mathcal{C}$ is calculated as

$$\mathsf{CBSS}_{\mathcal{C}}(I) = \frac{1}{|I|} \sum_{i=1}^{k} |C_i| \min_{j \in \{1,...,m\}} \mathsf{SPar2}(s_j, C_i)$$

Naming Conventions *VBS*, *SBS* and *SBSS* can be calculated for an arbitrary subset of our dataset instances. When referring to the *VBS*, *SBS* and *SBSS* in this thesis, we talk about the scores of all instances in the dataset.
If we calculate the *VBS*, *SBS* or *SBSS* for the instances in a cluster C_i, we call it the *Cluster Virtual Best Solver* (*CVBS*), *Cluster Single Best Solver* (*CSBS*) or *Cluster Single Best Stability Adjusted Solver* (*CSBSS*) to differentiate between the *VBS*, *SBS* and *SBSS* of all instances and individual clusters.
Please note that the *Clustering Best Solver* (*CBS*) and *Clustering Best Stability-adjusted Solver* (*CBSS*) describe scores on the complete clustering, while the *Cluster Single Best Solver* (*CSBS*) and *Cluster Single Best Stability Adjusted Solver* (*CSBSS*) describe scores on single clusters.

Measuring the similarity of clusterings

Another way to evaluate clusterings is to compare them to other clusterings. We will use this to evaluate the difference between the clusterings when using different parameters for clustering algorithms, as well as similarity to the families of each instance.

Mutual Information Mutual information is based on entropy. The entropy of a clustering $\mathcal{C} = \{C_1, ..., C_k\}$ is defined as

$$\mathcal{H}(\mathcal{C}) = \sum_{i=1}^{k} P(i) \log_2 P(i)$$

where $P(i)$ describes the probability that an element is in the cluster C_i of $\mathcal{C}$.

$$P(i) = \frac{|C_i|}{n}$$

$|C_i|$ is the number of instances in the cluster C_i and $n = \sum_{i=1}^{k} |C_i|$. It measures the uncertainty of the cluster of a randomly picked element. We can extend this measure to compare two clusterings based on the same set of elements. Mutual information describes how much we can on average reduce the uncertainty about the cluster of a random element when knowing its cluster in another clustering of the same set of elements. Mutual information between two clusterings $\mathcal{C}$ and $\mathcal{C}'$ can be defined as

$$\mathcal{I}(\mathcal{C}, \mathcal{C}') = \sum_{i=1}^{k} \sum_{j=1}^{l} P(i,j) \log_2 \frac{P(i,j)}{P(i)P(j)}$$

where $P(i,j)$ is the probability that an element belongs to cluster C_i in $\mathcal{C}$ and to cluster C'_j in $\mathcal{C}'$.

$$P(i,j) = \frac{|C_i \cap C'_j|}{n}$$

The mutual information in itself is hard to interpret, because it is bounded by the entropies of the two clusters.

$$\mathcal{I}(\mathcal{C},\mathcal{C}') \leq \min\{\mathcal{H}(\mathcal{C}),\mathcal{H}(\mathcal{C}')\}$$

(S. Wagner and D. Wagner 2007)

Normalized mutual information To make the mutual information easier to interpret, we normalize it using the arithmetic mean of the entropies (Fred and Jain 2003). There are propositions using the geometric mean (Strehl and Ghosh 2002). However, we will use the arithmetic mean.

$$\mathsf{NMI}(\mathcal{C},\mathcal{C}') = \frac{2\mathcal{I}(\mathcal{C},\mathcal{C}')}{\mathcal{H}(\mathcal{C}) + \mathcal{H}(\mathcal{C}')}$$

This ensures that we have

$$0 \leq \mathsf{NMI}(\mathcal{C},\mathcal{C}') \leq 1$$

(S. Wagner and D. Wagner 2007)

Cluster Solver Strip So far, we only looked at a single best solver determined by the *SBS* or *SBSS* for a set of instances. For a cluster, we are interested if there is a set of solvers that perform well on the cluster. We will determine this set of solvers by using the *Par2* score of the *CSBS* as a guiding value, which decided whether other solvers are in the set or not. We determine the solver set S over the solvers $A = \{s_1, ..., s_m\}$ and the cluster C_i as follows

$$S = \{s_j | s_j \in A \wedge \mathsf{Par2}(s_j, C_i) \leq (\mathsf{SBS}(s_j, C_i) + \mathsf{offset}) \cdot \mathsf{gradient}\}$$

We choose a $\mathsf{gradient} = \frac{11}{10}$ and $\mathsf{offset} = 3$ for our evaluations. In Figure 3.1, we can see the linear function showing the runtime of the *SBS* as well as the upper bound for solvers in the set S that is determined by the *SBS* with the formula $(\mathsf{SBS}(s_j, C_i) + \mathsf{offset}) \cdot \mathsf{gradient}$. Using this strip, we can determine sets of solvers that perform similar on the clusters, even if their runtimes vary more for clusters with higher *Par2* scores.

The goal of this measure is to consider the higher variance of runtimes when they increase. A fixed difference makes it hard to evaluate if solvers behave similar on instances of different difficulty. The strip solves this, by increasing the difference with the increase of runtime.

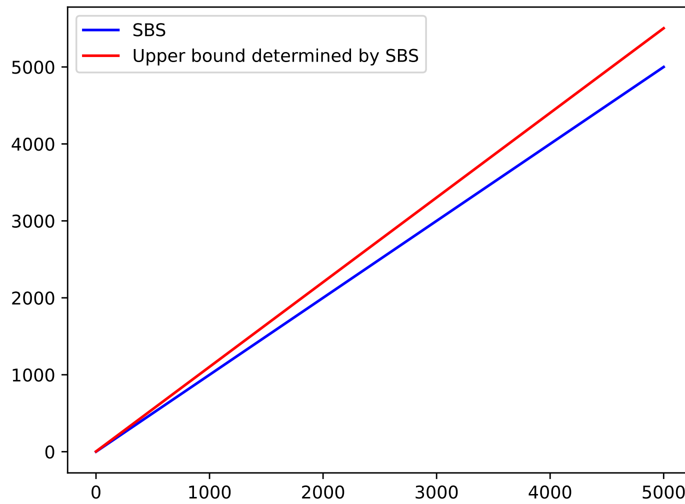

Figure 3.1: Strip determining which solvers are in the set M

Values of the scores

The *SBS*, *SBSS*, *VBS* and *VBSS* are measurements not influenced by the clustering. We can therefore calculate these scores for our dataset. You can see the scores in Table 3.1.

Score	Value (s)	Solver
SBS	3947.35	kissat
SBSS	3973.85	kissat
VBS	2875.94	-

Table 3.1: Dataset scores

The *SBSS* shows us, that there is high standard deviation when it comes to the runtimes of all instances using *kissat*. This is expected, as *kissat*'s runtimes range from 0s to 5000s. We can also observe that *SBS* and *VBS* have a difference of more than 1100s. This shows that creating a good algorithm selector can improve the performance significantly in comparison to the *SBS*.

Furthermore, we calculate the *Par2* score and *SPar2* score for each solver over all instances (sorted by *Par2*). The values can be seen in Table 3.2.

Solver	Par2 (s)	SPar2 (s)
kissat	3947.35	3973.85
relaxed	4542.67	4516.43
cadical-pripro	4625.24	4607.84
cadical	4822.18	4811.75
cadical-stability	4956.04	4936.42
cadical-elimfalse	5097.31	5088.55
lingeling	5600.46	5608.42
glucose	5828.79	5847.76
glucose-chanseok	5888.57	5911.71
glucose-var-decay099	6052.66	6067.90
candy	6195.15	6193.27
minisat	6800.75	6810.66
march-nh	8802.59	8814.13

Table 3.2: *Par2* and *SPar2* scores for each solver

While we sorted the solvers after the *Par2* score, we can observe, that the *SPar2* score is sorted as well, indicating that the best performing solvers might also the most stable solvers.

3.2 Experimental setup

This section gives an overview over the environment and libraries used for the experiments as well as discusses filter parameters used to determine clustering algorithms and clusters that show high performance with our evaluation measures. It further introduces multiple dataset characteristics used in the following experiments.

3.2.1 Environment

We used different machines for each evaluation of the pipeline based on the computing power needed. However, the only case in which the result should differ when using other machines is the calculation of the feature sets for the instances. We split the pipeline between machines as follows:
The feature extraction was done by Iser, Springer, and Sinz 2020. The preprocessing and clustering was performed on a machine with 32 CPU Cores, 2.00 GHz, 128 GB RAM. The Evaluation was performed on a machine with 12 CPU Cores, 3.80 GHz, 16 GB RAM, RTX 2070S.

3.2.2 Libraries

The pipeline uses multiple different libraries. For many math operations, we use *numpy* (1.22.1). The *gbd-tools* (3.6.5) library (Iser, Springer, and Sinz 2020) is used to read the feature sets from the given databases. For preprocessing, clustering and evaluation we use *scikit-learn* (1.0). The elbow calculations in section 3.2.4 use the *yellowbrick* (1.3.post1) library. Furthermore, we use *matplotlib* (3.4.3) and *pandas* (1.3.4) for generating the figures of this document.
The codebase, written for data exploration, also enables the search for clusterings and clusters using pareto-optimal points. For this we use *OApackage* (2.7.1). However, we do not use pareto-optimal points in this thesis.

3.2.3 Filter parameters

We will start out using the 7 feature set combinations and multiple different cluster algorithms with different parameter settings. This yields a high number of clusterings that we can examine. To be able to find the answers to our research questions, we will use multiple steps of filtering to reduce the number of clusterings. Each filter step will be justified in the dedicated section.
To get an overview of the following filtering steps, we will explain them here. These can be split into the two parts of preprocessing and the comparison of feature sets and clustering algorithms:

Preprocessing Before running the cluster algorithms, we determine the scaling applied to the selected features (section 4.1.1) and the feature set we cluster on (section 4.1.2).

Comparing feature sets and clustering algorithms In the next step we will evaluate the clusterings created by different combinations of feature sets, clustering algorithms and their parameters. For this, we will use the following steps to filter out clusterings:
First, we filter out clusterings by number of clusters. While many clusters have settings for the amount of clusters, others do not and therefore can produce any number of clusters. We will therefore omit all clusterings with more clusters than a threshold we will determine in section 3.2.4
Next, we filter by the combination of feature set used for the clustering. The remaining clusterings will be used to determine clustering algorithms that perform well.
We will observe the behavior of the selected clustering algorithms, when changing parameter values, and determine parameter values for each cluster algorithm (section 4.2.2).
Lastly, using the clustering algorithms and the selected parameters, we select one clustering for each clustering algorithm and examine the clusters they contain.

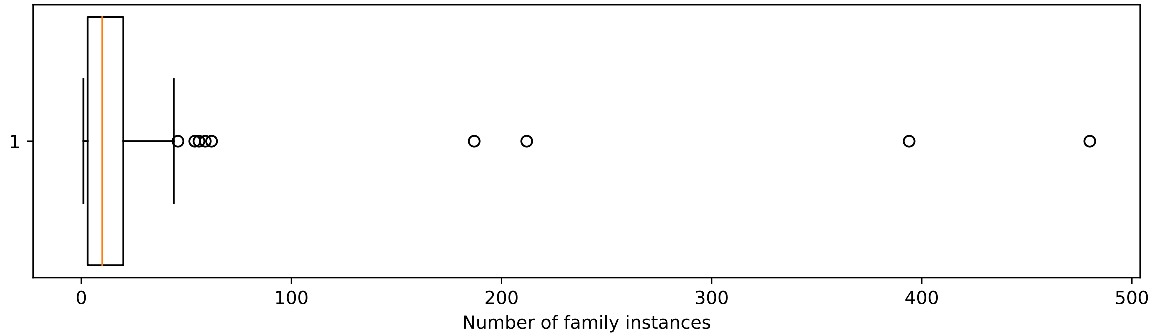

Figure 3.2: Distribution of family occurrences

3.2.4 Dataset characteristics

To choose useful parameter ranges and compare the results, we introduce multiple measurements derived from the dataset. We calculated these values on the remaining 2524 instances of the dataset, where we excluded the runtimes of *yalsat* and *glucose_syrup* and subsequently removed all other instances with missing values.

Instance families The dataset contains instances of 83 different families. These families have very different sizes, as can be seen in Figure 3.2. The family with the most occurrences is *cryptography*, with 483 instances. However, there are multiple families with only one occurrence. It is therefore hard to determine what good cluster sizes could be.

Number of clusters Another aspect we might want to use for selecting clusterings is the number of clusters a clustering contains. Taking the number of families as a benchmark value, we could say that using more than 83 clusters in a clustering unnecessarily splits families. However, we do not know if instances from the same family share the same best solver. We will use the elbow-method in combination with *K-Means* to get an estimation for an optimal number of clusters. The elbow-method was performed on all combinations of '*base*', '*gate*' and '*runtimes*', using standard scaling (section 4.1.1) for preprocessing and a range of $k \in \{1, .., 100\}$ for *K-Means*. The elbow method then measures the resulting K-Means clusterings based on the *distortion score* (the sum of squared distances from each point to its assigned center). The resulting values can be seen in the Table 3.5.

Except '*runtimes*', all values are close to a value $k \in \{20, .., 30\}$. However, the '*runtimes*' elbow is the most pronounced when it comes to an elbow shape, as Figure 3.3 shows. In comparison to that, the shapes of the other combinations, such as '*base gate*' (Figure 3.4) are not as clear. Because of this, we need to consider, that the actual optimal number of

clusters might be higher. We will use a value of 35 for the maximum number of clusters in a clustering. Unless stated otherwise, we will, filter clusterings with more than 35 clusters.

