Master Thesis

Increasing the Robustness of SAT Solving with Machine Learning Techniques

Enrique Matos Alfonso

10. September 2014

Technische Universität Dresden
Fakultät Informatik
Institut für Künstliche Intelligenz
Professur für Wissensverarbeitung

Betreut von:

Prof. Dr. rer. nat. Steffen Hölldobler

Dipl.-Inf. Norbert Manthey

Enrique Matos Alfonso
Increasing the Robustness of SAT Solving with Machine Learning Techniques
Master Thesis, Fakultät für Informatik
Technische Universität Dresden, September 2014

Task of the Master Thesis

Surname, Name:	Matos Alfonso, Enrique
Course of Studies:	European Master's Program in Computational Logic (EMCL)
Matrikelnummer:	3928013
Title:	***Increasing the Robustness of SAT Solving with Machine Learning Techniques***
Task Description:	The question whether a propositional formula in conjunctive normal form is satisfiable (SAT) is answered with powerful clause learning SAT solvers. However, the formulas originate from many different applications and scientific questions, and many different solving techniques have been proposed. Usually, the default configuration of a given SAT solver can handle formulas that are already known to the community. However, for novel formulas this configuration might fail. In this thesis machine learning techniques should be applied to improve this situation. Given a novel formula, then from a set of preselected SAT solver configurations a configuration should be picked that can solve the formula. For this task, existing CNF feature extraction methods should be used as a starting point. The extraction might be adopted to new machine learning techniques. While the overall performance should be improved for novel formulas, the performance on the overall benchmark should not decrease.

Betreuer:	Norbert Manthey
verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. Steffen Hölldobler
Institut:	Künstliche Intelligenz
Lehrstuhl:	Wissensverarbeitung
Beginn am:	21.04.2014
Einzureichen am:	10.09.2014

Abstract. Algorithm portfolios have become very popular in SAT competitions. The portfolio together with a good algorithm selection model can solve more instances than the best algorithm. For the algorithm selection task normally machine learning techniques are used based on features computed from the SAT instances. When the algorithm selection model is tested on instances that were part of the training dataset the results are very accurate but when novel instances are included in the testing dataset the performance decreases.

The primary purpose of this thesis was to study the performance on novel instances of the algorithm selection models based on machine learning techniques. A portfolio of different configurations of the Riss solver was built. New features were proposed and ten versions of features computation were tested. Furthermore, Six versions of machine learning models were proposed for the algorithm selection task. Four of them were based on binary classification models that predict when the configurations are "good" or "bad" for the given instance. The remaining two models used the *k*-nearest neighbor algorithm and a selection method based on the maximum value of a weighted contribution to predict the algorithm that was going to be used on the instance. The models were tested on novel instances and also on the complete benchmark.

The obtained results show that the proposed features can be computed efficiently, having 96 % of non redundant information. Only the models that were not based on binary classification could outperform the best configuration when tested on novel instances. It was observed that the best prediction models performed better than the best configuration and better than the Lingeling solver, when tested on the complete benchmark.

Contents

List of Figures

List of Tables

1 Introduction

The propositional satisfiability problem (SAT) is one of the most studied NP-complete problems. It has several applications in the areas of hardware verification [KSHK07], routing [ARMS02] and scheduling [HW10, GHM+12].

Starting in year 2002 until the present year, nine SAT competitions have being organized mainly to promote new solvers for the SAT problem and to compare them with state-of-the-art solvers. Since the incursion of SATzilla [XHHLb07] in 2007, in the SAT competitions the “algorithm portfolios” have become very popular in the competition [Rou12, KMS+11, XHHLb07] and also in literature [NMJ09, NMJ11, XHHLB12]. The algorithm portfolios combine different SAT algorithms in order to achieve a performance as close as possible to the *virtual best solver* (VBS), that solves a SAT instance as fast as the fastest of the algorithm in the portfolio. The considered algorithms can represent versions of the same solver with different configuration of parameters [AM14a] or they can also be completely different solvers [XHHLb07, NMJ09].

Most of the existing algorithm portfolios for SAT use *machine learning* techniques to select the algorithm that will take care of solving the given instance. *Regression* was used for SATzilla first version [XHHLb07] but nowadays *classification* and *instance based learning* are more commonly used to select an algorithm in the portfolio. Although the features proposed by SATzilla authors have been a popular choice, nevertheless other researchers have also proposed different features [Gab, AM14a].

In the SAT competitions the instances that belong to benchmarks of previous years can also be repeated on the current year and only about 50 % percent of the total of instances in the benchmark are truly novel instances. The performance of the algorithm portfolios can be highly influenced by the fact that most of the instances were already part of the training dataset and rather than predicting a configuration we are just remembering the configuration that worked with the repeated instances. In general, a good algorithm selector should perform well on previously seen instances and also on novel instances i.e. selecting an algorithm that can solve the instance within the given timeout.

This thesis explores some of the existing machine learning techniques in order to come up with a model that performs well on both novel instances and repeated instances. First, different versions of features computation are proposed and analyzed experimentally. Also different algorithm selection models based on machine learning techniques are proposed and an experimental study is performed in order to select a model that has the best empirical performance on novel instances and also on instances that were used for the training process of the model.

The features computed provided 96 % of non redundant information that can be used to represent CNF instances. The best prediction model obtained outperforms Lingeling solver [Bie13] and outperforms the best configuration of the Riss [Man14] solver in the novel instances and also when the model is tested on novel instances as well as on previously seen instances.

In the next section the definitions and notation used in this thesis are presented. Following in Section 3, the related work is reviewed. Section 4 presents the features extracted from SAT instances. Section 5 introduces the proposed machine learning models to be used for the algorithm selection using the extracted features. Later in Section 6 the evaluation results of the proposed models are shown. This thesis is finalized in Section 7, where the conclusions are presented as well as the future work.

2 Basics

The Boolean Satisfiability problem (SAT) is formally defined as the problem of deciding whether a propositional formula F is satisfiable or not. Usually it is also important to know the interpretation that satisfies the formula. Finding an efficient algorithm to solve the SAT problem is very important, because such algorithm would influence many other problems in the NP complete class, because they all have polynomial time reductions to SAT [Kar72].

2.1 CNF Formulas

Most SAT solvers require as input a *Conjunctive Normal Form* (CNF) formula. A variable $v_i \in \{v_1, \dots, v_m\}$ occurring in a CNF instance F is represented by a positive natural number $i \in \{1, \dots, m\}$, where m is the number of variables present in the instance. A literal is either a positive variable (v, positive polarity) or a negated variable ($-v$, negative polarity). A clause will be represented as a set of literals $C_i = \{l_1, \dots, l_n\}$. Sometimes, a clause will also be represented with squared brackets ($C_i = [l_1, \dots, l_n]$). Finally, a CNF formula will be represented by a set of clauses $F = \{C_1, \dots, C_n\}$ but sometimes is also written in the following way $F = \langle C_1, \dots, C_n \rangle$.

We define the complement operation over literals and clauses:

$$\bar{l} = \begin{cases} -v & \text{if } l = v \\ v & \text{if } l = -v \end{cases}$$

$$\overline{C} = \{\bar{l} |\ l \in C\}.$$

A literal that appears with the same polarity in all the clauses of the formula is called *pure*. Similarly, a clause where all the literals have the same polarity is called pure.

An *interpretation* I is a set of non complementary literals (i.e. $I \cap \overline{I} = \emptyset$). A CNF formula F is *satisfied* by the interpretation I if and only if $I \cap C_i \neq \emptyset$ for every clause $C_i \in F$. Two CNF formulas F and F' are *semantically equivalent* $F \equiv F'$ if and only if they coincide in the set of interpretations that satisfy them i.e.

$$\{I \mid I \text{ satisfies } F\} = \{I' \mid I' \text{ satisfies } F'\}.$$

Resolution is an important operation between clauses. When a clause $C_i \in F$ contains a literal l and another clause $C_j \in F$ contains its complement $\bar{l}$, one can apply the *resolution rule* and add the resolvent $C_i \otimes C_j = ((C_i \setminus \{l\}) \cup (C_j \setminus \{\bar{l}\}))$ to the original formula, resulting in a semantically equivalent formula:

$$F \equiv F \cup \{C_i \otimes C_j\}.$$

From an algorithmic point of view it only makes sense to apply resolution when the resolvent is not a tautology i.e. when the resolution literal l is unique for the two clauses:

$$C_i \cap \overline{C_j} = \{l\}.$$

A clause C_i *subsumes* another clause C_j when the condition $C_i \subseteq C_j$ is satisfied. The subsumed clause can be removed from the original formula resulting in a semantically equivalent formula

$$F \equiv F \setminus \{C_j\},$$

because $I \cap C_i \neq \emptyset$ implies $I \cap C_j \neq \emptyset$ due to the set inclusion.

2.2 Graphs

Graphs are commonly used to describe the structure of a CNF formula and the complexity of the relations encoded inside [Her06]. A graph is defined as the structure $G = \langle V, E\rangle$ representing a set of *vertices* V (also called nodes) and a set of *edges* $E \subseteq V \times V$. For the nodes of G one can define the *out degree*:

$$deg_G(x) = \left|\{y \mid \langle x, y\rangle \in E\}\right|.$$

A graph is called *clique* when each node is connected to the other nodes in the graph. For all nodes x in a clique $deg_G(x) = |V| - 1$.

Sometimes we need to consider weighted edges, so we assign weights to them by defining a function:

$$w : E \mapsto \mathbb{R}$$

When we need to represent the relation between elements of different sets we then use the notion of *bipartite graph*, defined as the triple $G = \langle V_W, V_B, E\rangle$, representing two disjoint set of nodes V_W and V_B and a set of edges connecting only vertices from different sets $E \subseteq (V_W \times V_B) \cup (V_B \times V_W)$.

2.3 Machine Learning and Classification

In artificial intelligence, *machine learning* deals with the study and construction of algorithms that can learn from data [MRT12]. Usually, the algorithms are designed to use the learned knowledge to process better similar inputs in the future. In the presence of large amount of data, the task of finding hidden patterns, is known as *data mining* [WFH11].

Classification is a machine learning and data mining technique that is used to predict the membership of individuals to particular classes. The individuals are represented by a vector of features and the classes are the labels assigned to individuals defining a partition of the set of individuals.

The individuals we are interested in are the CNF formulas that will be represented as a vector of n numeric features $\mathbb{R}^n$ extracted from the formulas. The classification models used in this thesis are built based on binary classification (only two labels) representing when each of the algorithms in the portfolio is “good” or “bad” to be used for solving the CNF.

In general we can represent a classifier as a function

$$class_{pred} : \mathbb{R}^n \mapsto \{good, bad\} \times [0, 1]$$

where $class_{pred}(x).label$ represents the first component $\{good, bad\}$ and $class_{pred}(x).prob$ the corresponding probability $[0, 1]$.