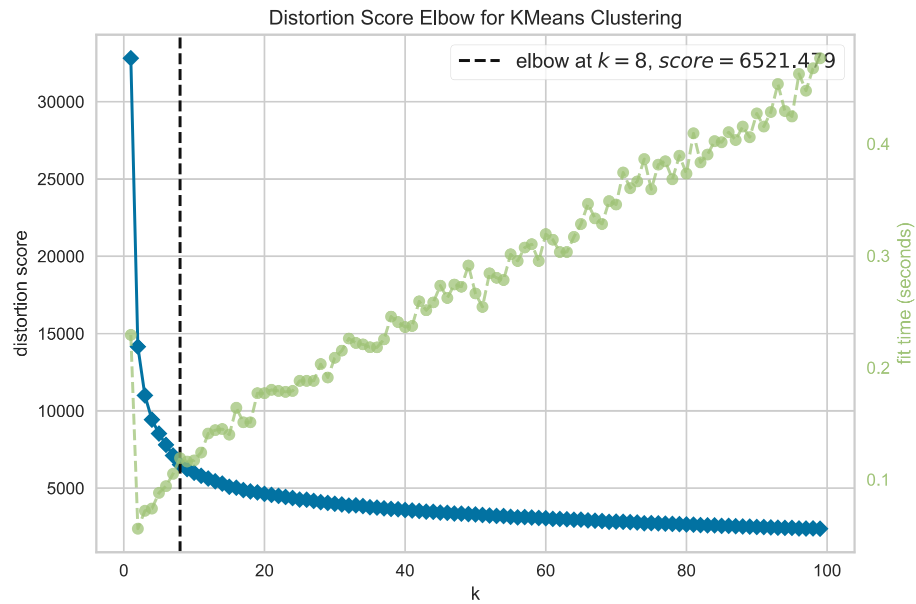

Figure 3.3: Distortion Score using '*runtimes*'

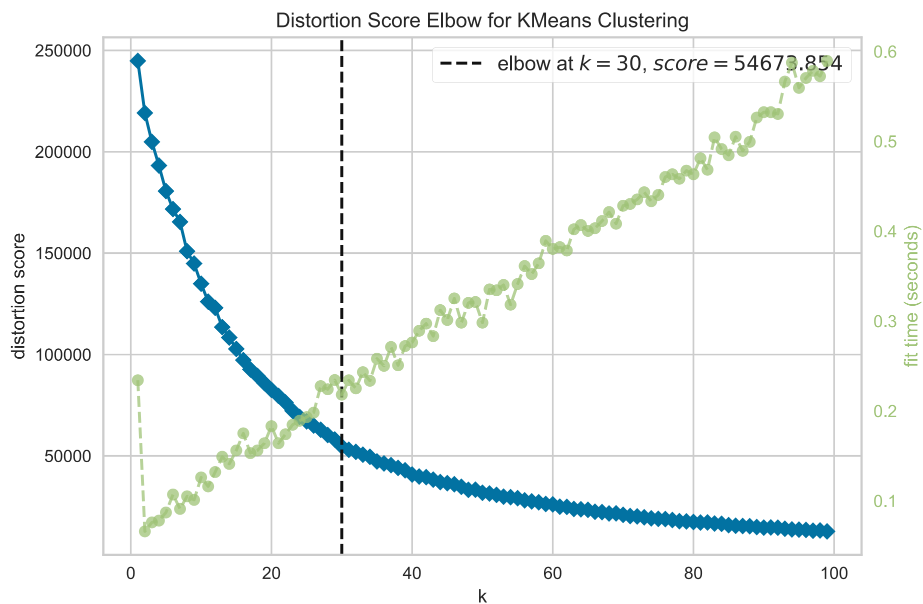

Figure 3.4: Distortion Score using '*base gate*'

Elbow-Method		
Combination	Elbow at k	Distortion Score
'*base*'	16	37251.664
'*gate*'	23	13086.134
'*runtimes*'	8	6521.479
'*base gate*'	30	54673.854
'*base runtimes*'	16	55163.280
'*gate runtimes*'	23	29442.259
'*base gate runtimes*'	27	81360.347

Figure 3.5: Results of the elbow method for all feature set combinations

4 Experimental evaluation

This chapter is split into 2 sections. In the first section 4.1 we will present experiments to explain our choice of preprocessing the data. In the second section 4.2, we will evaluate clustering algorithms and selected clusterings.

4.1 Preprocessing of the data

In this section, we will discuss two important preprocessing steps before clustering. We first give reason on the scaling method we use on the features in section 4.1.1. After that, we will examine the single features to decide how we will select feature sets for clustering in section 4.1.2.

4.1.1 Scaling

Scaling the data is an important step before applying clustering algorithms. In this section, we compare two scaling methods to decide which one to use in further experiments.

Standard Scaling: Given a feature vector $f = (x_1, ..., x_n)^\top$, we calculate the standard score z_i of a sample x_i as

$$z_i = \frac{x_i - \mu}{\sigma}$$

where $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$. (scikit-learn 2021b)

Linear Scaling to [-1,1]: Given a feature vector $f = (x_1, ..., x_n)^\top$, we calculate the scaling l_i to [-1, 1] of a sample x_i as:

$$l_i = \frac{x_i - c}{d}$$

with

$$c = \frac{\max_{i\in\{1,...,n\}} f + \min_{i\in\{1,...,n\}} f}{2}$$
$$d = \frac{\max_{i\in\{1,...,n\}} f - \min_{i\in\{1,...,n\}} f}{2}$$

To test the performance of both scaling algorithms, we scaled the combinations of the feature sets '*base*', '*gate*' and '*runtimes*' with both scaling algorithms and then applied the

same clustering algorithms. The resulting clusterings were evaluated using the *CBS*.

Figure 4.1 shows the distribution of the *CBS* Scores over all combinations of feature sets. Both linear scaling and standard scaling have a majority of clusters that are rated with *CBS* scores above 3800s and are close to the *SBS* (3947.35s). In comparison to that, the *VBS* (2875.94s) is far away from the best clusterings.

In Figure 4.2, we can see the distributions of the clusters for only '*base*', '*gate*' and '*base gate*'. The best *CBS* score has decreased in comparison to Figure 4.1, but the general distribution of the clusters stayed the same.

For both Figure 4.2 and Figure 4.1, linear scaling produces more clusters with lower *CBS* scores than standard scaling, however, both have similar low outliers.

We decided to use standard scaling for the further experiments, because it behaves better on features with outliers. For the linear scaling, outliers can have the effect of moving the outliers close to -1, while it moves all other values very close to 1. This makes it hard when clustering to determine whether instances with different values of this feature should be in the same cluster or be split. Standard scaling does not experience this effect as strongly. While linear scaling has more clusterings with lower *CBS* scores, both have outliers with low *CBS* scores. The analysis of the single clustering outliers should therefore yield similar results for both scaling techniques.

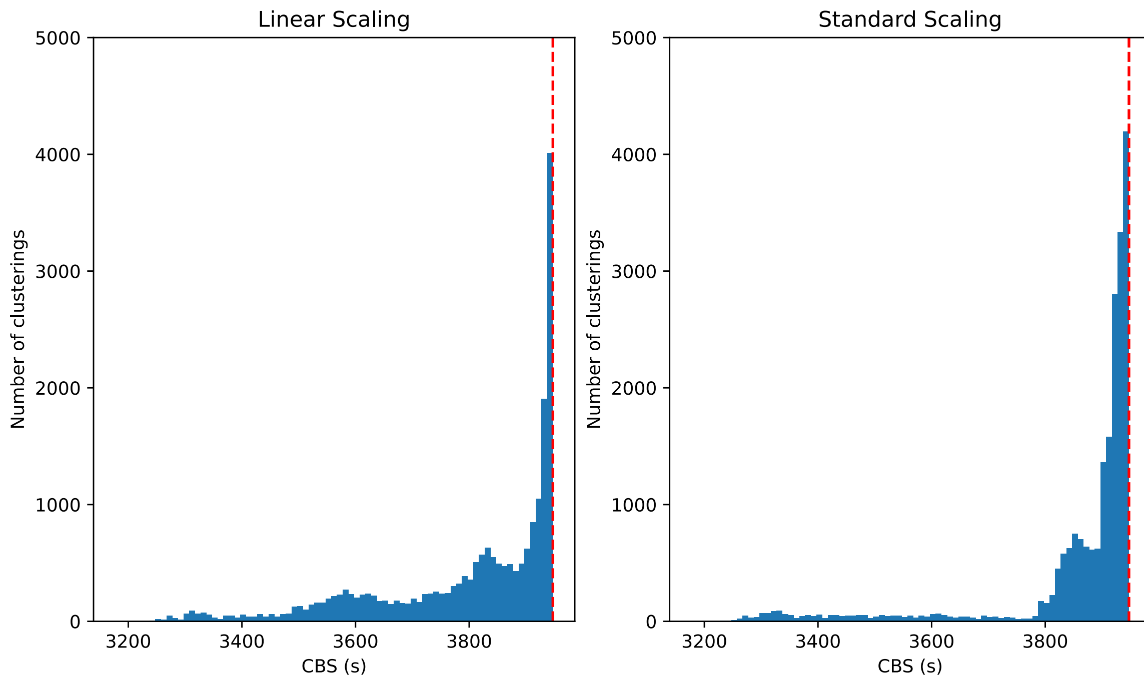

Figure 4.1: Histogram of the distribution of *CBS* scores for linear and standard scaling with all feature sets

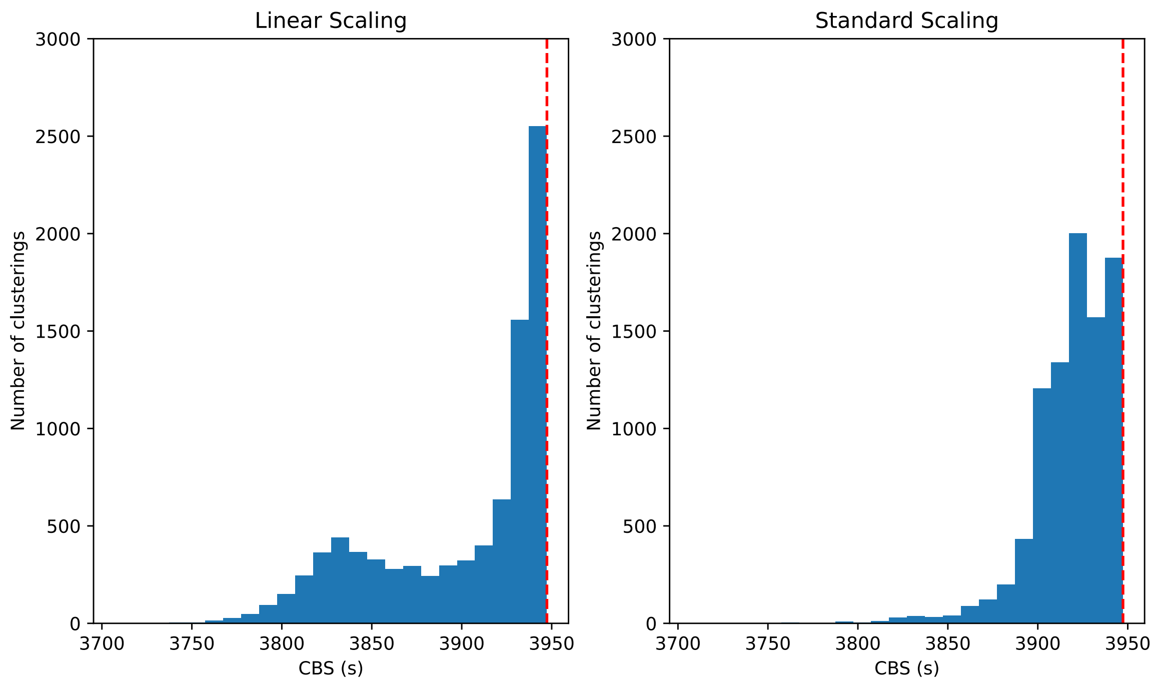

Figure 4.2: Histogram of the distribution of *CBS* scores for linear and standard scaling with '*base*', '*gate*' and '*base gate*'

4.1.2 Clusterings with single features

Another important step before applying clustering algorithms to the data is feature selection. In this step, we want to identify the useful feature vectors used in the clustering. Excluding unnecessary feature vectors can lead to an improvement in the resulting clusterings. One method is to create the feature set with a greedy approach, by iteratively adding features to the feature set used for clustering and then evaluating the resulting clustering. For this, we need a ranking of the performance of each feature when generating a clustering to select the next best feature. One way to get this ranking is by clustering each feature separately.

In this experiment we clustered each feature from the three feature sets '*base*', '*gate*' and '*runtimes*' separately and evaluated their performance using the *CBS* score. To limit the number of clusterings we only used *DBSCAN*, *K-Means* and *Agglomerative* clustering. For clustering, we use the default settings. The result of each feature set can be seen in the three Figures 4.3, 4.4 and 4.5. Features that did not have a clustering with *CBS* scores that were at least 25s faster than the *SBS* were removed from the figures. The y-axis, displaying the *CBS* score, shows the same interval for each figure to make the values more easily comparable.

In Figure 4.3, no clustering of '*base*' features, manages to lower the *CBS* score below 3800s. The best performing features for '*base*' are `clause_size_1`, `clause_size_7`, `positive_clauses`, `balance_clause_entropy` and `vcg_vdegrees_entropy`.
'*gate*' (Figure 4.4) performs similar to '*base*'. The only feature with clusterings close to 3800s is `n_roots`. Also '*gate*' experienced the removal of more features than '*base*', because they showed no or very small improvement in comparison to the *SBS*.
'*runtimes*' (Figure 4.5) performs better in comparison to '*base*' and '*gate*'. Especially, the *SBS kissat* performs well when used for clustering. However, we will see in the section 4.2.1 that using all '*runtimes*' features for clustering proves more effective.

In conclusion, we can see that a greedy approach based on the '*base*' and '*gate*' features would be hard, because all features have very similar *CBS* scores for their resulting clusterings. This makes it hard to select features in a greedy approach. Because of this, we will use the seven combinations of the feature sets '*base*', '*gate*' and '*runtimes*' presented in section 3.1.2.

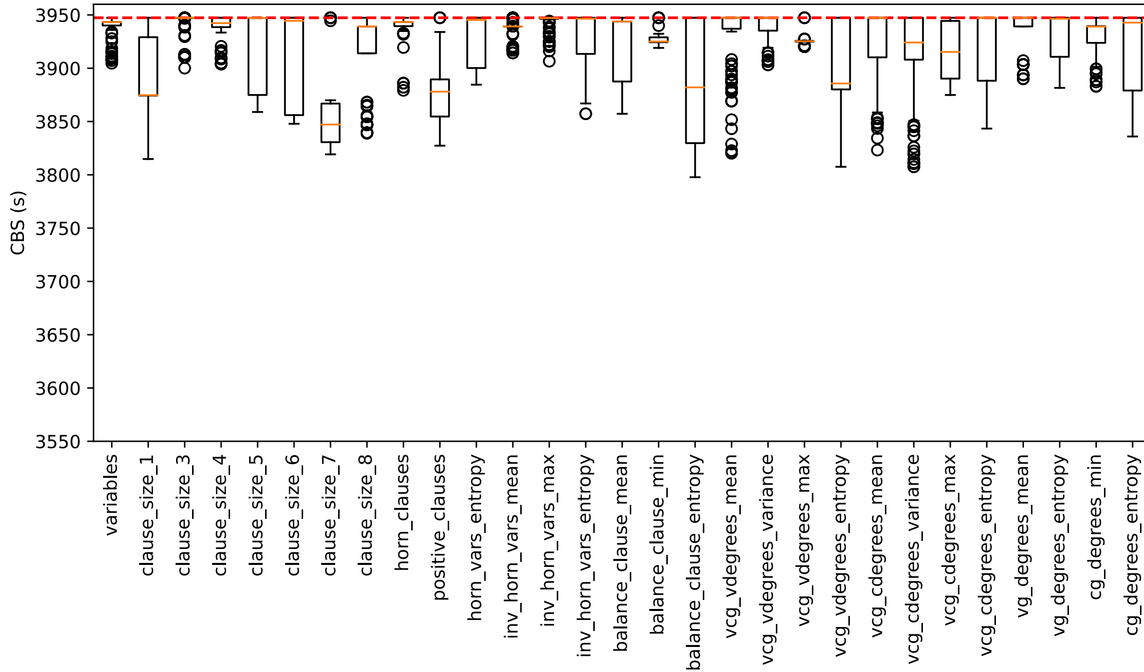

Figure 4.3: *CBS* score of each '*base*' feature

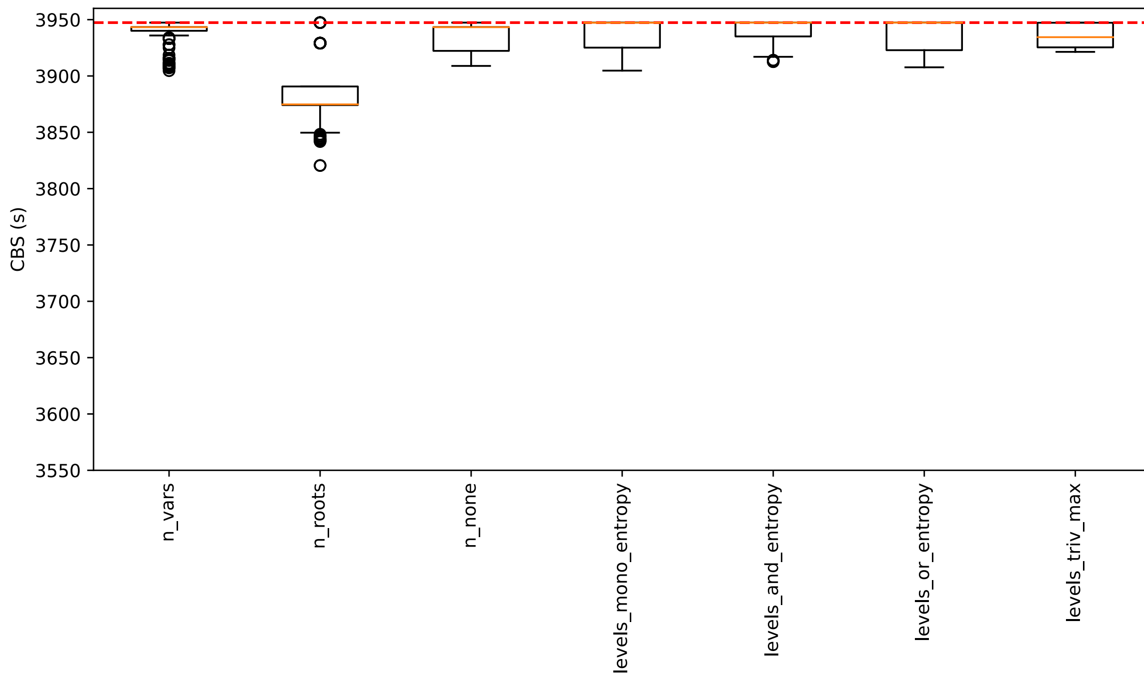

Figure 4.4: *CBS* score of each '*gate*' feature

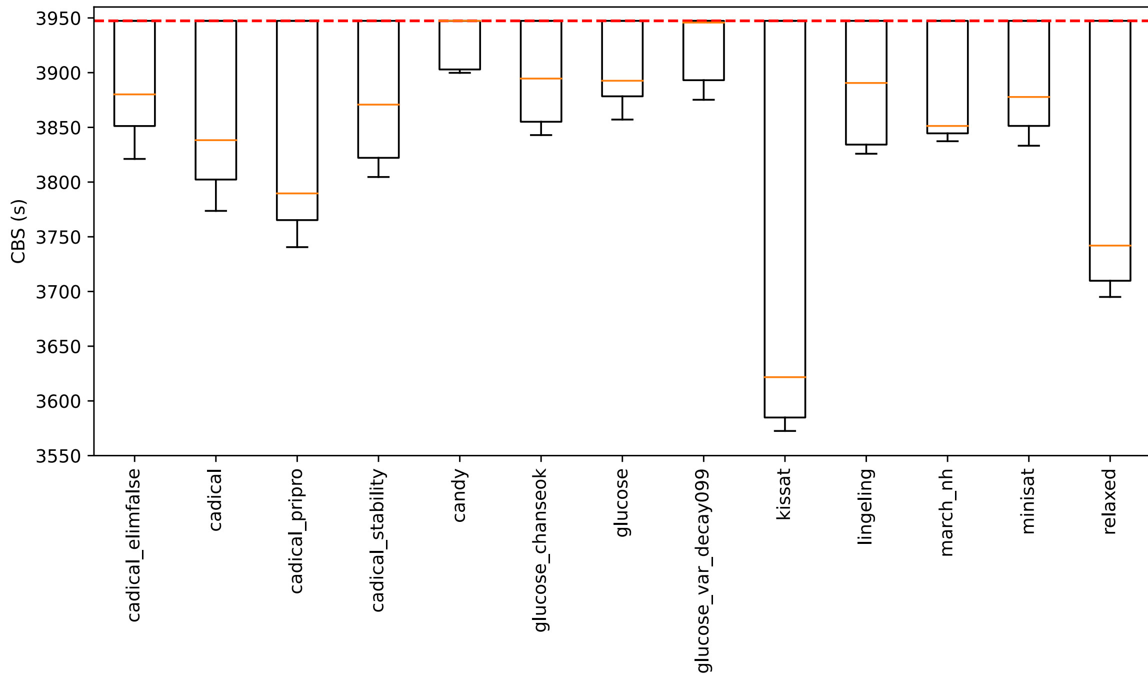

Figure 4.5: *CBS* score of each '*runtimes*' feature

4.2 Comparing feature sets and clustering algorithms

In the previous sections, we decided to use combinations of the '*base*', '*gate*' and '*runtimes*' feature sets in combination with standard scaling for clustering. We used these combinations to run different clustering algorithms, with a parameter range for each. The parameter ranges of each clustering algorithm can be found in section A.2. However, the range of `eps` for *DBSCAN* was increased to [0.1, 3] for the following experiments, to allow *DBSCAN* to create clusterings with a lower number of clusters. We decided in section 3.2.4 that clusterings with more than 35 clusters show no significant improvement. Therefore, clusterings with more than 35 clusters will be filtered out of the following plots.
This chapter is split in three parts. First, we will look at all clusterings created with the combinations of feature sets and cluster algorithms (section 4.2.1). Next, we will choose a limited number of feature sets and clustering algorithms and determine the behavior of the clustering algorithms when changing their parameters (section 4.2.2). Lastly, we will select interesting clusterings and examine them in detail (section 4.2.3).

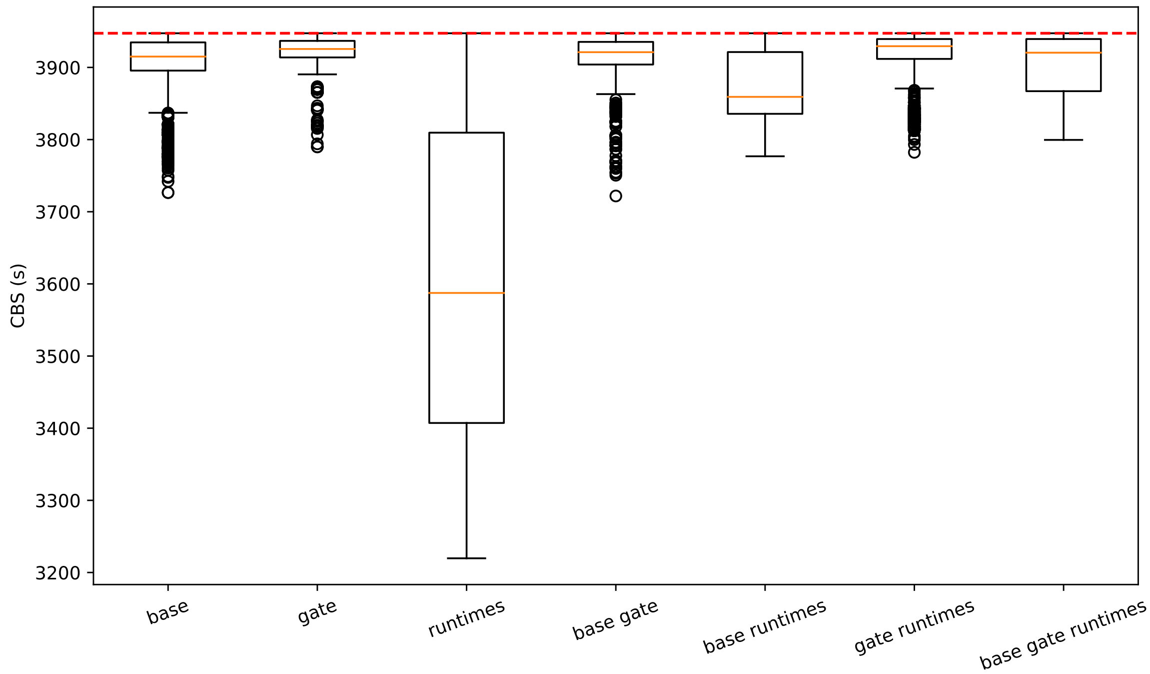

Figure 4.6: Histogram of the distributions of the *CBS* scores for each combination of features

4.2.1 Evaluating clustering algorithms

In Figure 4.6, we can see the *CBS* scores for each combination of feature sets. The clusterings using only '*runtimes*' have the lowest *CBS* scores. In comparison to this, the other combinations of feature sets perform worse, having their median always above 3800s. On the other hand, while '*base*', '*gate*' and '*base gate*' have very high median *CBS* scores, there are some outliers that manage lower *CBS* scores. Including '*runtimes*' with '*base*', '*gate*' or '*base gate*' seems to improve the median performance of the *CBS* scores of the clusterings. However, the best *CBS* scores decrease (except for '*gate runtimes*'). Overall the inclusion of '*runtimes*' into these combinations does not have a big positive effect as we may expect when looking at the clustering created when using only '*runtimes*'.

The clusterings using the '*runtimes*' feature set, performs better because the clusterings have access to the runtimes for each solver. Therefore, each cluster should contain problem instances that have similar solver runtimes for the selection of solvers in '*runtimes*'. This leads to creation of clusters in which problem instances can be solved quickly by the same solver. This effect gets eased when '*runtimes*' is used in combination with '*base*', and '*gate*' explaining the increase of *CBS* scores.
The features in '*base*' and '*gate*' do seem to contain much less information regarding the formation of clusters with fast solvers. Most of their *CBS* scores are only slightly below the *SBS* score.