Such a function is built during the process of *training* the classifier. We need a set of individuals and the membership of them to the defined classes

$$class : \mathbb{R}^n \mapsto \{good, bad\}.$$

The idea of this process is to come up with a prediction $class_{pred}$ as accurate as possible and also to avoid the *overfitting* to the training dataset. The accuracy of a classifier $class_{pred}$ is measured usually as the percent of individuals classified correctly of a given dataset T [WFH11]:

$$acc(class_{pred}, T) = \frac{|\{x \in T \mid class_{pred}(x).label = class(x)\}|}{|T|}$$

Overfitting occurs when attempting to model too closely the training dataset. One can by mistake also model the random noise within the data, resulting in a classifier that has a reduced predictive power even when the performance on the training dataset was very good. The overfitting can be avoided using *cross-validation* while performing the training process. A popular technique to perform cross-validation is called 10-fold cross-validation [WFH11]. The training set is randomly partitioned in 10 sets, and each of them is treated as a test set to measure the performance of the classifier created by training on the remaining 9 sets. Finally, the overall performance is the average performance across the 10 test sets.

2.3.1 Random Forest

Most of the popular classification techniques are based on *decision trees* [WFH11] and *random forest* [Bre01] has proven to be the most robust of them. The random forest algorithm operates building a collection of randomized decision trees. The resulting prediction of the random forest is the most frequent element (mode) of the predictions provided by the decision trees. According to the answer of all trees one can compute the probability of the mode by just counting the number of trees whose output coincides with the mode and then dividing this number by the number of trees of the forest.

Decision trees are defined as either a *leaf* with an output class or a *splitting node*

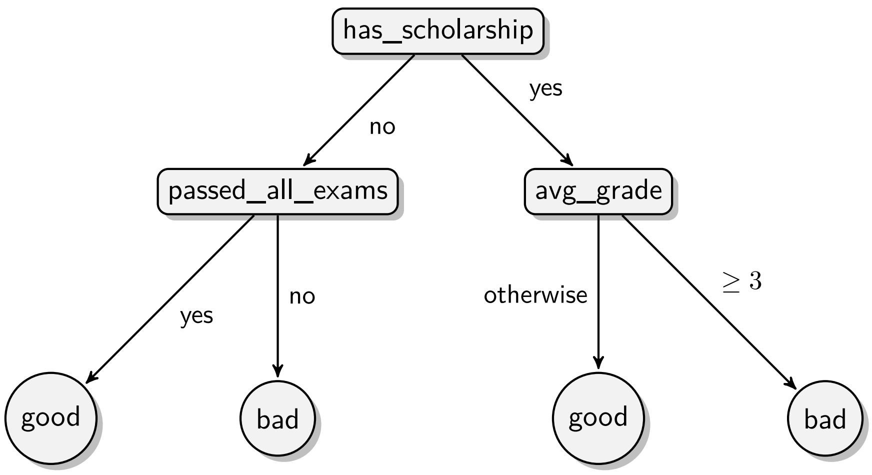

Figure 1: Decision tree example.

with more than one decision trees as sub-trees. The splitting node defines a partition of the values of some attributes. Each of the sub-trees is associated to a set of the defined partition. For a given instance we start at the root of the tree and as long as we are in a splitting node we compute to which of the sets of the partition defined by this node our instance belongs. Next, we move to the corresponding sub-tree. When we reach a leaf node we consider the output class in the node as the prediction of the tree. Sometimes leaf nodes can also contain a set of classifications, or a probability distribution over all possible classifications [WFH11].

In Figure 1 we can see a decision tree built with features defined for master students to decide if taking a student job is a “good” or “bad” idea. One can notice that the split nodes (squared shapes) have sub-trees that define a partition of the domain of the attribute in the node. If we have a student with a scholarship and his average grade is 3.5 we start on the root of the tree and move to the right sub-tree that handles students with scholarships. Next, we move to the right sub-tree that represents students with average grade greater than or equal to 3. We have reached a leaf, therefore we take the output class present in the node. For this particular student it is a bad idea to take a student job.

Given a set of instances for which the *class* function is defined we can compute the *information gain* [WFH11] of a split node by computing the difference of entropy of the class distribution before and after the split defined by the split node. For a given attribute a that takes values $\{v_1, \dots, v_k\}$ in a dataset T we can compute the information gain in the following way:

$$IG(T, a) = H(T) - \sum_{i=1}^{k} \frac{|\{x \in T | x_a = v_i\}|}{|T|} H(\{x \in T | x_a = v_i\}),$$

where x_a represents the value of the attribute a for the instance x of the dataset and

$H(A)$ is defined as the entropy of the class distribution in the set A. For the binary classification the entropy will be:

$$H(A) = \ln(|A|) - |\{x \mid class(x) = good\}| \ln(|\{x \mid class(x) = good\}|)$$
$$-|\{x \mid class(x) = bad\}| \ln(|\{x \mid class(x) = bad\}|)$$

To normalize the values of information gain the *intrinsic value* is used:

$$IV(T,a) = -\sum_{i=1}^{k} \frac{|\{x \in T | x_a = v_i\}|}{|T|} \log_2 \left(\frac{|\{x \in T | x_a = v_i\}|}{|T|} \right).$$

Finally, the *information gain ratio* is defined as

$$IGR(T,a) = \frac{IG(T,a)}{IV(T,a)}.$$

This magnitude can help us ranking the attributes of our domain, but in general the information gain ratio is just a measure that helps us to define greedy strategies. The problem of finding the optimal decision tree is known to be NP-complete [HR76].

Randomized decision trees are built by randomly selecting the attributes for the split nodes. The information gain ratio associated to the attribute is used to define the probability of selecting it. This process provides a more stable outcome and less influenced by random behaviors in the training dataset. The use of the *strong law of large numbers* shows that random forest always converge in such a way that overfitting is not a problem [Bre01]. Therefore, large number of trees produce more robust and stable results. The number of features to consider by each random tree is recommended to be $log(n) + 1$, where n is the number of features in the dataset [CWZ11].

2.3.2 k-Nearest Neighbors and Instance Based Classifiers

One of the simplest ways to learn things is to memorize them. When we memorize a set of training instances and a new instance is given to be classified we can search in the memorized instances the most similar instance and use the same output. This process is known as *instance-based* learning but in a sense all the other classification methods are also based on instance learning, but by memorizing the instances defines a instance-based way to represent the knowledge rather than on the other methods that we can use trees and rules to represent what is learned.

Mainly all the real work in instance-based learning is done when we need to classify a new instance rather than at the training process. In general the training will consist on preparing a data structure that allows us to search new instances. In instance-based classification when we have a new instance to classify we compare that instance with the existing ones using a distance metric and the closest instance is used to assign the class for the new instance. This process is known as the *nearest neighbor* classification method. Usually more than one nearest neighbor is used and the most frequent class between the neighbors is assigned to the new instance. This algorithm is then called the *k-nearest neighbors*.

To compute the distance between two instances generally we can use the Euclidean distance

$$d_e(x, x') = \sqrt{\sum_{i=0}^{n} (x_i - x'_i)^2}.$$

However it is better to normalize the attributes so that all of them have equal importance in the distance computation. Other distance measures can also be used when evaluating how close two instances are. When the number of features used to compute the distance is too high the computational cost is higher and all the points of the space tend to be equidistant from the instance we are trying to classify. Filtering of the features can be done by ranking them according to some univariate metric and selecting the features that are better ranked.

The process of searching the k nearest neighbors is simple and effective but it is usually slow. The straightforward way to find the k nearest neighbors is an algorithm of linear time complexity with respect to the size of the training set $|S|$. When we use a test set T to evaluate the algorithm the time complexity becomes proportional to the product of the two sizes $|S| \cdot |T|$.

The complexity of the algorithm for finding the k nearest neighbors can be reduced by representing the training set with a tree. A kD-Tree is a binary tree that divides the instances space with a hyperplane and splits again each resulting partition recursively. The splits are made perpendicular to one of the axes. In other words, in each node one of the attributes is making a split of the instances space by defining a hyperplane

$$x_i = \mathit{cutpoint}$$

perpendicular to the i-th dimension of the instances space.

Each node of the tree contains an n dimensional point x, the non-leaf nodes also specify the dimension i they are splitting. The left sub-tree only contains points x' such that $x'_i \leq x_i$ and the right sub-tree contains points x' such that $x'_i > x_i$.

Figure 2 shows an example of kD-Tree for a two dimensional space. Squared nodes are inner nodes and round nodes are the leaves. Inner nodes contain besides the point a dimension that splits the space in 2 subsets.

The search process in the kD-Tree is illustrated in Figure 3 by trying to search in the described example the closest point to $(5, 8)$. The tree is traveled from the root until the corresponding leaf by moving to the sub-tree that contains the points with the i-th dimension at the same side of the hyperplane defined by the node (dashed lines over the dots). In the root since $8 > 4$ we go to the right sub-tree. Now, since $5 < 6$ we go to the left sub-tree that is a leaf.

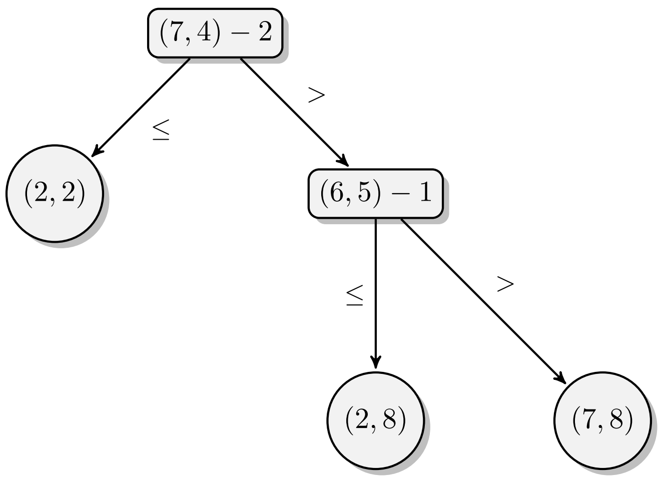

Figure 2: KD-Tree example.

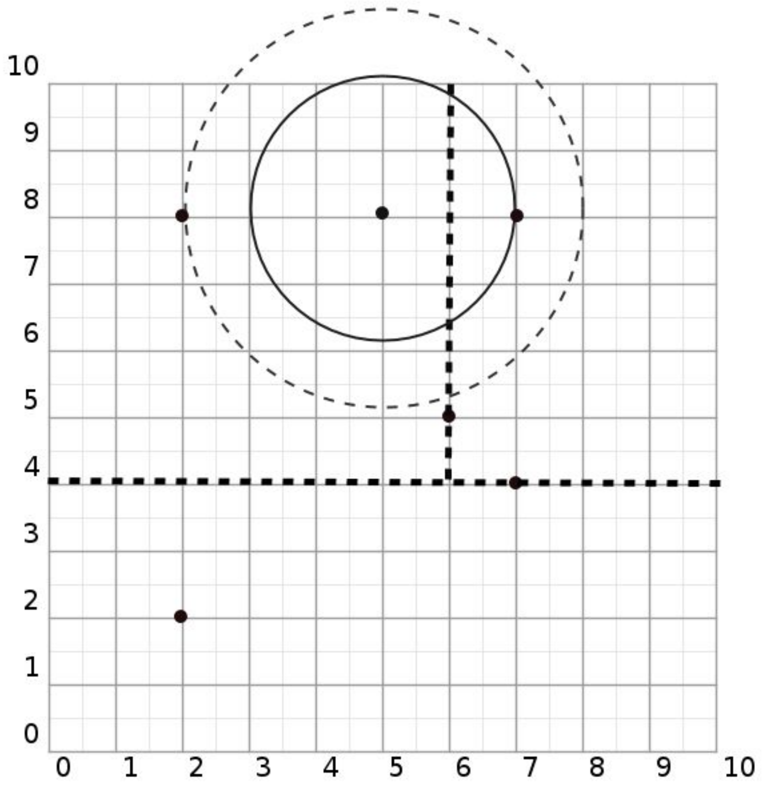

Figure 3: Search Process in a kD-Tree.

The resulting leaf node is the candidate for the closest neighbor but then we need to define a circle (dashed line circumference in the figure) centered in the point we are searching $(5,8)$ with circumference on current closest neighbor ($radius = 3$). By backtracking on the path we visited and checking whether the hyperplane intersects with the circle, we try to find points that are closer than the current candidate. When we backtrack to node $(6,5) - 1$ we find that the hyperplane intersects with the circle defined by $center = (5,8)$ and $radius = 3$, then we need to visit the other sub-tree. When we visit a node we compare with the current closest candidate and update it in case the visited node is closer to the point we are searching. In our case $(7,8)$ is closer and now we define a circle with $center = (5,8)$ and $radius = 2$ (solid line circumference in the figure). We now backtrack to the root node and since the hyperplane defined by it does not intersect with the circle we obtained before we do not visit the left sub-tree and the algorithm finishes with the closest point it could find $(7,8)$.

The k-nearest neighbors method can be used for classification tasks by finding the k closest neighbors to the instance that is going to be classified and then defining a way to compute the prediction based on the class associated to the k nearest neighbors. This classifier is known by the name IBk [WFH11]. The most common way to compute the prediction is to select the class with maximum weighted contribution of the k neighbors:

$$class_{pred}(X).label = c' \text{ such that maximizes the value } \sum_{class(x^{(i)})=c'} weight(x^{(i)})$$

where $x', \ldots, x^{(k)}$ are the k nearest neighbors of X, c' is one of classes labels and $weight : \mathbb{R}^n \mapsto \mathbb{R}$ represents the weights associated to the k neighbors. One of the ways that neighbors can be weighted is by the inverse of their distance to the instance X:

$$weight(x^{(i)}) = \frac{1}{0.001 + d(X, x^{(i)})}.$$

3 Related Work

Given a portfolio of algorithms and a SAT instance, this thesis focuses on the task of selecting one of the algorithms to obtain the best performance for the given instance. The performance can be considered as the time that takes the selected algorithm to solve the SAT instance or one could also just care only whether the instance was solved or not for a given timeout. To measure the performance of the algorithm selector, one also needs to add up the time spent by the selector before calling the selected solver from the portfolio of solvers.

Normally the selection task is done based on the runtimes of all the algorithms for the instances of a benchmark. The aim is to perform better than the best available algorithm and ideally as close as possible to the virtual best solver.

3.1 SATZilla

The first version of SATzilla solver [XHHLb07] was based on a runtime prediction model. For a given instance SATzilla would predict the runtime that each of the solvers in the portfolio would take to solve the instance. The fastest solver according to the prediction was selected to solve the instance. The implemented model used linear regression to predict the runtime based on the values of the features computed for the given instance.

To improve the performance of the initial version, SATzilla authors proposed another idea based on binary classification instead of regression. The authors used binary classifiers that for a given instance predicted the best solver for each pair of solvers in the portfolio [XHHLB12].

When asked to solve an instance SATzilla computes very cheap features and uses them to predict if the computation of the more expensive features will be possible within a certain timeout. In case the prediction is greater than the timeout they use the *backup solver* that has the best performance on instances with large feature costs. Otherwise, the algorithm sequentially runs presolvers trying to find fast solutions. If no solution is found by the presolvers the features are computed (calling the backup solver if the computation exceeds the defined timeout) and a prediction model that guesses the best solver for each pair of solvers in the portfolio is run. The solver with the majority of votes is selected.

The prediction model was built using a random forest of decision trees. The training process was based on a cost-sensitive technique that allows to compute the performance based on a cost function applied to the instance. In their case the authors tried to predict for a given instance which is the best solver (for each pair $\langle A, B\rangle$ of solvers) and the cost function is the absolute value of difference of the runtimes for the instance in both solvers $|time_A(x) - time_B(x)|$.

In a study carried out by SATzilla authors with the benchmarks of 2011 SAT competition [XHHLB12], SATzilla has outperformed consistently the solvers used

Size Features

1-3. Number of clauses c, Number of variables v, Ratio v/c.

Variable-Clause Graph Features

4-8. Variable nodes degree statistics: mean, variation coefficient, minimum, maximum and entropy.
9-13. Clause nodes degree statistics: mean, variation coefficient, minimum, maximum and entropy.

Balance Features

14-16. Ratio of positive and negative literals in each clause: mean, variation coefficient and entropy.
17-21. Ratio of positive and negative occurrences each variable: mean, variation coefficient, minimum, maximum and entropy.
22-23. Fraction of binary and ternary clauses.

Proximity to Horn Formula

24. Fraction of Horn clauses.
25-29. Number of occurrences in a Horn clause for each variable: mean, variation coefficient, minimum, maximum and entropy.

Figure 4: SATzilla features used by ArgoSmArT k-NN system.

in the portfolio, other solvers inscribed in SAT competition and also previous version of SATzilla. SATzilla authors have also found out that the solvers with best contributions to SATzilla were not often competitions winners but mainly solvers that were able to solve instances that no other solver could solve. This encourages programmers to think about efficient solvers for specific domains rather than a general purpose solver.

3.2 ArgoSmArT k-NN

Based in a SAT instance, ArgoSmArT k-NN system selects a solver from a portfolio of solvers [NMJ11]. Instances are represented using a subset of the features proposed by SATzilla authors (Figure 4).

When an input instance is given, the features are computed and according to a *distance metric* the closest k instances from the training set are found. The ArgoSAT solver is run then with the solver S_i that has the smallest *penalty* for the k instances. In case of ties the solver that performs better in the whole training set is selected.

Distance metric: The authors used a distance that performed well in previous experiments [NMJ09]:

$$d(x, y) = \sum_i \frac{|x_i - y_i|}{1 + \sqrt{x_i y_i}}$$

where x_i and y_i are the values of feature i for instances x and y. Note that values are not normalized for this distance metric.

Penalty: To compute the penalty over a set of instances the authors use the sum of the PAR10 score for each of the instances. The score is computed as follows:

$$PAR10_{S_i}(x) = \begin{cases} time_{S_i}(x) & \text{if } time_{S_i}(x) \leq timeout \\ 10 \cdot timeout & \text{otherwise.} \end{cases}$$

The ArgoSmArT k-NN system was evaluated using the benchmark and solvers that SATzilla first version used to evaluate their system. The results showed that ArgoSmArT k-NN performed better than the first version of SATzilla .

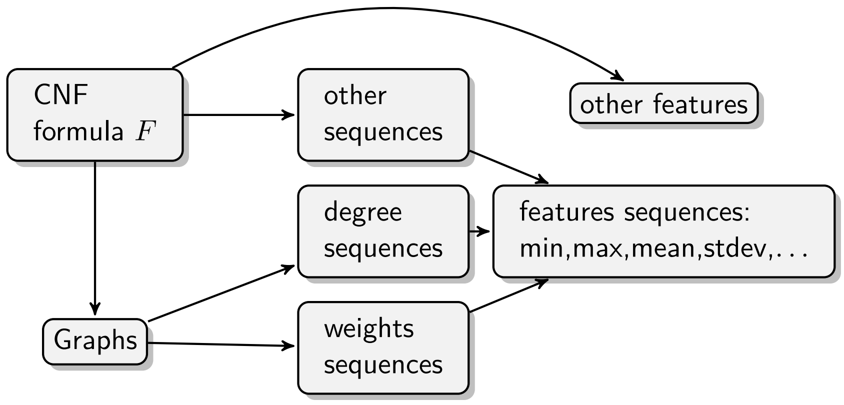

Figure 5: Features extraction overview.

4 Features

For the machine learning task the CNF instances are represented as a vector of features that describe the information encoded. The features used in this thesis were proposed in our previous work [AM14a]. Additionally, the XOR gate extraction is implemented for this thesis.

In Figure 5 we can see an overview of the features extraction process. The key aspects on the features computation are the graphs that are computed from the CNF formula, the sequences of degrees and weights computed form those graphs and also some other sequences computed directly from the formula. Additionally, there are some features that are not computed for sequences present in the formula.

4.1 Graphs Features

The main source of the computed features are the graphs that describe relations encoded in the CNF formula. For those graphs the features are computed based on the sequence of weights in the edges and also on the sequence of out-degrees of the nodes. Some of the graphs computed were used before in the features proposed by SATzilla [NLBH+04, XHHLb07] but without considering the weights.

In Table 1 we can see the list of graphs computed in our previous work. For all of them we considered the sequence of the degrees and for most of them we also considered the sequences of weights as shown in the table. The *Clause-Variable graph* (CV^+ for positive literals, CV^- for negative literals) connects (positive or negative respectively) literals with the clauses in which they appear. In the *Variables graph* two variables are connected when they appear in the same clause without considering the polarity. The *Clause graph* connects clauses that share literals. The *Resolution graph* connects clauses when they produce a non-tautological resolvent. The *binary*

Graphs	Vertices	Edges	Weights
CV$^+$ (CV$^-$)	$B = \{1, \ldots, m\}$ $W = \{1, \ldots, n\}$	$\langle i, j\rangle$ and $\langle j, i\rangle$ iff $j \in C_i$ $(-j \in C_i)$	-
Variables	$V = \{1, \ldots, n\}$	$\langle i, j\rangle$ iff there is $1 \leq k \leq n$ $\{i, j\} \subseteq (C_k \cup \overline{C_k})$	2^{-k}
Clauses	$V = \{1, \ldots, m\}$	$\langle i, j\rangle$ iff $C_j \cap C_i \neq \emptyset$	$\lvert C_j \cap C_i\rvert$
Resolution	$V = \{1, \ldots, m\}$	$\langle i, j\rangle$ iff $\lvert C_i \cap \overline{C_j}\rvert = 1$	$2^{-(\lvert C_i \cup C_j\rvert - 2)}$
BIG	$V = \{\pm 1, \ldots, \pm n\}$	$\langle \bar{i}, j\rangle$ iff $\{i, j\} \in F$	$2^{-(\lvert C_i \cup C_j\rvert - 2)}$
AND-GATE	$V = \{\pm 1, \ldots, \pm n\}$	for each $l_0 \leftrightarrow l_1 \wedge \ldots \wedge l_k$ we add all edges $\langle l_0, l_i\rangle$.	2^{-k}
BAND-GATE	$V = \{\pm 1, \ldots, \pm n\}$	for each $l_0 \rightarrow l_1 \wedge \ldots \wedge l_k$ we add all edges $\langle l_0, l_i\rangle$.	2^{-k}

EX1L-GATE	$V = \{\pm 1, \ldots, \pm n\}$	for each $EX1L(l_1, \ldots, l_k)$ we add
		for each $i \neq j$ add edges $\langle l_i, l_j\rangle$.
			-

Table 1: Graph structures considered for features computation of a formula F with $|F| = m$, and n variables.

implication graph (BIG) contains the implications between literals that appear in binary clauses in the formula. The AND-gate graph represents the relation between literals that belong to an AND-gate ($l_0 \leftrightarrow l_1 \wedge \ldots \wedge l_k$) encoded in the CNF formula. When the encoding of an AND-gate is partially present in the formula and the rest of the clauses can be added by blocked clause addition [Kul99] a *blocked* AND-gate (BAND-gate) is recognized and the BAND-gate graph contains the relations between the literals in the recognized gate. The EX1L-gate graph contains the relation between literals that belong to the *exactly one literal* gates encoded in the formula.