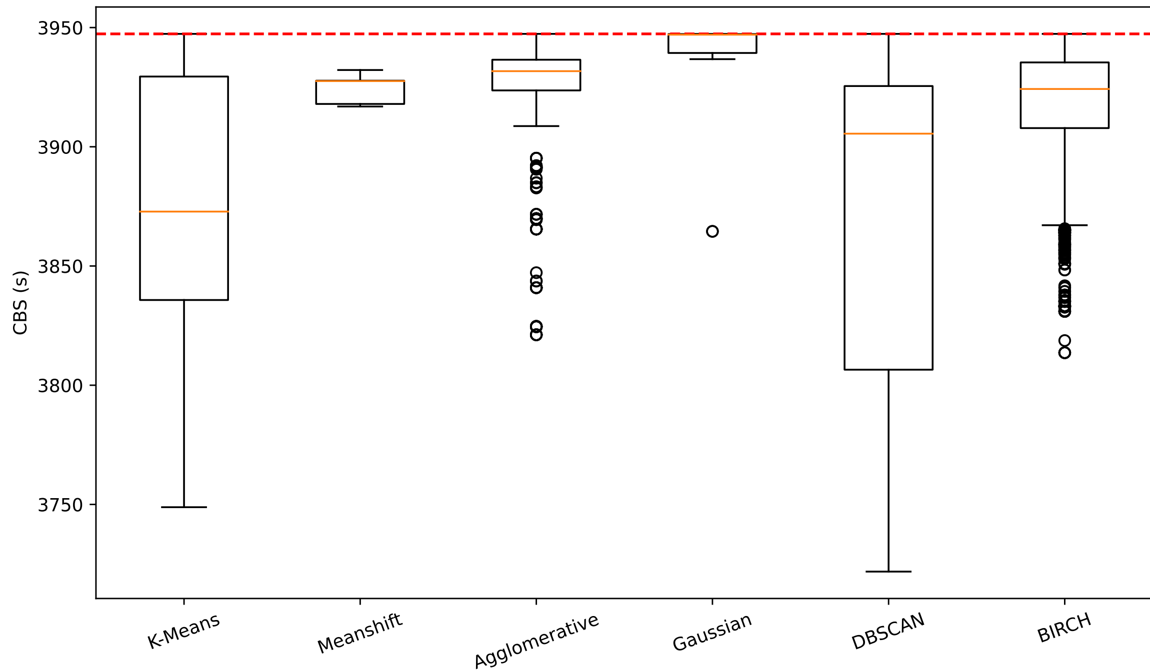

Figure 4.7: Histogram of the distributions of the *CBS* scores for clustering algorithms using '*base*', '*gate*' and '*base gate*'

Next, we examine how the clusterings are distributed between the clustering algorithms we used. We saw, that clustering with '*runtimes*' creates many of the clusterings with low *CBS*. This would skew the results, because we are interested what clustering algorithms perform well on features we can extract from a new instance quickly. Calculating runtimes on a new instance, however, is time-consuming and would make the clustering unnecessary. Furthermore, we saw that the inclusion of '*runtimes*' to '*base*', '*gate*' or '*base gate*', did not improve the clusterings by much. Therefore, we will continue by only using clusterings created with '*base*', '*gate*' or '*base gate*'.
In Figure 4.7 we can see the distribution of the clusterings created with '*base*', '*gate*' or '*base gate*' split into the different clustering algorithms used. Furthermore, all clustering algorithms that had no clusterings with *CBS* scores that differed more than 25s from the *SBS* were removed. Figure 4.7 shows us, that *K-Means* and *DBSCAN* perform the best regarding low *CBS* scores. Both *Agglomerative clustering* and *BIRCH* show outliers with low *CBS* score, but in comparison to *K-Means* and *DBSCAN* their median is much higher.

In conclusion, '*runtimes*' manages to create the best clusters. However, this is not very useful, as knowing the runtimes of the instances we want to sort into a cluster beforehand makes the goal of the clustering unnecessary. The combinations '*base*', '*gate*' and '*base gate*' all have high *CBS* scores and all perform very similar, with small differences in their

outliers and median. The combinations '*base runtimes*', '*gate runtimes*' and '*base gate runtimes*' show no significant advantage over their counterpart '*base*', '*gate*' and '*base gate*' without runtimes.
When using '*base*', '*gate*' or '*base gate*' for clustering, both *K-Means* and *DBSCAN* create the best clusterings in comparison to other clustering algorithms.

4.2.2 Analysis of selected clustering algorithms

In the previous section, we realized that *K-Means* and *DBSCAN* perform best when it comes to creating clustering on '*base*', '*gate*' and '*base gate*'. Now, we want to examine how the clusterings of each algorithm behave based on different parameters and how the generated clusters differ from each other.

K-Means For *K-Means*, we only have the parameter k we can change. k determines how many clusters *K-Means* creates. To evaluate how an increase of k changes the clustering, we calculate the *Normalized Mutual Information* (*NMI*), introduced in section 3.1.4. The *NMI* for the clustering with a parameter value of k is calculated in comparison to the clustering with a parameter value of $k - 1$.

In Figures 4.8, 4.9 and 4.10 the *NMI* and *CBS* for the combinations '*base*', '*gate*' and '*base gate*' is shown. We can see that the *NMI* behaves similar on all combinations and is over 80% for $k > 15$ for almost all clusterings. This indicates that increasing k even more only splits off small parts of other clusters, leaving the distribution of the instances to the clusters relatively unchanged. As can be expected, the *CBS* score decreases in most cases with the increase of k. This also suggests that in most cases, an existing cluster gets split in two smaller clusters, either finding better solvers on the new clusters or staying the same. Still, small increases of the *CBS* can occur, when the algorithm moves instances in clusters that decide on solvers, that perform worse on them. However, it is observable that a further increase of k, while leading to better *CBS* scores, would cause an approximation of the *VBS* where each of the instances has its own cluster. The values of the *NMI* suggest, that further increases only lead to small changes in the clusters for each increase, until a *VBS* is reached. We can therefor conclude that a clustering with a $k > 15$ should give a good representation on how *K-Means* clusters the instances.

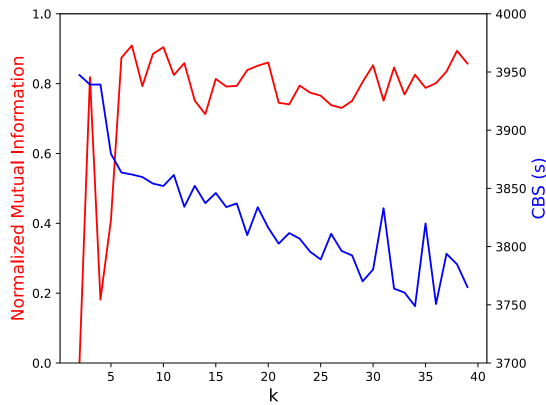

Figure 4.8: *NMI* and *CBS* for *K-Means* clustering using '*base*'

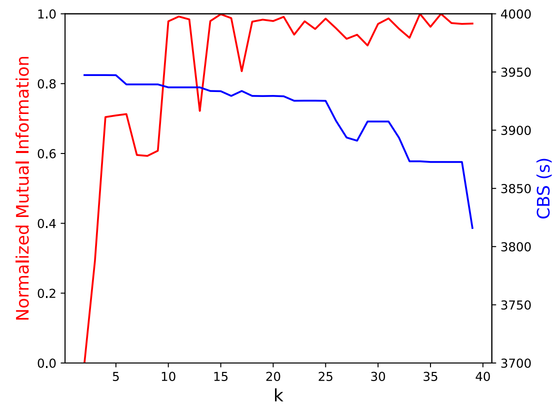

Figure 4.9: *NMI* and *CBS* for *K-Means* clustering using '*gate*'

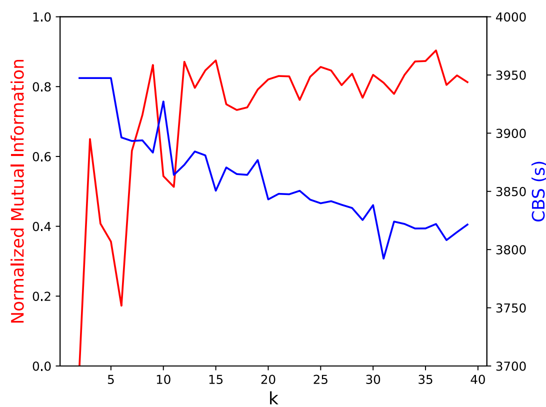

Figure 4.10: *NMI* and *CBS* for *K-Means* clustering using '*base gate*'

DBSCAN For *DBSCAN* we can change the epsilon value `eps` and the minimum samples `min_sample_size`. When exploring, we both iterated using `eps` and `min_sample_size` value while keeping the other one fixed. Inside our chosen range of values, the selection of either parameter to keep fixed or iterate do not seem to change the resulting *NMI*. We will therefore show the result for `min_sample_size`= 5 while iterating over `eps`. As expected with increasing `eps` values the *CBS* score increases, because we get bigger clusters, for which the *CSBS* is often worse, than for two separate clusters with the same instances. The *NMI* is relatively constant for all `eps` values. The only exception is a drop at `eps`= 2.2 for '*gate*'. We will discuss this observation next using reachability plots for each combination of feature sets.

Using *OPTICS*, we generated a reachability plot using `min_sample_size`= 5 for the combinations '*base*', '*gate*' and '*base gate*'. In Figures 4.14, 4.15 and 4.16 the reachability plots for each combination can be seen. It is clear that picking a value of `eps` that would result in bigger, tightly-packed clusters is hard. There are many "valleys" suggesting we can either create a few big cluster or many very small clusters. Selecting the option of small clusters, the `eps` value must lie below 1 for '*base*', '*gate*' and '*base gate*'. However, a problem with these values is that, this will also create many clusters with very few instances. Bigger `eps` values cause the formation of a few very big clusters. This also is not favorable as very big clusters will most likely have the *SBS* as their *CSBS*, resulting in no improvement of performance when clustering.
The sharp drop for '*gate*' when choosing `eps`= 2.2 becomes apparent when looking at its reachability plot (Figure 4.15). We can observe, that changing `eps` form 2.1 to 2.2 causes two big clusters to be combined into one, causing a huge drop in the *NMI*. In comparison to this, the reachability plots of '*base*' (Figure 4.14) and '*base gate*' (Figure 4.16) do not have as strong "spikes" as '*gate*', causing them to only have small changes in the *NMI* when changing `eps`.

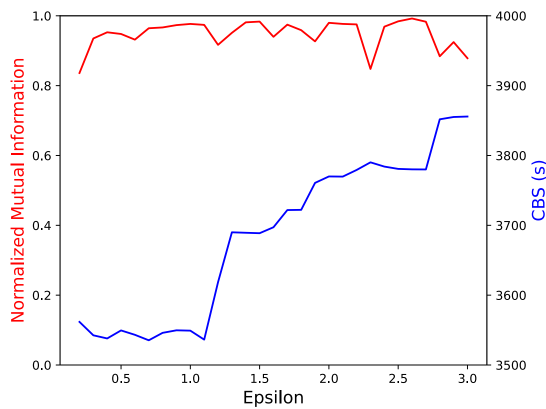

Figure 4.11: *NMI* and *CBS* for *DBSCAN* clustering using '*base*'

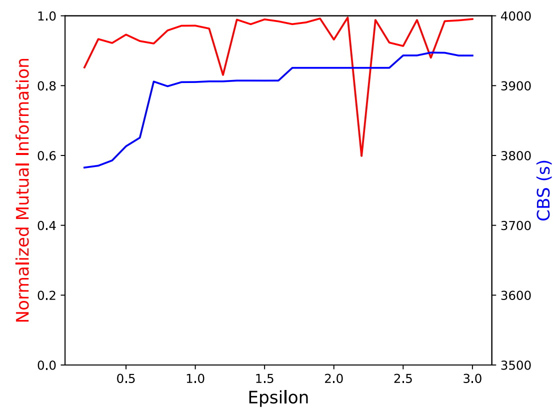

Figure 4.12: *NMI* and *CBS* for *DBSCAN* clustering using '*gate*'

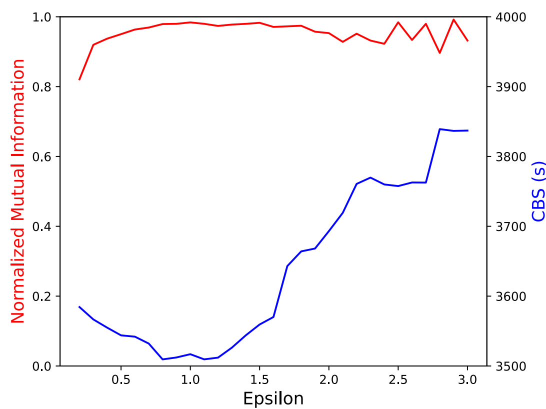

Figure 4.13: *NMI* and *CBS* for *DBSCAN* clustering using '*base gate*'

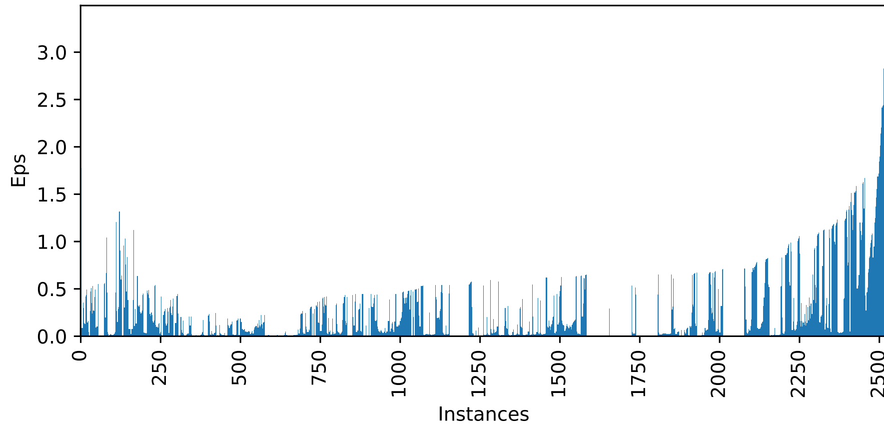

Figure 4.14: Reachability plot of '*base*'