In the clauses variable graph, the sequence

$$deg^{\pm}(i) = \frac{max(deg_{CV^+}(i), deg_{CV^-}(i))}{deg_{CV^+}(i) + deg_{CV^-}(i)}$$

is used for the features computation, representing the purity degree of each clause and each variable.

4.1.1 XOR Graph

XOR constraints are important given the fact that many SAT problems contain them (especially cryptographic ones). Also we can find some modifications of the DPLL techniques to deal with parity constraints (XOR) [LJN10]. The XOR constraint is a logical expression represented in the following way:

$$l_1 \oplus \ldots \oplus l_k.$$

The formula states that only an odd number of the literals present in the expression can be mapped to true by a satisfying interpretation. The XOR constraint is usually represented as a CNF formula S with 2^{k-1} clauses that contain k literals. The number of negative literals in each of the clauses in the formula S will have the same *parity*.

To search the XORs encoded in a CNF formula F it is very convenient to sort first the literals of each of the clauses in F according to the modular value and also to sort all the clauses in the formula according to their size, then according to the variables they contain and finally according to the parity of the negative literals count. To obtain the described order, first the literals of each clause $C_j = [l_1, \ldots, l_m]$ need to be sorted according to the value $|l_i|$, then the clauses C_j can be represented by a triple

$$\langle m, varlex(C_j), negpar(C_j) \rangle ,$$

where $varlex(C_j) = \langle |l_1|, \ldots, |l_m| \rangle$, and $negpar(C_j) = |\{l_i \mid l_i < 0\}| \; mod \; 2$. Finally one can sort the clauses in the CNF formula with the $\preceq_n$ relation defined for tuples of size n:

$$\langle x_1, \ldots, x_n \rangle \preceq_n \langle x'_1, \ldots, x'_n \rangle \equiv x_d \preceq^{(d)} x'_d,$$

where d is the first position such that $x_d \neq x'_d$ and the $\preceq^{(d)}$ is an ordering relation defined for the elements in position d. In case x_d is an integer $\preceq^{(d)}$ will be the normal *less than* $<$ relation defined for integers. In case x_d is a tuple of dimension n_d, the relation $\preceq^{(d)}$ will be the relation $\preceq_{n_d}$ defined for tuples of size n_d.

After this process the clauses with the same size, variables and negative literals parity will be located next to each other. The XOR formulas S encoded in F will then be the groups of 2^{k-1} consecutive clauses with size k, with the same variables and with the same parity of the number of negative literals. In case that not all the 2^{k-1} clauses can be found, it can be the case that some of the missing clauses can be deduced using subsumption. The subsumption can be easily checked by searching clauses with a subset of the k variables in the candidate XOR encoding.

The XOR graph will have as nodes the set of literals of the formula. For each XOR $l_1 \oplus \ldots \oplus l_k$ encoded in the formula the clique between the literals $l_1, \ldots, l_k$ is added to the XOR graph with weights 2^{-k} associated to the edges. Sequences of weights and nodes degrees present in the XOR graph are considered for the features computation.

4.2 Sequences Features

Other sequences used come from the iterative computation of the Recursive Heuristic Weights and the Symmetry [AM14a]. The Recursive Heuristic Weights sequence computes values associated to the literals in the formula that represent the tendency of each literal to be present in the final model. On the other hand, the symmetry sequence computes values associated to variables in the formula such that variables with the same associated value belong to the same symmetry group.

For the computed sequences $S = \langle x_1, \dots, x_n \rangle$ we extract several parameters to be considered:

- The *minimum*(min) of the sequence, defined as the smallest value of the sequence.
- The *maximum*(max) of the sequence, defined as the greatest value of the sequence.
- The *mode*, defined as the most frequent value of the sequence.
- The *mean* of the sequence, computed in the following way $mean = \frac{1}{n} \sum x_i$.
- The *standard deviation* of the sequence, defined as $stdev = \frac{1}{n-1} \sum (x_i - mean)^2$.
- The *values rate* $\frac{1}{n} |\{x_1, \dots, x_n\}|$ to give an idea of how different are the values present in the sequence.
- The zero count for some sequences in which the zero values can be frequent and have a special meaning.
- The *entropy* of the sequence, computed as follows: $entropy = \ln(n) - \sum c_i \ln(c_i)$, where c_i are the counts of each different element that appears in the sequence.
- The values $Q_1, \dots, Q_{k-1}$ being the k-*Quantiles* dividing the sequence after being ordered in k regular intervals of approximately the same number of elements.

The derivative of the sorted sequence of values $derivative(S) = \langle (x'_2 - x'_1), \dots, (x'_n - x'_{n-1}) \rangle$, where $sort(S) = \langle x'_1, \dots, x'_n \rangle$ is also be used as another sequence for features computation.

4.3 Other Features

Additionally we also consider the number of clauses, of unit (2 literals, 3,...8, 9 or more literals) clauses and of horn clauses present in the formula. Also the computations time and steps are computed for some of the features groups. In Table 2 one can see some names and descriptions of these features.

Name	Description
clauses	Number of clauses.
vars	Number of variables.
	Number of clauses of size:
clauses_size_1	1
clauses_size_2	2
clauses_size_3	3
clauses_size_4	4
clauses_size_5	5
clauses_size_6	6
clauses_size_7	7
clauses_size_8	8
clauses_size_ >= _9	9 or more.
horn_clauses	Number of horn clauses in the formula.
	Number of steps used to compute:
Clause_graph_steps	the clause graph,
Clause-Var_steps	clause-variable graph,
Var_graph_steps	variable graph,
Bin_Implication_graph_steps	binary implication graph,
Exactly1Lit_steps	exactly one constraint graph,
Full_AND_gate_steps	full AND graph,
Blocked_AND_gate_steps	blocked AND graph,
XOR_gate_steps	XOR graph,
Symmetry0_steps	iteration 1 of symmetry recognition,
Symmetry1_steps	iteration 2 of symmetry recognition,
Symmetry2_steps	iteration 3 of symmetry recognition,
Symmetry_computation_steps	complete symmetry recognition,
RWH-1steps	iteration 1 of RWH algorithm,
RWH-2steps	iteration 2 of RWH algorithm,
RWH-3steps	iteration 3 of RWH algorithm,
Resolution_graph_steps	and resolution graph.
	Time spent to compute:
Clause-Var_and_Var_graphs_time	clause-variable and variable graph together,
Resolution_and_Clause_graphs_	resolution and clause graph together,
Bin_Implication_graph_time	binary implication graph,
Constraints_recognition_time	native encoded constraints,
Symmetry_computation_time	and symmetry features.
	(not considered by classifiers)

Table 2: Simple and time related features.

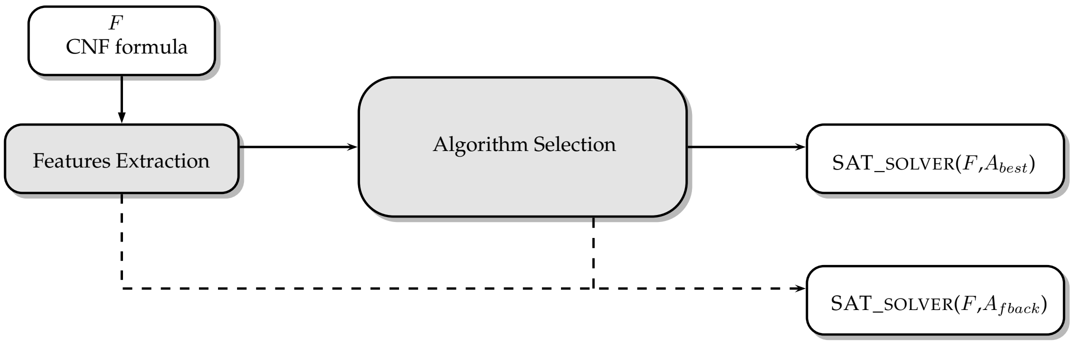

Figure 6: Overview of the execution of the algorithm portfolio model.

5 Proposed Models

In this thesis we explore binary classification models based on random forest classifiers and also on k-nearest neighbors classifiers. The classification is used to decide when an algorithm is "good" or "bad" to be used on a given instance. A model based on the k-nearest neighbors method but not in binary classification is also proposed.

Figure 6 shows the overview of the algorithm portfolio execution process. The process starts with the input of a CNF formula F and then the features for F are extracted. After the feature extraction one of the available algorithms is selected using machine learning techniques. Finally, the selected algorithm A_{best} tries to obtain the solution of the problem encoded in the SAT instance F.

The process of features extraction and algorithm selection have limited time and memory resources. Therefore, in case one of the process fails the *fall-back* algorithm A_{fback} tries to solve the instance. The best algorithm in the training set is normally used as the fall-back algorithm.

An important detail in machine learning algorithms is the training process. For training the models defined in this thesis a benchmark of SAT instances is needed. For those instances in the training set the features are computed and also the time that each algorithm of the portfolio takes to solve the instance.

5.1 Binary Classification Models

The selection of an algorithm using binary classification is performed by associating to each algorithm present in the portfolio a binary classifier that is able to predict when the algorithm is "good" or "bad" for a given instance.

For each algorithm A_i the criteria of classification for a certain CNF formula considers the time spent by the algorithm to find the solution of the formula and defines a binary partition of the time domain. Three different criteria that let us define the class membership are explored.

The simplest criteria is called “global” and states whether there is enough time to solve the formula after computing the features and selecting the algorithm A_i. In other words, if the sum of the time used for the algorithm $time_{A_i}(x)$, the features computation time $time_f(x)$ and an estimate of the classification time b is less than a certain timeout:

$$class_{A_i}(x) = \begin{cases} good & \text{if } time_{A_i}(x) + time_f(x) + b < timeout. \\ bad & \text{otherwise.} \end{cases}$$

The “relative” criteria is defined with respect to the fall-back algorithm A_{fback}. It defines when a certain algorithm is better than the A_{fback}:

$$class_{A_i}(x) = \begin{cases} good & \text{if } time_{A_i}(x) + time_f(x) + b < \\ & \min(timeout, time_{A_{fback}}(x) + time_f(x) + b) \\ bad & \text{otherwise.} \end{cases}$$

Relative criteria will only be defined for $i \neq fback$.