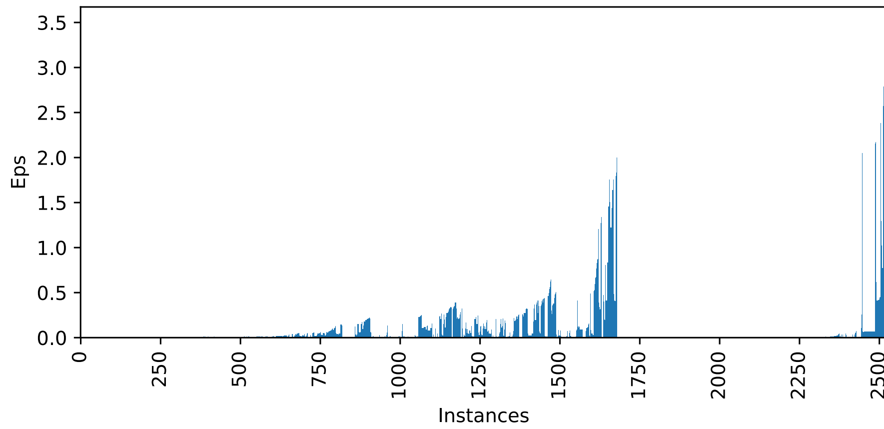

Figure 4.15: Reachability plot of '*gate*'

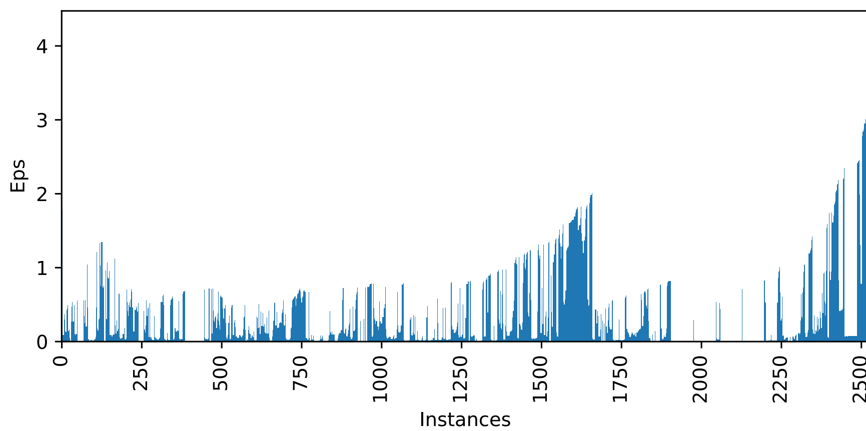

Figure 4.16: Reachability plot of '*base gate*'

4.2.3 Evaluation of selected clusterings

In the previous section, we saw, that the variation of the parameters of *K-Means* and *DB-SCAN* only cause small changes between clustering with similar parameter values. To get a deeper understanding on how these clusterings are structured, we will select a clustering for each algorithm to evaluate. For these clusterings, we will evaluate interesting clusters.

Filtering of clusters for selected clusterings For the following clusterings we will filter out selected clusters from the figures. Clusters will be included if the size of the cluster is bigger than the 0.25-Quantile of all cluster sizes of the selected clustering.

Clustering notation Next, we sorted the clusters in the selected clustering using the size of each cluster. This gives us a total ordering of all clusters. We can therefore assign each cluster an ID, beginning with 0 for the cluster with the most instances. We will use these IDs to uniquely identify each cluster in the following plots and tables. For *DBSCAN* the cluster with the ID 0 is the cluster that contains all outliers.

K-Means

Selection of the clustering When selecting a clustering for K-Means, we want to include multiple criteria:

(i) The clustering should use either '*base*', '*gate*' or '*base gate*'. We realized early on, that only using '*runtimes*' for clustering yields the best results, but does not help us when identifying common properties of the instances.

(ii) The clustering should have a low *CBS* score. We are interested in clusterings that group instances with similar fast solvers into the same cluster, therefore the *CBS* score should be low.

(iii) If possible, the k-parameter of *K-Means* should be similar to the one we estimated in section 3.2.4.

We have 117 possible clusterings that use *K-Means* and combinations of '*base*', '*gate*' or '*base gate*'. We selected the clustering that had the lowest *CBS* score. The settings of the selected clustering can be seen in Table 4.17.

Parameter	Value
Clustering Algorithm	K-Means
Feature sets	'*base*'
Number of clusters	34

Figure 4.17: Parameter values for the selected *K-Means* clustering

Clustering characteristics We calculated the scores for the complete clustering, without omitting any clusters. The 'Best' column shows the best value for the 117 clusterings using *K-Means* clustering and '*base*', '*gate*' or '*base gate*'. The *NMI* describes the *normalized mutual information* between the clustering and the clustering induced by the families of the instances. The score of the clustering can be seen in Table 4.18.

Score	Value	Best
CBS	3748.85s	3748.85s
CBSS	3676.23s	3667.59s
NMI with family	0.663	0.663
0.25-Quantile of cluster sizes	5	-

Figure 4.18: Scores of the selected *K-means* clustering

As we can see, the selected clustering performs best in the *CBS* and *NMI* score. It is second in the *CBSS* score for the 117 clusterings. Furthermore, it has 34 clusters, which is close to the estimated number in section 3.2.4. In section 4.2.2 we observed, that small changes in k does not cause big changes in the clusterings. Therefore, the selected clustering should be representative of *K-Means* clustering.
As stated previously, we will omit all clusters, that have sizes lower than the 0.25-Quantile. Therefore, clusters that do not have sizes larger than 5 are omitted, leaving 23 clusters. The following evaluation will only include these filtered clusters.

Clustering evaluation

Cluster sizes and families: In Figure 4.19, we can see the size of each cluster, as well as the shares of the families with more than 20% in the cluster. We can observe, that the distribution of cluster sizes is not uniform. Instead, the cluster sizes vary greatly. *K-Means* often sorts instances with the same family into the same clusters. This is especially the case for small clusters. Clusters with more than 100 instances are not as homogeneous when it comes to the families in them. In total, 7 of the 23 clusters (30.4%) contain only one family.
In Figure 4.21, we can see the distribution of the biggest families (0.75-Quantile of families by size) between multiple clusters. Please note that the same color, between two families, does not represent the same cluster. Instead, we annotated the cluster ID on each bar. We scaled all distributions to $[0, 1]$ because of different family sizes.
We can observe that some families are completely part of a single clusters. However, *K-Means* splits most families between multiple clusters. For the biggest families, most family instances are in a maximum of three clusters.
An overview of the distributions of families between clusters, with no filtering of families

and clusters, can be found in section A.4.

CSBS performance: In Figure 4.20, we can see the *SBS* and *CSBS* for each cluster. The solver annotated in bold for each bar is the *CSBS*. The other solvers are the solvers in the strip, sorted from best to worse by their *Par2* score. If there were too many solvers in the strip, we only showed *CSBS* and the other 5 best solvers in the strip. More solvers can be seen in Table 4.1. The annotation says **unsolvable** if there was no solver that could solve at least one instance in the cluster before the timeout.
We observe, in Figure 4.20, that 14 of 23 clusters (61%) use the *SBS kissat* as their *CSBS*. The clusters that use the *SBS kissat* contain 2014 of the total 2524 instances (79.79%). Because most instances in the clustering use the *SBS*, the overall *CBS* scores of the clusterings are always close to the *SBS*. 7 of the 23 clusters (30.4%) use *CSBS*s different from the *SBS*. All 7 clusters have a speedup of more than 100s when using the *CSBS* compared to the *SBS*. The highest absolute speedup can be observed in cluster 20 of approximately 2000s using *glucose_chanseok*. However, this has relatively low significance for the overall *CBS* score of the clustering, because cluster 20 only contains 9 of the 2524 instances. The remaining 2 of the 23 clusters (8.7%) contain only unsolvable instances.

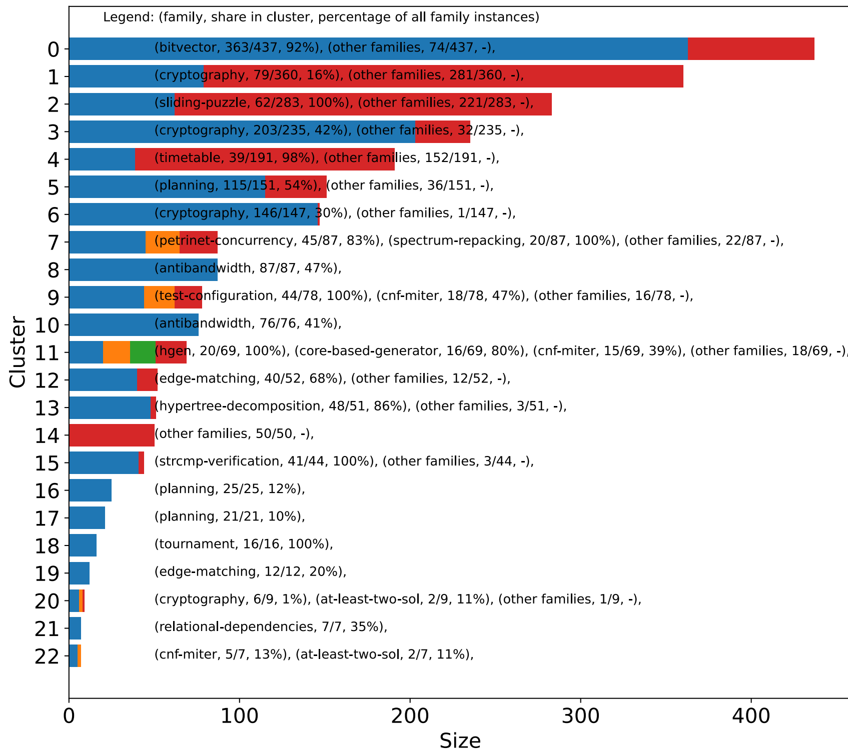

Figure 4.19: Size of each cluster showing, the shares of the biggest families

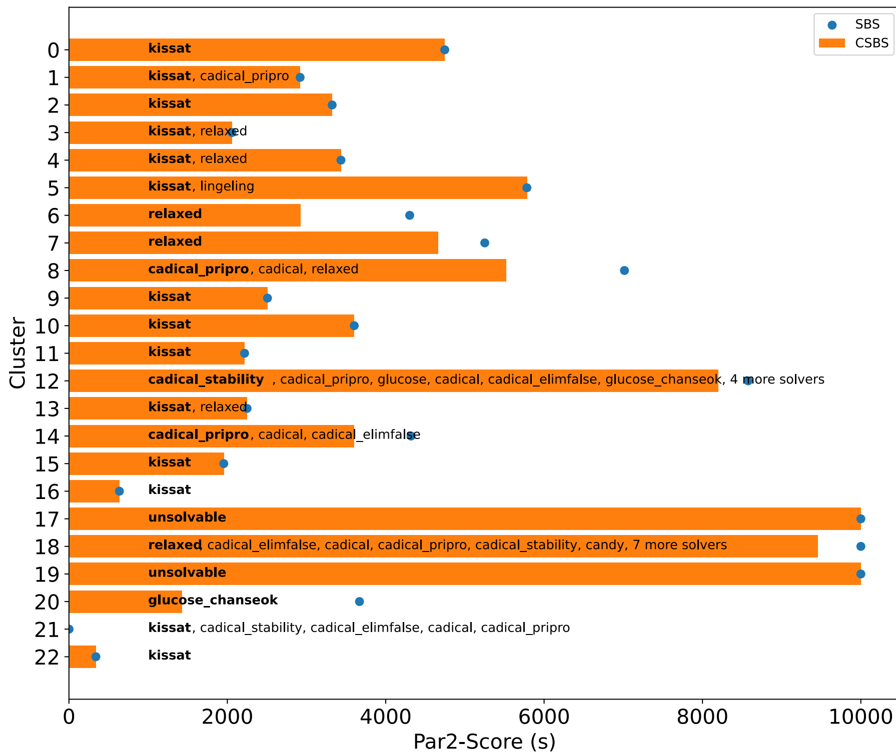

Figure 4.20: *SBS* and *CSBS* for each cluster

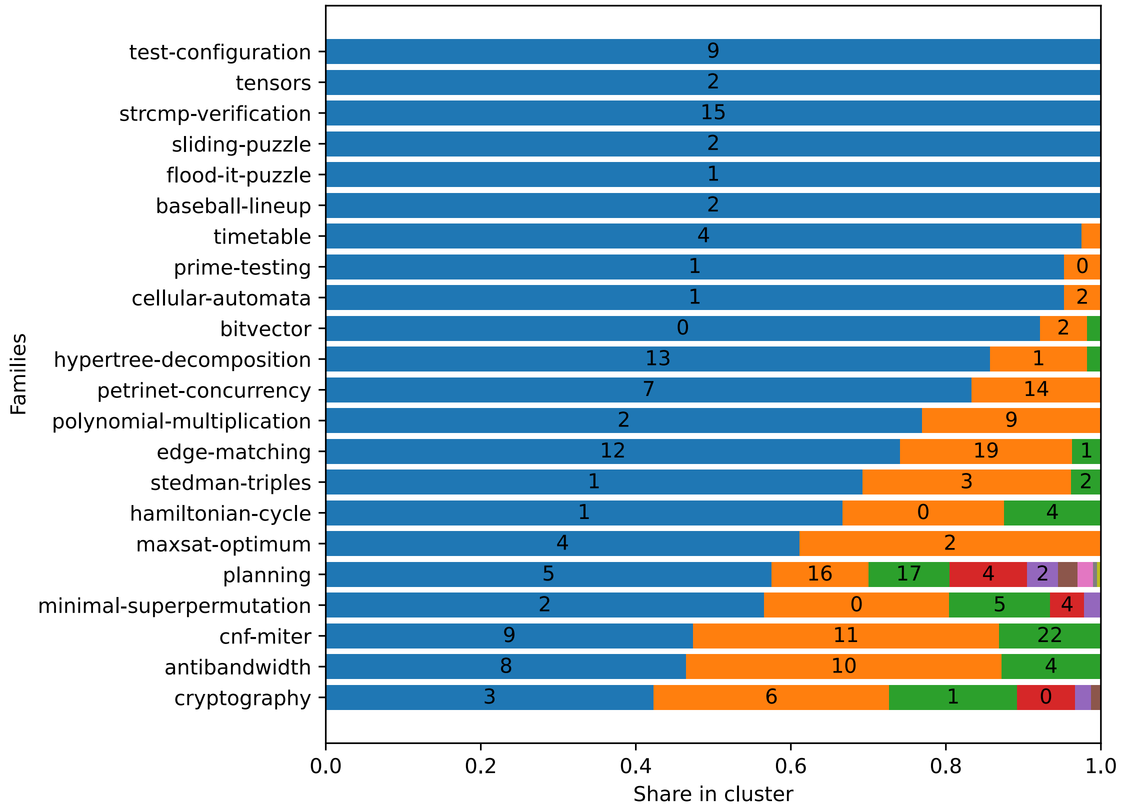

Figure 4.21: Distribution of the biggest families between the clusters

CSBS strip: While *kissat* might be the best *CSBS* for most clusters, we want to examine, if a cluster has other solvers with similar performance as the *CSBS*. To do this, we defined the strip of solvers with similar performance in section 3.1.4. The solver in the strip with similar performance for each cluster can be seen in Figure 4.20 and Table 4.1. We sorted the solvers in each strip from lowest to highest by their *Par2* score on the instances in the cluster.
We can observe that for most clusters, there is only one solver with high performance. The exceptions are the clusters with very low and very high *CSBS* scores. For the clusters 21 with low *CSBS* scores, there are multiple fast solvers in the strip. This is most likely the case because the instances in the cluster are easy and can therefore be solved by multiple solvers quickly as well as being a cluster with only 7 instances. The other extreme are the clusters with very high *CSBS* scores. Clusters 12 and 18 have *CSBS* scores over 8000s and both have at least 8 solvers in the strip. *kissat* occurs in 16 of the 23 (69.57%) of all strips.

Cluster	Strip
0	kissat
1	kissat, cadical-pripro
2	kissat
3	kissat, relaxed
4	kissat, relaxed
5	kissat, lingeling
6	relaxed
7	relaxed
8	cadical-pripro, cadical, relaxed
9	kissat
10	kissat
11	kissat
12	cadical-stability, cadical-pripro, glucose, cadical, cadical-elimfalse, glucose-chanseok, relaxed, kissat, lingeling, candy
13	kissat, relaxed
14	cadical-pripro, cadical, cadical-elimfalse
15	kissat
16	kissat
17	unsolvable
18	relaxed, cadical-elimfalse, cadical, cadical-pripro, cadical-stability, candy, glucose-chanseok, glucose, glucose-var-decay099, kissat, lingeling, march-nh, minisat
19	unsolvable
20	glucose-chanseok
21	kissat, cadical-stability, cadical-elimfalse, cadical, cadical-pripro
22	kissat

Table 4.1: Solvers in strip for each cluster

Stability of solvers: So far, we only used the *CSBS* score to evaluate the clusters of the clustering. Next, we want to compare the *CSBS* of each solver with the *CSBSS*. In Table 4.2, we can see the clusters where *CSBS* and *CSBSS* differ. If the *CSBS* and *CSBSS* are identical, we omit the cluster from the table. We can observe, that only two clusters (13 and 15) have different solvers for *CSBS* and *CSBSS*.
We can see the distribution of the runtimes for the *CSBS* and *CSBSS* for both clusters in Figures 4.22 and 4.23. For cluster 13 the overall distribution of the runtimes is much smaller for the *CSBSS* than the *CSBS*. Furthermore, the *CSBS* and *CSBSS Par2* scores are very close. Therefore, the *CSBSS* is in the strip of the *CSBS* as can be seen in Table 4.1. For cluster 15 the *Par2* scores of *CSBS* and *CSBSS* are not as close. For both solvers, the *SPar2* score is very similar, because the runtime distributions of both solvers are similar.
Furthermore, clusters 13 and 15 are interesting, because they both have a high overlap with a family. As can be seen in section 4.19, cluster 13 mostly contains *hypertree-decomposition* instance and cluster 15 contains all *strcmp-verification* instances.

Cluster	CSBS	CSBSS
13	kissat	relaxed
15	kissat	cadical-stability

Table 4.2: *CSBS* and *CSBSS* for each cluster

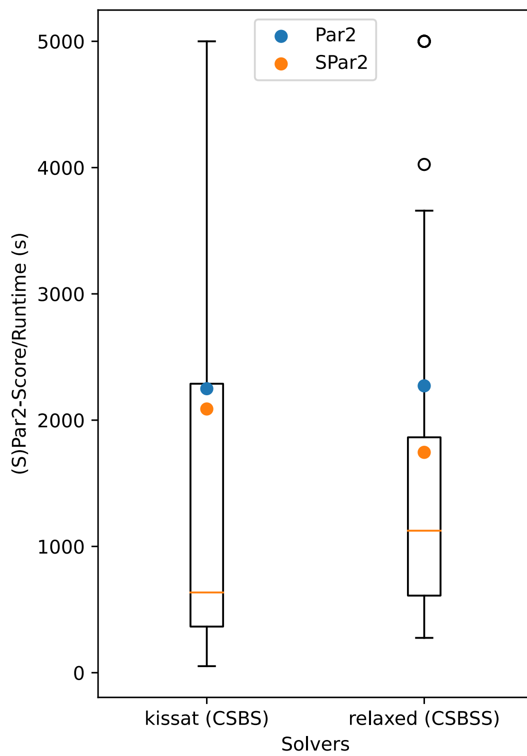

Figure 4.22: The distribution of *CSBS* and *CSBSS* runtimes with the *Par2* and *SPar2* scores for Cluster 13

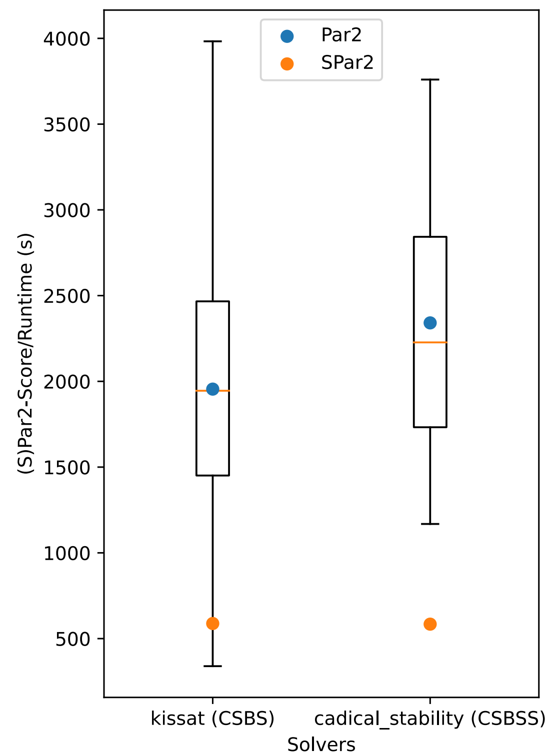

Figure 4.23: The distribution of *CSBS* and *CSBSS* runtimes with the *Par2* and *SPar2* scores for Cluster 15

DBSCAN

Selection of the clustering When selecting a cluster for *DBSCAN*, we want to include similar criteria to *K-Means*:

(i) The clustering should use either '*base*', '*gate*' or '*base gate*' for clustering. We realized early on, that only using '*runtimes*' for clustering yields the best results, but does not help us when identifying common properties of the instances.

(ii) The clustering should have a low *CBS* score. Because we are interested in clusterings that group the instances in clusters with similar fast solvers, the *CBS* score should be low.

(iii) The number of clusters should be similar to the one we estimated in section 3.2.4. While this estimation is better for *K-Means* the evaluation in section 4.2.2 shows that it is hard to estimate a good value for eps.

We have 810 possible clusterings that use *DBSCAN* and combinations of '*base*', '*gate*' or '*base gate*'. We then selected a clustering that had low scores for *CBS*, *CBSS* and *NMI* score. The selected clustering can be seen in Table 4.24.

Parameter	Value
Clustering Algorithm	DBSCAN
Feature sets	'*base gate*'
Epsilon	2
Min sample size	9
Number of clusters	37

Figure 4.24: Settings of the selected *DBSCAN* clustering

We saw in section 4.2.2 that changing the values of eps had no significant effect on the *NMI*. The reachability plot of '*base gate*' suggests that $eps = 2$ is a good value for a clustering. Furthermore, the number of clusters is close to the number we estimated in section 3.2.4. The chosen clustering should therefore be representative of the section *DBSCAN* clusterings.

Clustering characteristics We calculated the scores for the complete clustering, without omitting any clusters. The 'Best' column shows the best value for the 810 clusterings. Please note, that the missing limit of number of clusters when running DBSCAN causes many clusterings with more than 1000 clusters, approximating the *VBS*. Therefore, there is a bigger difference in the 'Best' column to the values of the selected clustering. The *NMI* describes the *normalized mutual information* between the clustering and the clustering that is induced by the families of the instances. The scores of the clustering can be seen in Table 4.25.

Score	Value	Best for all 810 clusterings
CBS	3725.20s	3140.44s
CBSS	3652.67s	2812.84s
NMI with family	0.655	0.804
0.25-Quantile of cluster sizes	16	-

Figure 4.25: Scores of the selected *DBSCAN* clustering

As stated previously, we will omit all clusters, that do not have sizes larger than the 0.25-Quantile. This omits all clusters not larger than 16, leaving 27 clusters (including the cluster containing the outliers). The following will only include these filtered clusters.

Clustering evaluation

Cluster sizes and families: In Figure 4.26, we can see the size of each cluster, as well as the shares of the families with more than 20% in the cluster. While *DBSCAN* does not have a uniform distribution, its clusters are not as different in size when compared to the clustering of *K-Means* (excluding the cluster 0 of outliers). Similar to *K-Means*, *DBSCAN* often sorts instances with the same family into the same cluster. 15 of 27 clusters (55.56%) only contain one family. This is an increase in comparison to the 30.4% of *K-Means*. Instead, many family instances get sorted into the outlier cluster 0, leaving more homogeneous clusters.
The distribution of the biggest families in Figure 4.28 is similar to *K-Means*. Most families get split between a maximum of three big clusters. For some families the split between clusters is similar for *DBSCAN* and *KMEANS* e.g., *cryptography*, *planning*, *tensor* are split similarly for both clusterings. However, that is not true for all families, e.g., *antibandwidth* has almost all instances in one cluster using *DBSCAN*, while it is split between two big clusters and one small cluster when using *K-Means*. An overview of the distributions of families between clusters without filtering families and clusters can be found in section A.4.

CSBS performance: Figure 4.27 shows the *SBS* and *CSBS* for each cluster. The solver annotated in bold for each bar is the *CSBS*. The other solvers are the solvers in the strip, sorted from best to worse by their *Par2* score. If there were too many solvers in the strip, we only showed the 5 best solver in the strip. The other solvers can be seen in section 4.3. If the annotation says **unsolvable**, there was no solver that could solve at least one instance in the cluster before the timeout.
15 of 27 clusters (55.56%) use the *SBS* as their *CSBS*. The clusters that use *kissat* contain 1732 of the total 2524 instances (68.62%). This is an improvement in comparison to the clustering of *K-Means*. 11 of the 27 clusters (40.74%) use *CSBS*s different from the *SBS*. 10 of these 11 clusters have a performance increase of more than 100s when using

the *CSBS* instead of the *SBS*. The biggest absolute performance increase shows cluster 21, using *march_nh* as its *CSBS*. However, cluster 21 only contains 23 instances. The cluster 20 is unsolvable.