Finally, the “complement” criteria is defined with respect to a fall-back algorithm A_{fback}. It is very similar to the “global” criteria but an algorithm A_i with $i \neq fback$ cannot be “good” if the A_{fback} is already “good”:

$$class_{A_i}(x) = \begin{cases} good & \text{if } time_{A_i}(x) + time_f(x) + b < timeout \\ & (i = fback \ \vee \ class_{A_{fback}}(x) = bad) \\ bad & \text{otherwise.} \end{cases}$$

Using one of the defined class partitioning criteria and the information about the runtimes for each algorithm in the portfolio present in the training dataset, the class partitioning is done.

The features computed for each instance and the class membership computed is used to train a binary classifier associated to each algorithm A_i. The classifier is able to predict given the features of a newly introduced instance if the algorithm is “good” to be used with the instance or not.

Normally classifiers also give a probability for each of the possible outcomes. The interesting information is the probability of the outcome to be “good” because it provides an idea of the truth degree of the fact that the configuration is good to be used with the instance. Finally, given all the probabilities of each algorithm A_i to be good on the instance, the highest value is selected. The algorithm associated to the highest value is the algorithm with the highest probability to be good on the given instance.

In this thesis random forest and k-nearest neighbors classifiers are part of the experiments. Both techniques are used to build the binary classifiers to know when an

Version	Description
RF	Number of trees: $10 \cdot n$
	Number of features to consider: $log(n) + 1$.
	Depth: unlimited.
RF1	Cost-sensitive version of RF.

Table 3: Explored RF classifiers.

Version	Description
IBK	Number of Neighbors: 2
	Distance: Euclidean.
	Weighted contribution: 1 for all instances.
IBK1	Number of Neighbors: Found using cross-validation.
	Distance: Euclidean.
	Weighted contribution: Inverse of the distance.

Table 4: Explored IBK classifiers.

algorithm A_i is "good" or "bad" for a given SAT instance.

5.1.1 Random Forest Classifiers

Two versions of random forest are defined for the experiments. The specific parameters used for both versions are shown in Table 3. The first one (RF), is a random forest with the recommended parameters in the literature [CWZ11]. The second version (RF1) is a cost-sensitive version of the RF version. The cost-sensitive is version is built by varying the proportion of instances in the training dataset according to their importance. The proportion of instances can be varied in a simple way by setting weights to the instances in the training dataset. The weights are assigned to the instances based on how many times the instance was labeled as "good":

$$weight_{A_i}(x) = \begin{cases} \frac{1}{goodcount(x)} & \text{if } class_{A_i}(x) = bad \\ \\ \frac{2}{goodcount(x)} & \text{otherwise,} \end{cases}$$

where $goodcount(x) = |\{A_i \mid class_{A_i}(x) = good\}|$.

5.1.2 Instance Based Classifiers

For the binary classification problem the k-nearest neighbors method can be used. The class prediction is computed by selecting the class to which the majority of the k nearest neighbors belong to. The instance based binary classifiers (IBK) tested in

Version	Description
k-NN	Number of Neighbors: 200 Distance: Euclidean. Weighted contribution: 1 for all instances.
k-NN1	Number of Neighbors: 200. Distance: $d(x,y) = \sum_i \frac{\lvert x_i - y_i \rvert}{1+\sqrt{x_i y_i}}$ used in related work [NMJ11]. Weighted contribution: Inverse of the distance.

Table 5: Explored k-NN classifiers.

this thesis are based on the euclidean distance. For computing the distance between two instances only 15 features are used. The features with higher information gain ratio are the ones considered to compute the distance. Table 4 shows the two versions of IBK classifiers used in this thesis for the experiments and their parameters.

5.2 k-Nearest Neighbors Models

The information in the training sets used has the following structure:

$$\mathbb{R}^n \times \{good, bad\}^m$$

where n is the number of features and m is the number of algorithms in the portfolio and for an instance X of the training we have that $class_{A_i}(X) = x_{n+i}$.

The structure above suggests that instead of building a classifier for each of the m algorithms in the portfolio, we could use the k-nearest neighbors method to find the closest instances $x', \ldots, x^{(k)}$ and then with the class information $class_{A_i}$ ($i \in \{1, \ldots, m\}$) of those instances compute the algorithm A_* to be selected by doing a modified weighted contribution:

$$A_* \text{ such that maximizes the value } \sum_{class_{A_*}(x^{(i)})=good} weight(x^{(i)})$$

Table 5 shows the k-NN prediction models used in this thesis and the description of the parameters used.

Version	Features	Description
None	118	Clause-variable and variable graphs, Symmetry sequence and Other features.
Xor	154	None + XOR graph.
Big	131	None + BIG.
Derivative	217	None + Derivative of the sequences in None.
Res	141	None + Resolution graph.
Rwh	154	None + RWH sequence.
Const	188	None + BIG, EX1L, AND & BAND graphs.
NoCGResXor	422	Const + RWH and the derivative of the sequences.
Clause	141	None + clause graph.
All	556	All the groups of features and the derivative of the sequences.

Table 6: Description of the versions of features computation.

6 Experimental Results

The instances of the SAT competitions benchmarks from year 2009 until 2013 were filtered to remove the repeated instances and to work only with the instances solved by at least one of the algorithms in our portfolio of algorithms.

The computation of the algorithms runtimes for the instances of the benchmark was carried out with a CPU time limit of 900 seconds and the memory limit was 7 GB on an Intel Xeon CPU ES-2670 with 2.6 GHz. The computation of the different features versions for each instance was executed with a CPU time limit of 100 seconds and the memory limit was also 7 GB on a cluster that uses AMD Opteron 6274 CPUs with 16 cores and 2 MB level 2 cache that is shared by two cores. In the following subsections the results of the experiments are presented.

6.1 Features Relevance and Performance

The features computation was run in 9 different versions with different parameters allowing to experiment with different sets of features. Table 6 shows the description of all the different features computation versions and the number of features computed in each case.

Table 7 shows for each features computation version, the number of instances with computed features for a 100 seconds timeout. The table is sorted in a decreasing order according to the number of instances with features computed. For the top 6 versions of features computation there was no significant difference with respect to the number of instances they were able to process. Then the Xor version was not able to compute the features for 36 instances of the benchmark and the Res version

Version	Computed Instances
Total	4719
None	**4719**
Big	**4718**
Derivative	**4717**
Rwh	**4716**
Const	**4713**
NoCGResXor	**4713**
Xor	4683
Res	4617
Clause	4497
All	4493

Table 7: Instances with computed features for each version of features computation on a 100 sec timeout.

was not able to compute the features for 102 instances of the benchmark. For the last 2 versions, the features of more than 200 instances could not be computed, that represents approximately 4 % of the total number of instances. These two last versions include the clause graph computation while the rest of the feature computation groups do not include it.

Figure 7 shows the runtime cactus plots for all the features computation versions defined. As can be seen in the figure, the performance shown by the plots were very similar for the computation versions that do not include the resolution and the clauses graphs. For these versions the features were computed in less than 10 seconds for 4500 instances. The remaining instances showed a significant increase in the time needed to compute their features.

The versions that compute the clause graph have the worst runtime performance but even for them we could compute the features in less than 20 seconds for 4300 instances. For the remaining instances computation time increased significantly.

The Res runtime performance was better than the versions that compute the clauses graph but not as good as the features versions that skip the computation of the clauses and resolution graphs.

In Figure 8 we can see the memory usage cactus plots. The plot for all features computation versions resembles in shape to the corresponding runtime cactus plots. Table 8 shows the correlation coefficients between the runtime and the memory used for each one of the features computation versions. It can be seen that indeed the runtime and the memory used were highly correlated, having correlation coefficients greater than or equal to 0.8 in all cases. In terms of algorithms efficiency having a very high (close to 1) correlation coefficient between runtime and memory means that

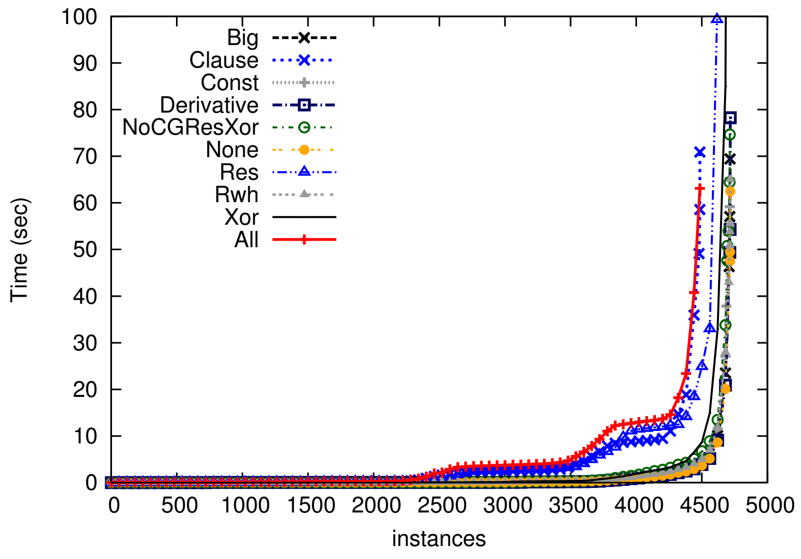

Figure 7: Runtime for the different feature computation versions.

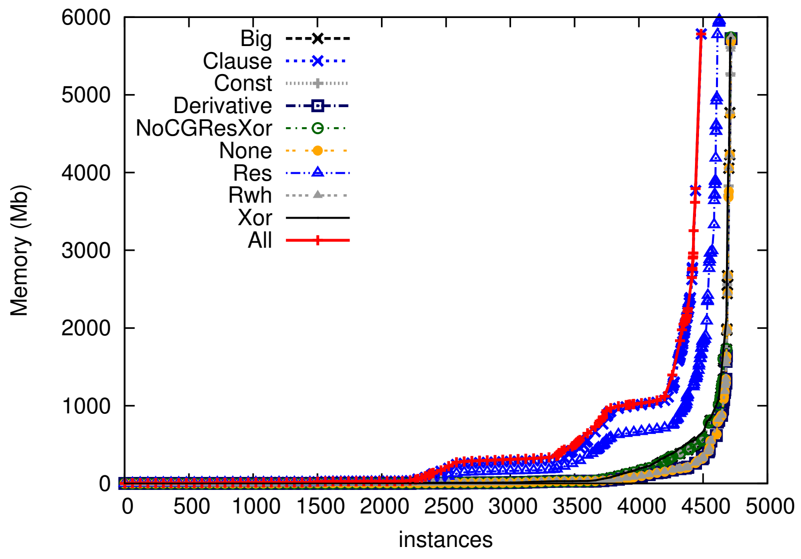

Figure 8: Memory usage for the different features computation versions.

Version	Correlation
All	1.00
Big	1.00
Clause	0.94
Const	1.00
Derivative	0.80
NoCGResXor	0.96
None	0.85
Res	0.91
Rwh	0.81
Xor	0.96

Table 8: Correlation coefficient between runtime and memory for the different features computation versions.

the relation between runtime and memory consumption is almost linear.

When the clauses and resolution graphs were not included the features could be computed using less than 1 GB of memory for 4500 instances. For the remaining instances the memory needed to compute their features increased rapidly.