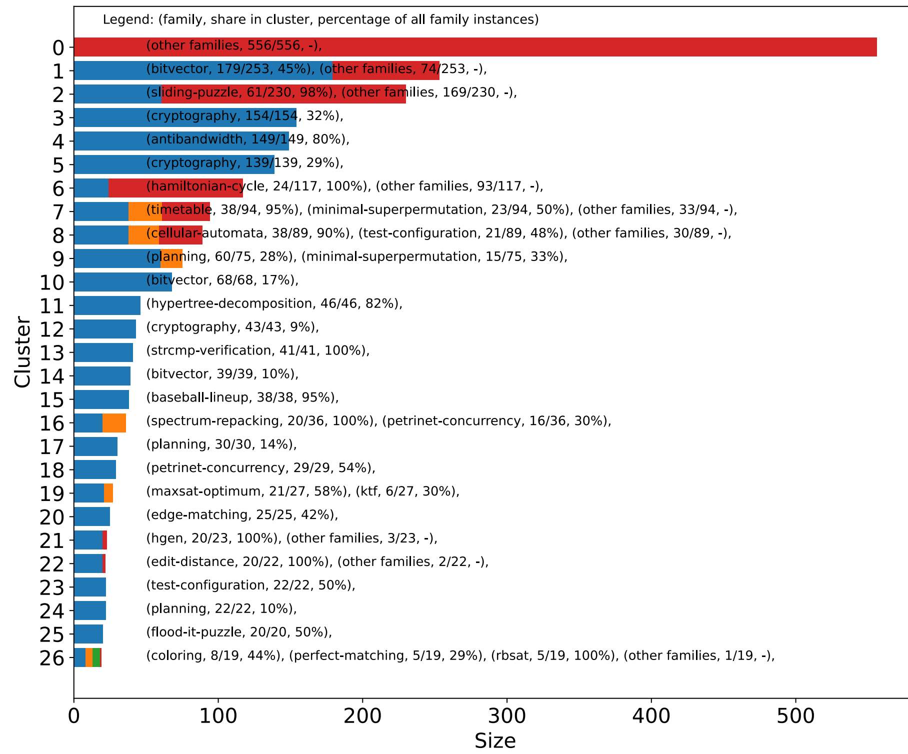

Figure 4.26: Size of each cluster showing, the shares of the biggest families

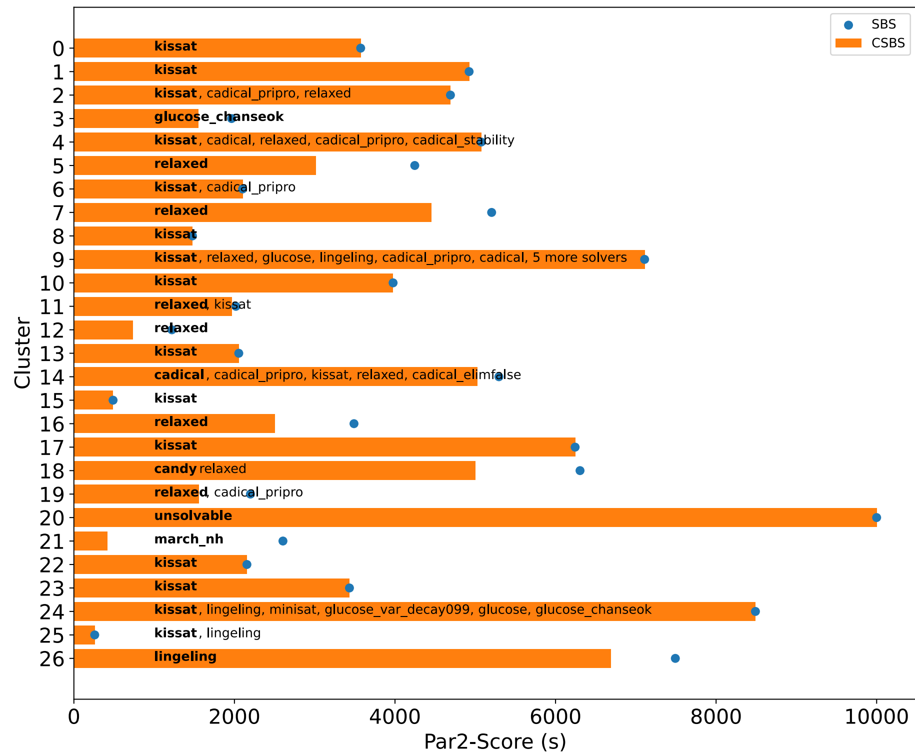

Figure 4.27: *SBS* and *CSBS* for each cluster

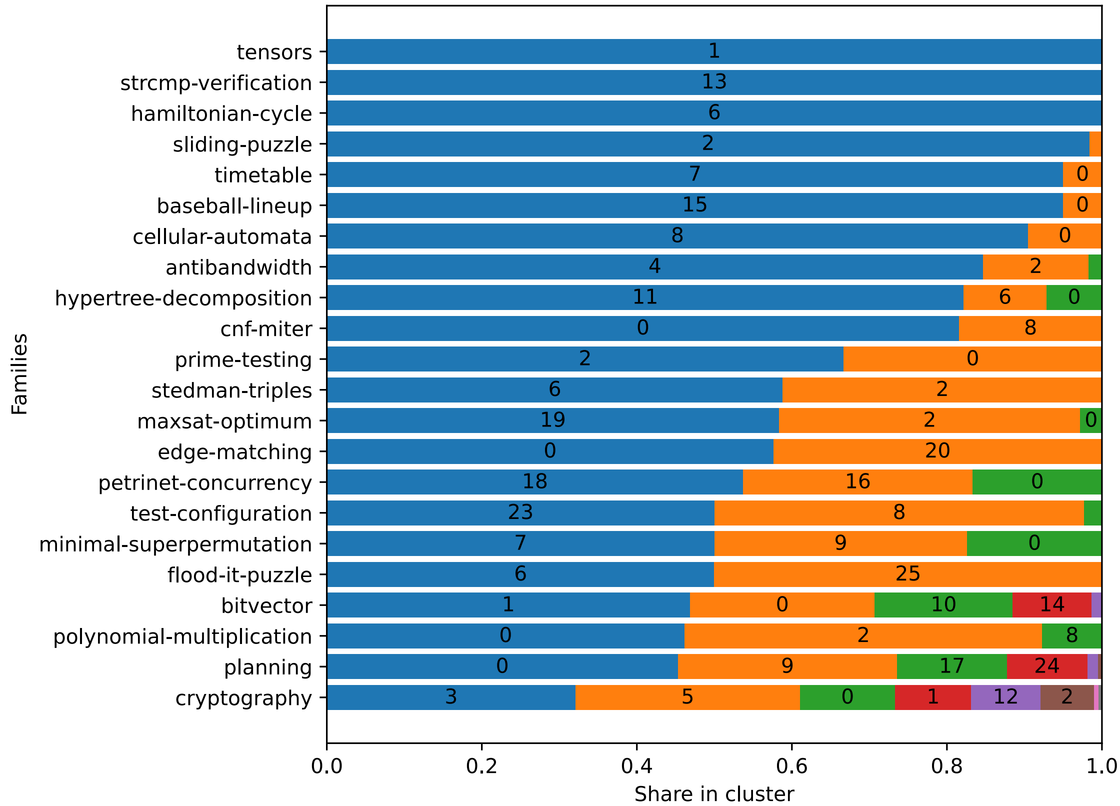

Figure 4.28: Distribution of the biggest families between the clusters

CSBS strip: While *kissat* might be the best *CSBS* there might be other solvers with similar performance. The solver in the previously defined strip can be seen in the Table 4.3 and Figure 4.27. We can see that the number of solvers for most clusters is again 1. The number of solvers increases with the increase of the *CSBS* score of each cluster. Especially, clusters using *kissat* as their *CSBS* often do not have a solver with similar performance. *kissat* occurs in 17 of 27 (62.96%) of strips.

Cluster	Strip
0	kissat
1	kissat
2	kissat, cadical-pripro, relaxed
3	glucose-chanseok
4	kissat, cadical, relaxed, cadical-pripro, cadical-stability
5	relaxed
6	kissat, cadical-pripro
7	relaxed
8	kissat
9	kissat, relaxed, glucose, lingeling, cadical-pripro, cadical, glucose-chanseok, cadical-stability, cadical-elimfalse, candy, glucose-var-decay099
10	kissat
11	relaxed, kissat
12	relaxed
13	kissat
14	cadical, cadical-pripro, kissat, relaxed, cadical-elimfalse
15	kissat
16	relaxed
17	kissat
18	candy, relaxed
19	relaxed, cadical-pripro
20	unsolvable
21	march-nh
22	kissat
23	kissat
24	kissat, lingeling, minisat, glucose-var-decay099, glucose, glucose-chanseok
25	kissat, lingeling
26	lingeling

Table 4.3: Solvers in strip for each solver

Stability of solvers: Next, we want to compare the *CSBS* and *CSBSS* of each cluster. Table 4.3 shows the clusters that have different solvers for *CSBS* and *CSBSS*. If the solvers are identical, we do not list the cluster. Only one cluster has different solvers. We can see the distribution of the runtime features of the *CSBS* and *CSBSS* of this cluster in Figure 4.29. The *CSBSS* is in general slower in the instances than the *CSBS*. Therefore, its *Par2* score is worse. However, the runtime distribution of the *CSBSS* is slightly smaller than the one of the *CSBS*, therefore having a higher stability. Because the *CSBS* and *CSBSS* have a similar *Par2* score, the *CSBSS* appears in the strip of the *CSBS* in Table 4.3.

Cluster	CSBS	CSBSS
4	kissat	relaxed

Table 4.4: *CSBS* and *CSBSS* for each cluster

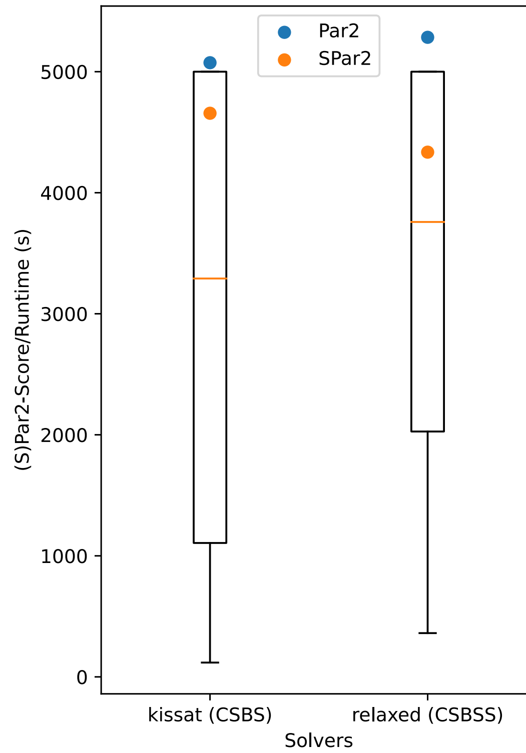

Figure 4.29: The distribution of *CSBS* and *CSBSS* runtimes with the *Par2* and *SPar2* scores for Cluster 4

5 Discussion

5.1 Conclusion

From our evaluations, we can take away multiple conclusions, which we want to reflect using the research questions stated at the beginning:

How do different sets of instance features influence the clustering quality of different clustering algorithms? When using the different combinations of '*base*', '*gate*' and '*runtimes*', '*runtimes*' generates the clusterings with the best *CBS* scores. Combinations of '*base*' and '*gate*' perform, *CBS* score wise, much worse than '*runtimes*'. Combining '*runtimes*' with '*base*' and '*gate*' also yields no improvement in the clusterings.

What clustering algorithms and parameter settings yield clusterings with high quality? When using only '*base*', '*gate*' or '*base gate*' for clustering the two best performing clustering algorithms, *CBS* score wise, are *K-Means* and *DBSCAN*. Closer observation of the behavior under different parameter values suggest, that for *K-Means* the optimal setting for the parameter k lies between 15 and 35. For *DBSCAN* the parameter settings are not as clear. The reachability plot suggests, that there are no clear clusters for *DBSCAN* in the dataset. Depending on our selection of eps, we will either create many small clusters or a few big clusters.

How are the instances in a clustering split into clusters? When looking at selected clusterings of *DBSCAN* and *K-Means* we noticed, that many clusters contain a majority of a single family. This is especially the case for *DBSCAN*, which moves instances, it can not sort into any cluster, into an outlier cluster. However, *K-Means* and *DBSCAN* often split families between multiple clusters. The instances are not in a uniform distribution of the clusters. Especially *K-Means* has very big differences between the sizes of the clusters.

Do instances in a cluster share solvers, which shows homogeneous runtime behavior? When examining selected clusterings under the aspect of *CSBS* we noticed that many clusters use the original *SBS kissat* as their *CSBS*. For the selected clustering, more than 30% of clusters show significant performance increases using the *CSBS* in comparison to the *SBS*. However, these are often relatively small clusters. This explains why

the *CBS* only has a small performance increase in comparison to the *SBS*.
The strips of the clusters show, that 37 of the 50 clusters (87%), from the two selected clusterings, only have one or two solvers that show low *CSBS* scores. Furthermore, the *CSBS* and *CSBSS* are the same solver for most clusters. This suggests that for all other clusters the solver with the fastest runtime behavior and the solver with the most homogeneous runtime behavior, on all instance in the cluster, are the same solver.

5.2 Future Work

In our different evaluations there are multiple interesting aspects not discussed here in-depth but which might be interesting for further analysis.

Scaling We found that both the linear scaling and standard scaling both had outliers with low *CBS* scores. However, the distribution of the *CBS* scores of both scaling techniques are different. A further analysis of linear scaling in combination with clustering algorithm might therefore be able to further support the results of this thesis.

Feature selection In our approach, we focused on clustering different combinations of the feature sets '*base*', '*gate*' and '*runtimes*'. However, in section 4.1.2 we noticed that both '*base*' and '*gate*' have subsets of features that create better *CBS* scores than other features. A clustering based on these subsets of features might therefore be interesting. Especially the *n_roots* feature of '*gate*' is a significant outlier in comparison to the other '*gate*' features.