When the clauses graph was included the memory usage performance was worse but for 4300 instances the features could be extracted using less than 1.5 GB of memory. The memory required to extract the features of the rest of the instances increased significantly.

The Res memory usage (similarly as the runtime) was better than the versions that compute the clauses graph but not as good as the features versions that skip the computation of the clause and resolution graphs.

The information gain ratio was computed as a measure of the importance of the features used. Figure 9 shows the plots of the IGR for each of the different versions of features computation for the "global" class partitioning criteria. The plots are reverse cumulative distributions that describe how many features have a IGR greater than or equal to a certain value in the IGR axis. One can notice that the IGR for all the versions varied in the same range $[0, 0.14]$ and the main difference was the area bellow the curve that they describe, which was proportional in size to the number of features computed in each version. The plots are very different for small values of IGR but they get closer for bigger values of IGR (greater than or equal to 0.1), which suggest that even when more features are computed the number of high ranked features remains the same.

Table 9 shows the two features with highest IGR for all the versions of features computation based on the "global" class partitioning criteria.

The zero count for the degrees sequence of the variable-clause graph (variable-clause_degree_zcount), with IGR of 0.13 for all versions of features computation

Version	Feature Name	IGR
ALL	XOR_gate_weights_derivative_entropy	0.14
	XOR_gate_weights_derivative_stdev	0.13
BIG	variable-clause_degree_zcount	0.13
	bin_implication_graph_degree_min	0.11
CLAUSE	variable-clause_degree_zcount	0.13
	clauses_size_4	0.12
CONST	variable-clause_degree_zcount	0.13
	bin_implication_graph_degree_min	0.11
DERIVATIVE	variable-clause_degree_zcount	0.13
	clauses_size_4	0.11
NOCGRESXOR	variable-clause_degree_zcount	0.13
	Exactly1Lit_derivative_mode	0.12
NONE	variable-clause_degree_zcount	0.13
	clauses_size_4	0.11
RES	variable-clause_degree_zcount	0.12
	clauses_size_4	0.11
RWH	variable-clause_degree_zcount	0.13
	clauses_size_4	0.11
XOR	variable-clause_degree_zcount	0.13
	bin_implication_graph_degree_min	0.11

Table 9: Features with highest IGR for all features computation versions.

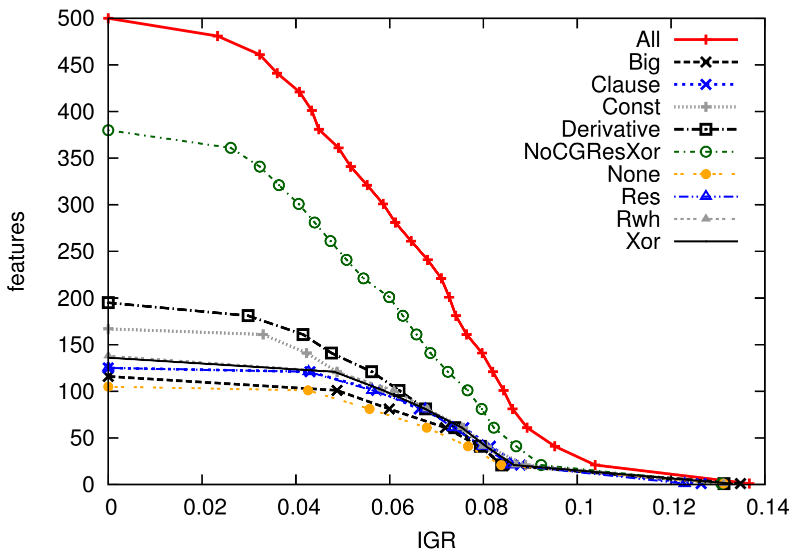

Figure 9: IGR for the features of the different feature computation versions.

was only exceed by the entropy of the derivative of the sequence of weights in the XOR_gate graph (XOR_gate_weights_derivative_entropy) with a IGR of 0.14. Even if the the Xor version also computes the XOR_gate_weights_derivative_entropy, the value of the IGR was smaller. The class distribution changed because not all features computation versions were able to compute the features for the same set of instances. Consequently, the IGR of some features also changed.

For the features computation, one of the aspects to pay attention to is how correlated are the values of those features. If the features have highly correlated values, the information computed is redundant and can affect the efficiency of the features computation because more memory is used and more time is consumed on information already present. The classification process can also be affected when the features values are too similar because the probability of using the redundant information is higher than using the non-redundant information when the classifier is built.

Figure 10 shows the correlation matrix for the values of all the computed features. Black areas represent high correlation coefficients (grater than 0.7 or also lower than -0.7) and white areas represent the correlation coefficients in the range $[-0.7, 0.7]$ that defines the non-redundant information present. The 96 % of the correlation coefficients fell in the range of non-redundant information.

The clauses_graph_steps and resolution_graph_steps features had the highest correlation coefficients with respect to the total time of the features extraction with values of 0.89 and 0.69 respectively. Hence, the time to compute the features was strongly influenced by the steps taken to build the clauses and resolution graphs. Although, the influence was bigger in the case of clauses_graph_steps.

Furthermore, the correlation coefficients of the features computed with respect to

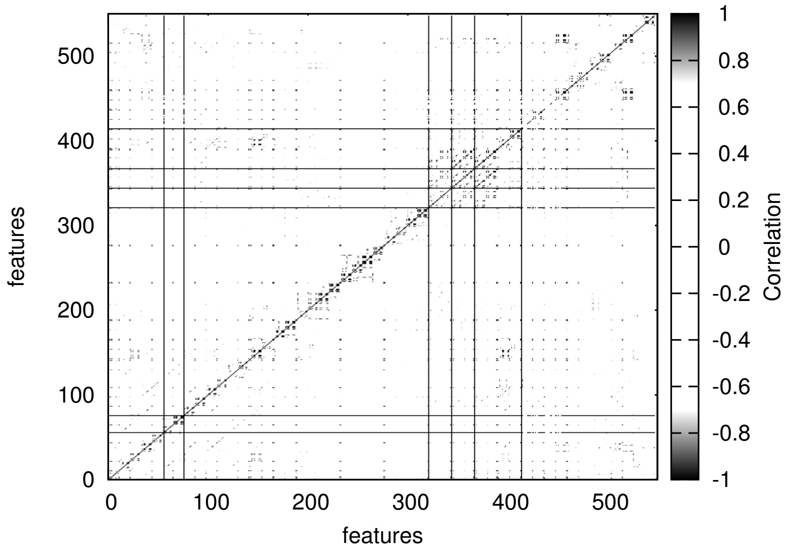

Figure 10: Correlation matrix for all the features computed.

the runtimes of the configurations was never greater than 0.25. Therefore, the runtimes of the configurations are not linearly determined by any of the computed features.

6.2 Algorithms Contributions

The algorithms used in the portfolio determine how good will be the performance of the selection model. In general, algorithms that solve instances that other algorithms do not solve (*unique contributions*) play a very important role in the portfolio. As a method for scoring the algorithms in a portfolio one can use a simplified version of the *purse score* [VGLBB$^+$05] where only the *solution purse* is considered. The solution purse is a value (one in this thesis) assigned to each instance in the benchmark. The solution purse of each instance is equally divided among the solvers that solve the instance. The purse score of a solver is then sum of all the fragments of solution purse provided by all the instances it solved. The number of solved instances and the number of unique contributions can also be used as measures to rank the algorithms of our portfolio.

Table 10 shows for each algorithm used in the portfolio the number of solved instances in the benchmark, the percent it represents from the total number of instances in the benchmark, the simple purse score and in the last column how many unique contributions the algorithm provided. The table is sorted in a decreasing order according to the number of solved instances. All the algorithms in the portfolio represent configurations of the parameters of the Riss solver [Man14] coming from the SAT Competition 2013 and the Configurable SAT Solver Challenge (CSSC) 2013 and some additional ones tuned for the 2014 SAT competition.

Configuration	Solved Instances	Percent	Simple Purse Score	Unique Contributions
VBS	4719	100 %	-	-
Lingeling	4444	94.2 %	-	-
Riss427-NoCLE	4408	93.4 %	446.5	0
Riss427	4408	93.4 %	446.3	1
RissND427	4399	93.2 %	442.9	0
PRB	4394	93.1 %	452.7	1
NOTRUST	4374	92.7 %	445.1	3
EDACC5	4393	93.1 %	451.0	1
EDACC6	4394	93.1 %	450.8	4
SUHLE	4383	92.9 %	442.4	2
FASTRESTART	4357	92.3 %	442.9	5
EDACC7	4389	93.0 %	449.0	2
Riss3g	4370	92.6 %	451.8	9
RERRW	4370	92.6 %	447.4	3
XBVA	4361	92.4 %	440.4	1
XOR	4360	92.4 %	443.0	3
fourthCL	4305	91.2 %	452.2	6
LABS	4200	89.0 %	446.7	18
RATEBCEUNHIDE	4180	88.6 %	429.1	8
RISSLGL3	3475	73.6 %	357.6	7
RISSLGL4	3257	69.0 %	345.6	15

Table 10: Solved instances for 900 sec timeout in whole benchmark.

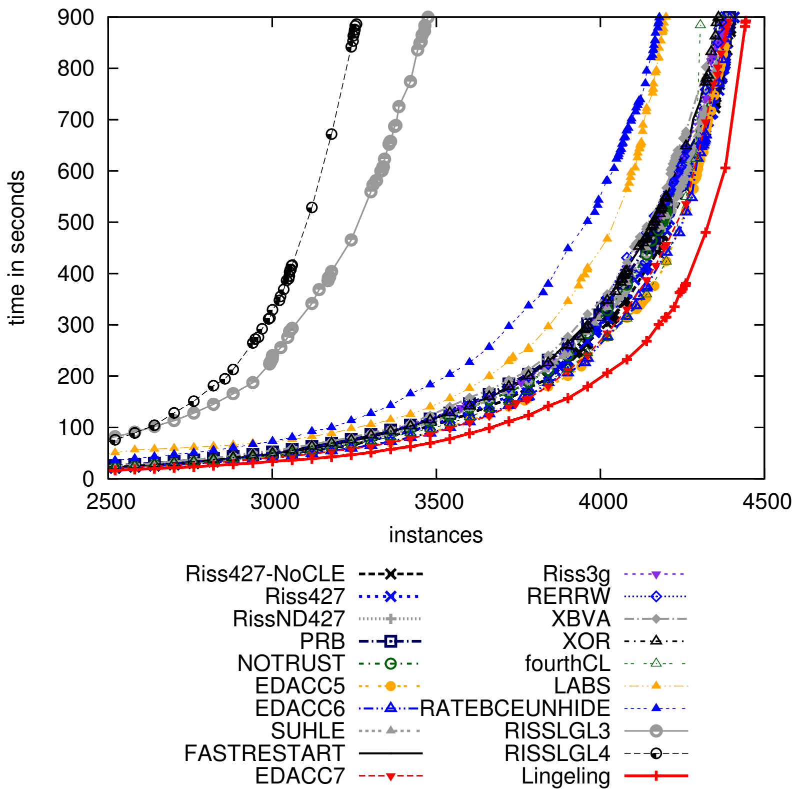

Figure 11: Cactus plot for the algorithms in the portfolio and Lingeling solver.