A Implementation Details

A.1 Project organization

The project used to generate the data of this paper can be found at https://github.com/SimuIacron/cluster_analysis_for_sat_instances. The project is written in python 3.8. All dependencies are in the requirements.txt.
The folder structure of the project is as follows:

```
cluster_analysis_for_sat_instances
├─ DataAnalysis
│  └─ Scoring
├─ DataFormats
├─ PlottingAndEvaluationFunctions
├─ PlottingTex
│  ├─ ComparingClusterAlgorithms
│  │  ├─ AnalysisOfGoodClusterAlgorithms
│  │  └─ SelectedSingleClusters
│  └─ Preprocessing
└─ UtilScripts
```

The two main scripts for the clustering (run_experiments.py) and evaluation (run_evaluation.py) are placed in cluster_analysis_for_sat_instances.
run_experiments.py contains the identically named function run_experiements which takes a set of clustering experiments to run and returns the clusterings as a text file. An example of a call of the function can be found at the bottom of the file.
run_evaluation.py contains the identically named function run_evaluation. For a text file of clusterings generated by run_experiements, it calculates the *CBS* as well as *CSBS* for the clusterings and saves them as a text file.
Both run_experiments.py and run_evaluation.py use functions supplied by scripts in DataAnalysis. DataAnalysis contains scripts for scaling, feature selec-

tion, clustering and scoring of clusterings.
The folder `PlottingTex` contains all scripts used to generate the plots in this thesis. The folders are named similarly to the sections the plots appear in. Each script reads in the text files generated by `run_evaluation` and `run_experiements`.
To generate the plots, the scripts in `PlottingTex` use plotting and evaluation functions supplied by `PlottingAndEvaluationFunctions`. This folder contains functions to both evaluate and plot clusterings. To differentiate, all files generating plots for the thesis have the suffix `plot_` and all files supplying functions for plotting have the suffix `func_plot_`.
`DataFormats` contains the abstractions `DBInstance.py` from the Database and `DatabaseReader.py` supplies methods for `DbInstance` to read out the gbd-Database. The `DbInstance` is used as an abstraction of the database in many scripts to read data about specific instances.

A.2 Default settings tables

The tables show the default values used in experiments, if not specified differently in the experiment section. Step size describes how big the steps were that were done in the given interval, e.g., if the interval is [1,10] and the step size is 1 the default values are {1,2,3,4,5,6,7,8,9,10}.

A.2.1 Scaling and feature selection

General Settings			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Scaling Algorithm	scaling_algorithm	Standard Scaling (4.1.1)	-
Feature Selection Algorithm	selection_algorithm	None	-

A.2.2 Clustering

K-Means			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Seed	seed	0	-
Number of Clusters	n_cluster_k_means	[1,39]	1

Affinity Propagation			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Seed	seed	0	-
Damping	damping_aff	[0.5, 0.9]	0.1
Preference	preference_aff	None	-
Affinity	affinity_aff	euclidean	-

Meanshift			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Bandwidth	badwidth_mean	[1, 9], None	1

Spectral Clustering			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Seed	seed	0	-
Number of Clusters	n_clusters_spectral	[2, 3]	1

Agglomerative Clustering			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Number of Clusters	n_clusters_agg	[1, 39]	1
Affinity	affinity_agg	euclidean	-
Linkage	linkage_agg	[ward, complete, average, single]	-
Distance Threshold	distance_threshold	None	-

Optics			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Minimum Samples	min_samples_opt	[1, 9]	1
Minimum number of clusters	min_cluster_opt	[1, 9], None	1

Gaussian			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Seed	seed	0	-
Number of Components	n_components_gauss	[1, 9], None	1

DBSCAN			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Eps	eps_dbscan	[0.1, 0.9]	0.1
Minimum Samples	min_samples_dbscan	[1,19], None	1

BIRCH			
Parameter	Variable Name of Parameter	Interval/Value	Step Size
Threshold	threshold_birch	[0.1, 0.9]	0.1
Branching factor	branching_factor_birch	[10, 99], None	10
Number of clusters	n_clusters_birch	[1, 39], None	1

A.3 Feature sets

This is a list of the features of each feature set. The dataset can be found at `https://git.scc.kit.edu/fv2117/gbd-data`. For more information about the feature extraction of '*base*' and '*gate*', see `https://github.com/sat-clique/cnftools`.

A.3.1 '*base*'

clauses, variables, clause_size_1, clause_size_2, clause_size_3, clause_size_4, clause_size_5, clause_size_6, clause_size_7, clause_size_8, clause_size_9, horn_clauses, inv_horn_clauses, positive_clauses, negative_clauses, horn_vars_mean, horn_vars_variance, horn_vars_min, horn_vars_max, horn_vars_entropy, inv_horn_vars_mean, inv_horn_vars_variance, inv_horn_vars_min, inv_horn_vars_max, inv_horn_vars_entropy, balance_clause_mean, balance_clause_variance, balance_clause_min, balance_clause_max, balance_clause_entropy, balance_vars_mean, balance_vars_variance, balance_vars_min, balance_vars_max, balance_vars_entropy, vcg_vdegrees_mean, vcg_vdegrees_variance, vcg_vdegrees_min, vcg_vdegrees_max, vcg_vdegrees_entropy, vcg_cdegrees_mean, vcg_cdegrees_variance, vcg_cdegrees_min,

vcg_cdegrees_max, vcg_cdegrees_entropy, vg_degrees_mean, vg_degrees_variance, vg_degrees_min, vg_degrees_max, vg_degrees_entropy, cg_degrees_mean, cg_degrees_variance, cg_degrees_min, cg_degrees_max, cg_degrees_entropy, base_features_runtime

A.3.2 '*gate*'

n_vars, n_gates, n_roots, n_none, n_generic, n_mono, n_and, n_or, n_triv, n_equiv, n_full, levels_mean, levels_variance, levels_min, levels_max, levels_entropy, levels_none_mean, levels_none_variance, levels_none_min, levels_none_max, levels_none_entropy, levels_generic_mean, levels_generic_variance, levels_generic_min, levels_generic_max, levels_generic_entropy, levels_mono_mean, levels_mono_variance, levels_mono_min, levels_mono_max, levels_mono_entropy, levels_and_mean, levels_and_variance, levels_and_min, levels_and_max, levels_and_entropy, levels_or_mean, levels_or_variance, levels_or_min, levels_or_max, levels_or_entropy, levels_triv_mean, levels_triv_variance, levels_triv_min, levels_triv_max, levels_triv_entropy, levels_equiv_mean, levels_equiv_variance, levels_equiv_min, levels_equiv_max, levels_equiv_entropy, levels_full_mean, levels_full_variance, levels_full_min, levels_full_max, levels_full_entropy, gate_features_runtime

A.3.3 '*runtimes*'

cadical_elimfalse, cadical, cadical_pripro, cadical_stability, candy, glucose_chanseok glucose, glucose_syrup, glucose_var_decay099, kissat, lingeling, march_nh, minisat relaxed, yalsat

A.4 Family distribution of selected clusters

This section contains figures showing the share of clusters in families and share of families in clusters for the selected clusterings in section 4.2.3 and 4.2.3. In these figures, all families and clusters of the selected clusterings are included. We sorted the clusters by size, therefor the cluster IDs of the figures in this section match the IDs in section 4.2.3 and 4.2.3.

A.4.1 K-Means

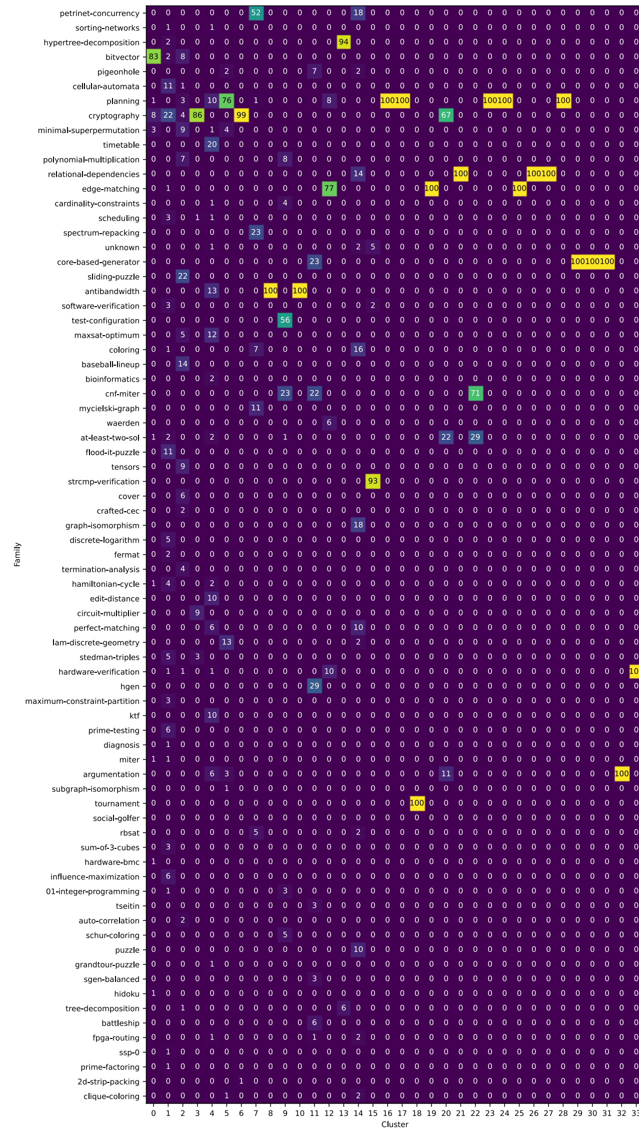

Figure A.1: Heatmap showing the share of each family in the cluster for the selected *KMEANS* clustering

Figure A.2: Heatmap showing the share of each cluster for a family for the selected *KMEANS* clustering

A.4.2 DBSCAN

Figure A.3: Heatmap showing the share of each family in the cluster for the selected *DBSCAN* clustering

Figure A.4: Heatmap showing the share of each cluster for a family for the selected *DBSCAN* clustering

Bibliography

Collautti, Marco et al. (Sept. 2013). *SNNAP: Solver-based Nearest Neighbor for Algorithm Portfolios*. Vol. 8190. Springer-Verlag Berlin Heidelberg. DOI: 10.1007/978-3-642-40994-3_28.

Fred, Ana and Arjun Jain (July 2003). "Robust data clustering". In: vol. 2, pp. II–128. ISBN: 0-7695-1900-8. DOI: 10.1109/CVPR.2003.1211462.

Heule, Marijn, Matti Jarvisalo, and Tomas Balyo (2017). *SAT Competition 2017*. URL: https://baldur.iti.kit.edu/sat-competition-2017/ (visited on 12/03/2021).

Heule, Marijn, Matti Jarvisalo, Martin Suda, et al. (2021). *SAT Competition 2021*. URL: https://satcompetition.github.io/2021/index.html (visited on 02/22/2022).

Heule, Marijn and Hans Maaren (Jan. 2009). "Look-ahead based SAT solvers". In: *Frontiers in Artificial Intelligence and Applications* 185. DOI: 10.3233/978-1-58603-929-5-155.

Hoos, Holger and Thomas Stützle (Jan. 2000). "Local Search Algorithms for SAT: An Empirical Evaluation". In: *Journal of Automated Reasoning* 24, pp. 421–481.

Iser, Markus (2022). *cnftools*. URL: https://github.com/sat-clique/cnftools (visited on 01/10/2022).

Iser, Markus, Luca Springer, and Carsten Sinz (Sept. 2020). *Collaborative Management of Benchmark Instances and their Attributes*.

Kadioglu, Serdar et al. (2010). *ISAC - Instance-Specific Algorithm Configuration*. The authors and IOS Press. DOI: 10.3233/978-1-60750-606-5-751.

Kerschke, Pascal et al. (2019). "Automated Algorithm Selection: Survey and Perspectives". In: *Evolutionary Computation* 27.1, pp. 3–45. DOI: 10.1162/evco_a_00242.

Lindauer, Marius et al. (2017). "Automatic construction of parallel portfolios via algorithm configuration". In: *Artificial Intelligence* 244. Combining Constraint Solving with Mining and Learning, pp. 272–290. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2016.05.004. URL: https://www.sciencedirect.com/science/article/pii/S0004370216300625.

Rice, John R. (1976). *The algorithm selection problem*. Ed. by Morris Rubinoff and Marshall C. Yovits. Vol. 15. Advances in Computers. Elsevier, pp. 65–118. DOI: https://doi.org/10.1016/S0065-2458(08)60520-3. URL: https://www.sciencedirect.com/science/article/pii/S0065245808605203.

scikit-learn (2021a). *2.3 Clustering*. URL: https://scikit-learn.org/stable/modules/clustering.html (visited on 10/21/2021).

– (2021b). *sklearn.preprocessing.StandardScaler*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html?highlight=standard#sklearn.preprocessing.StandardScaler (visited on 12/08/2021).

Silva, João, Inês Lynce, and Sharad Malik (Jan. 2009). "Conflict-Driven Clause Learning SAT Solvers". In: *Frontiers in Artificial Intelligence and Applications* 185. DOI: 10.3233/978-1-58603-929-5-131.

Strehl, Alexander and Joydeep Ghosh (2002). "Cluster Ensembles — A Knowledge Reuse Framework for Combining Multiple Partitions". In: *J. Mach. Learn. Res.* 3, pp. 583–617.

Wagner, Dorothea, Jonas Sauer, and Guido Brückner (2019). "Theoretische Grundlagen der Informatik: Vorlesung am 19.11.2019". In.

Wagner, Silke and Dorothea Wagner (Jan. 2007). "Comparing Clusterings - An Overview". In: *Technical Report 2006-04*.

Xu, Lin et al. (June 2008). "SATzilla: Portfolio-based Algorithm Selection for SAT". In: *Journal of Artificial Intelligence Research* 32, pp. 565–606. DOI: 10.1613/jair.2490.