Notice that the configurations Riss427-NoCLE and RissND427 had zero unique contributions which means that the instances solved by them were also solved by the rest of the configurations. Additionally, the correlation index between Riss427-NoCLE and RissND427 was rather high (0.88) which suggest that one of them can be removed from the portfolio without changing the number of instances solved by the VBS. In general, one can see that the number of unique contributions for most of the configurations was lower than 5. In the table the configurations that solved less instances have higher number of unique contributions. Finally, the configurations Riss427-NoCLE and Riss427 solved the same number of instances but the first had a greater simple purse score and was selected as the fall-back configuration to try to solve the instances when the features computation or the algorithm selection process fail.

Figure 11 shows the cactus plots of the runtimes of the configurations used together with the runtime of the Lingeling solver [Bie13]. On the other hand, Figure 12 shows the correlation between the solving times of the configurations for the whole benchmark. Black color shows a strong (close to 1) correlation coefficient and the white color shows a weak correlation between the configurations.

We can notice that the plots started to differentiate after 2500 instances. In other

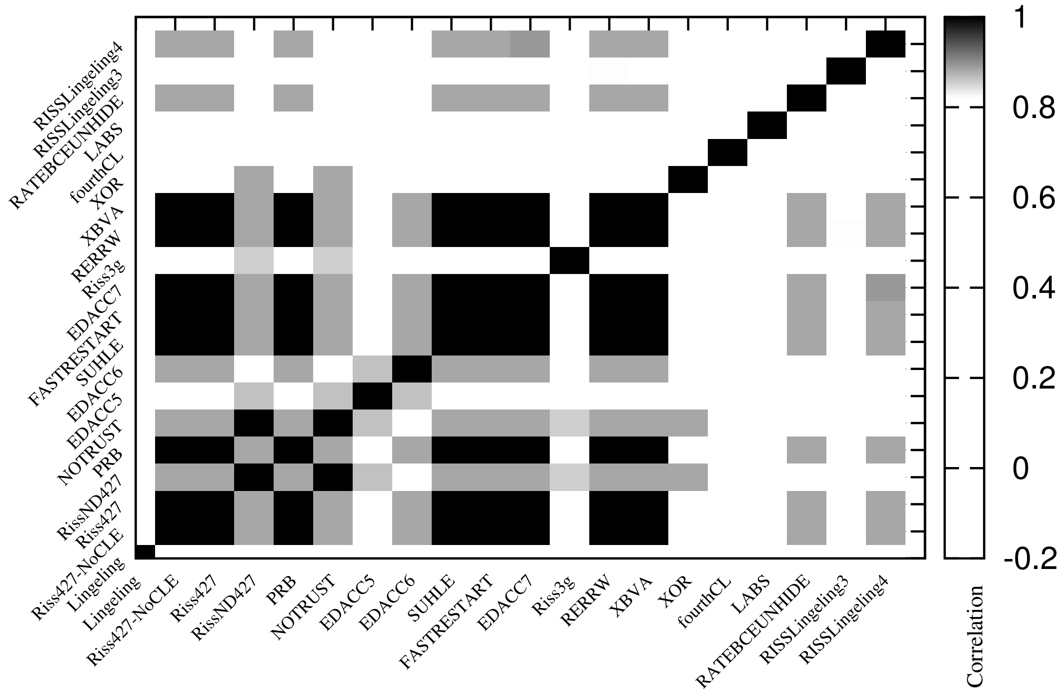

Figure 12: Correlation matrix for the runtimes of the configurations used together with LINGELING solver.

words, all the configurations were able to solve 2500 instances in less than 100 seconds. The performance shown in the figure was very similar for the configurations with more than 4300 solved instances, which explains the area of strongly correlated configurations on the correlation matrix. The LINGELING solver cactus plot showed a better performance than the configurations used for the RISS solver. The solving times of LINGELING solver showed no strong correlation with the solving times of the configurations of our portfolio, because LINGELING is based on a different search engine and uses simplification during the search process.

6.3 Prediction Models Performance

SAT solvers try to find the solution of instances with limited resources. Consequently, not all the instances in a benchmark can be solved. When the features computation and the algorithm selection are executed before calling the SAT solver, the overall runtime increases. For this reason, the estimated runtime is computed and then compared to the defined timeout in order to know if an instance can be solved by the prediction model.

The estimated runtime of the prediction models is considered to be the sum of the features extraction runtime, a constant (one second for this thesis) representing the algorithm selection runtime and the runtime of the selected configuration. The number of solved instances based on the estimated runtime is used to measure the performance of the algorithm selection models.

6.3.1 Performance on Novel Instances

The prediction models were trained on a percent of the instances (30 %, 50 % and 70 %) and then tested on the remaining instances. In order to have an idea of how stable were the obtained results, the experiments were repeated three times with a shuffled version of the instances and when the benchmark was divided in the training and testing dataset produced three different versions of training and testing datasets. Instances in the test dataset were not part of the training dataset. Therefore, the test dataset is purely composed of novel instances with respect to the training dataset.

For None features computation version all the class partitioning versions and machine learning models were tested. In Table 11 we can see the information of the best models in the test dataset. The percent of solved instances with respect to the number of instances solved by the virtual best solver (VBS) is shown for the best models of each machine learning model tested. Additionally, the results of the Lingeling solver and of the best configuration in the test dataset are shown.

For the same prediction model when the size of the training dataset increased the percent of solved instances fluctuated with variations smaller than one percent. The model IBK1 obtained better results than IBK in two thirds of the tested datasets. k-NN1 outperformed k-NN model also for two thirds of the tested datasets. On the other hand, RF prediction model always presented better results than RF1 model. It can be seen in the Table that none of the binary classification models performed better than the best configuration or than the Lingeling solver. Yet the results using random forest are better than the results obtained using k nearest neighbor for binary classification models. Only the k-NN and k-NN1 prediction models outperformed the best configuration (bold format) in the portfolio and in more than two thirds of the cases they also outperformed the results of the Lingeling solver (italic format).

The “relative” class partitioning criteria obtained the best results for the k-NN models and for random forest models the best class partitioning criteria was “complement”. For IBK and IBK1 models “complement” criteria obtained the best results for two thirds of the tested datasets and the “global” criteria obtained the best results for the other third of the tested datasets.

For None features computation version, k-NN1 was the best machine learning model and RF was the best model based on binary classification. Both models were then tested with all the features computation versions and all the class partitioning versions.

Training	Model		Solved %		
%	Version	ML Model	1	2	3
30 %	VBS		3304 instances		
	relative	k-NN1	***94.79***	***94.58***	***94.76***
	relative	k-NN	***94.92***	***94.25***	**94.34**
	complement	RF1	92.22	92.19	91.89
	complement	RF	92.40	92.28	92.07
	global	IBK	92.19	92.01	91.92
	complement	IBK1	92.31	92.13	91.86
	Lingeling		94.52	94.04	94.34
	Best Configuration		93.86	93.67	93.61
50 %	VBS		2360 instances		
	relative	k-NN1	***94.45***	***94.66***	**94.66**
	relative	k-NN	**94.36**	***94.62***	**94.49**
	complement	RF1	91.57	91.99	91.91
	complement	RF	92.08	92.16	92.16
	complement	IBK	91.57	92.03	91.95
	complement	IBK1	91.65	91.95	92.08
	Lingeling		94.36	93.64	94.66
	Best Configuration		93.18	93.64	94.03
70 %	VBS		1416 instances		
	relative	k-NN1	***94.14***	***95.20***	***94.56***
	relative	k-NN	***94.14***	**94.35**	***94.77***
	complement	RF1	91.95	92.23	92.51
	complement	RF	92.58	92.80	92.73
	complement	IBK	92.16	91.81	92.44
	global	IBK1	90.47	92.30	92.58
	Lingeling		94.00	94.42	94.07
	Best Configuration		93.08	94.07	93.86

Table 11: Best models results based on None features computation compared with the VBS, Lingeling solver and the best configuration in the portfolio.

Training %	Features	Model Version	ML Model	Solved % 1	2	3
30 %	VBS			3304 instances		
	XOR	relative	k-NN1	***95.10***	***94.73***	***94.95***
	NONE	complement	RF	92.40	92.28	92.07
	LINGELING			94.52	94.04	94.34
	Best Configuration			93.86	93.67	93.61
50 %	VBS			2360 instances		
	NOCGRESXOR	relative	k-NN1	**93.60**	***94.83***	***94.92***
	NOCGRESXOR	complement	RF	91.82	92.25	92.33
	LINGELING			94.36	93.64	94.66
	Best Configuration			93.18	93.64	94.03
70 %	VBS			1416 instances		
	NONE	relative	k-NN1	***94.14***	***95.20***	***94.56***
	RWH	complement	RF	92.73	92.87	93.15
	LINGELING			94.00	94.42	94.07
	Best Configuration			93.08	94.07	93.86

Table 12: Best models results compared with the VBS, LINGELING solver and the best configuration in the portfolio.

Table 12 shows the percent of solved instances of the best models compared to the performance of the LINGELING SAT solver and to the best configuration. The percent is based the number of solved instances of the virtual best solver (VBS).

None of the features computation versions in the best prediction models computes the clauses graph or the resolution graph. The results of RF prediction model remained worse than the results of the best configuration and LINGELING solver.

The best result obtained was with NONE features computation, "relative" class partitioning criteria and with k-NN1 machine learning model, solving for one of the testsets 95.20 % of the instances solved by the VBS in the testing dataset. The rest of the combinations of models obtained results bellow 95 %.

Figure 13 shows the cactus plot of best prediction model (prediction plot) tested on novel instances, along with the cactus plots of the LINGELING solver, the best and worst configuration on the test set, the virtual best solver (VBS) and virtual worst solver (VWS). The model in the figure is based on NONE features computation, "relative" class partitioning, k-NN1 machine learning model and was trained on 70 % of the instances.

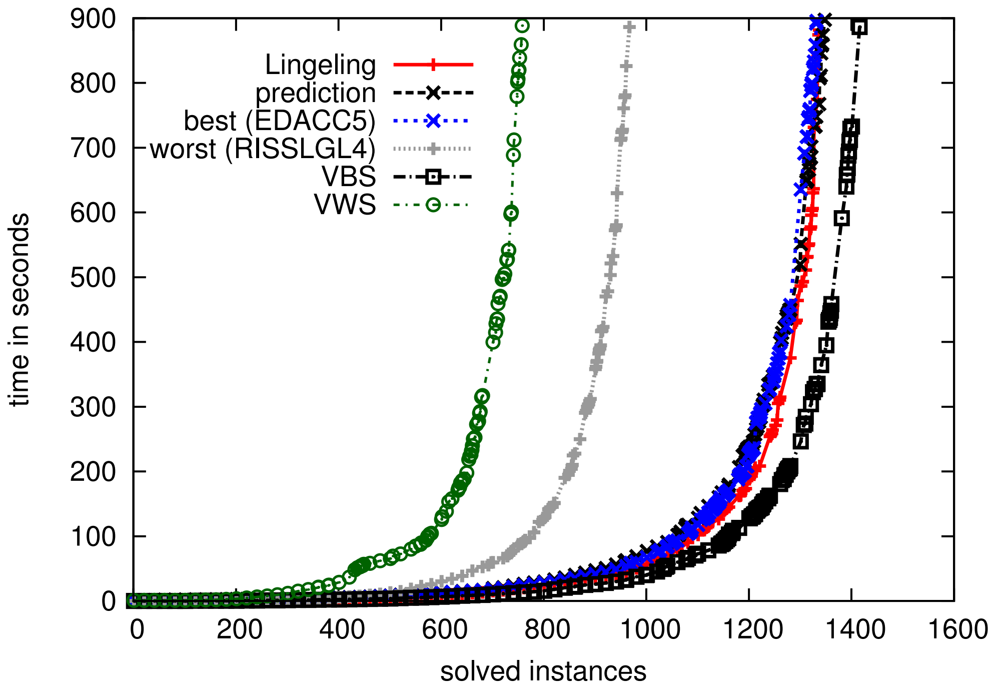

Figure 13: Cactus plot for the best prediction model tested on novel instances.

One can notice in the graph that the performance of the prediction model was very similar to the performance described by the plots of the best configuration and the Lingeling solver.

6.3.2 Overall Performance

The overall performance of a prediction model is also important, specially when there is a possibility that the instances for which the prediction is done are also part of the training dataset. The proposed models were also tested in the instances of the whole benchmark.

Table 13 shows the number of solved instances by the best prediction models on the whole benchmark that use k-NN1 and RF. The results of the Lingeling solver, of the best configuration and of the virtual best solver (VBS) are also shown.

One can notice that the best prediction models on the whole benchmark are the same best prediction models on novel instances with the exception of the k-NN1 best prediction model that uses Big features computation version instead of the Xor version. The size of the training dataset increases resulted in an increase on the number of solved instances by the models. The k-NN1 best prediction model trained on 30 % of the benchmark solved more instances than the RF model trained on 70 %. All the best prediction models outperformed the best configuration in the portfolio. All the

Training		Model		Solved		
%	Features	Version	ML Model	1	2	3
		VBS			4719	
30 %	BIG	relative	k-NN1	***4549***	***4539***	***4540***
	NONE	complement	RF	**4434**	**4421**	**4422**
50 %	NOCGRESXOR	relative	k-NN1	***4551***	***4594***	***4598***
	NOCGRESXOR	complement	RF	***4468***	***4483***	***4485***
70 %	NONE	relative	k-NN1	***4634***	***4648***	***4640***
	RWH	complement	RF	***4534***	***4531***	***4540***
		LINGELING			4444	
		Best Configuration			4408	

Table 13: Overall performance of the best prediction models compared with the VBS, LINGELING solver and the best configuration in the portfolio.

best prediction models that use k-NN1 together with the RF best prediction models trained on more than 30 % of the instances of the benchmark outperformed the LINGELING solver.

Figure 14 shows the *improvement percent* of the best prediction models. The improvement percent (*impr%*) of a model is computed based on the interval defined by the number of solved instances of the VBS and of the best configuration:

$$impr\%(model) = \frac{solved(model) - solved(best)}{solved(VBS) - solved(best)} \times 100\,\%.$$

The values are shown only for one of the randomized versions of training testing datasets of the benchmark. For RF best prediction model the greatest improvement percent obtained was 40 % which is less than the worse improvement percent obtained by k-NN1 best prediction models. In the figure when the training dataset size increases the improvement percent of the prediction models also increases. The highest improvement percent obtained was 77 % for the best k-NN1 prediction model trained on 70 % of the instances of the benchmark.

Figure 15 shows the cactus plot of best model (prediction plot) in the whole benchmark, along with the cactus plots of the LINGELING solver, the best and worst configuration on the test set, the virtual best solver (VBS) and virtual worst solver (VWS). The model in the figure is based on NONE features computation, “relative” class partitioning, k-NN1 machine learning model and was trained on 70 % of the instances.

The number of solved instances by the prediction model was greater than the number of instances solved by the best configuration from 200 seconds on and grater than the number of instances solved by LINGELING solver from 600 seconds on. One can

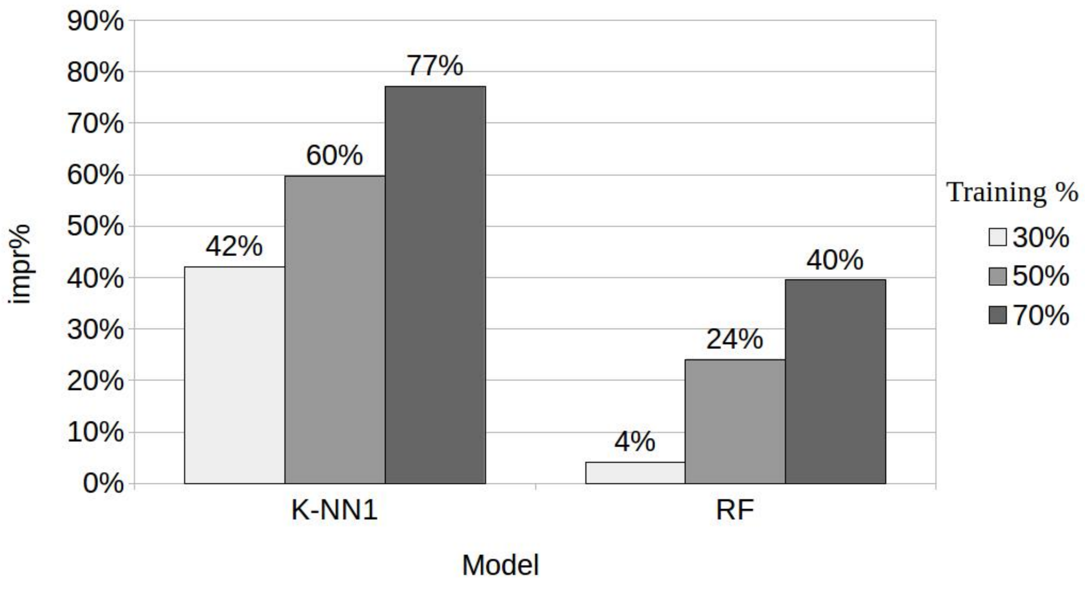

Figure 14: Improvement performance of the best prediction models.

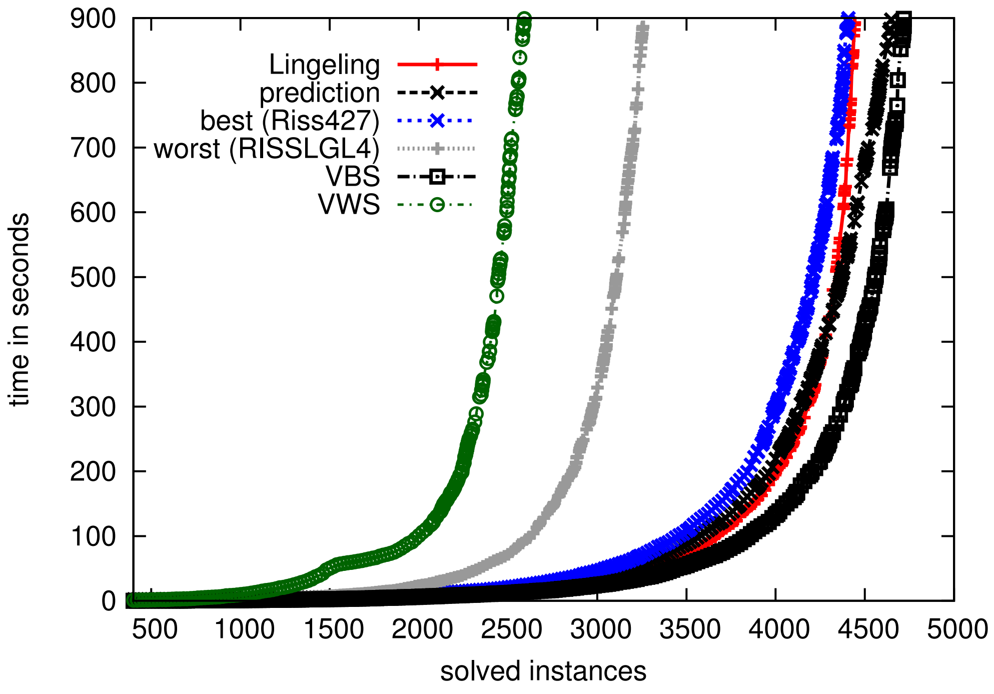

Figure 15: Cactus plot for one of the best models tested on novel instances.

Training %	Features	Model Version	ML Model	Solved 1	Solved 2	Solved 3
70%	VBS			4719		
	ALL	relative	k-NN1	***4596***	***4606***	***4603***
	BIG	relative	k-NN1	***4634***	***4647***	***4642***
	CLAUSE	relative	k-NN1	***4605***	***4612***	***4608***
	CONST	relative	k-NN1	***4637***	***4648***	***4641***
	DERIVATIVE	relative	k-NN1	***4631***	***4646***	***4644***
	NOCGRESXOR	relative	k-NN1	***4631***	***4610***	***4595***
	NONE	relative	k-NN1	***4634***	***4648***	***4640***
	RES	relative	k-NN1	***4623***	***4633***	***4624***
	RWH	complement	k-NN1	***4631***	***4638***	***4638***
	XOR	relative	k-NN1	***4625***	***4637***	***4637***
	ALL	complement	RF	***4496***	***4492***	***4495***
	BIG	complement	RF	***4530***	***4531***	***4533***
	CLAUSE	complement	RF	***4500***	***4492***	***4498***
	CONST	complement	RF	***4533***	***4527***	***4537***
	DERIVATIVE	complement	RF	***4532***	***4528***	***4533***
	NOCGRESXOR	complement	RF	***4532***	***4530***	***4537***
	NONE	complement	RF	***4532***	***4530***	***4534***
	RES	complement	RF	***4519***	***4519***	***4520***
	RWH	complement	RF	***4534***	***4531***	***4540***
	XOR	complement	RF	***4523***	***4519***	***4524***
	LINGELING			4444		
	Best Configuration			4408		

Table 14: Best models results for all features computation compared with the VBS, LINGELING solver and the best configuration in the portfolio.

see that the plot of the prediction got closer to the plot of the VBS than in the case of novel instances.

Table 14 shows the results of the best models based on k-NN1 and RF and trained in 70 % for each of the features computation versions. The results of the LINGELING solver, of the best configuration and of the virtual best solver (VBS) are also shown. One can clearly see that all the resulting best models solved more instances than the best configuration and than the LINGELING solver when tested in the whole benchmark. There were differences between the results of all different features computation, more noticeable when the computation of the clauses graph was included (ALL and CLAUSE versions). In spite of the variations observed by changing the features computation versions, all the best models that use k-NN1 outperform all the best models

Version	Description
BlackBox2-k-NN1	Machine learning model: k-NN1 Features computation version: None Training set: the complete benchmark. Class partitioning timeout: 900 seconds.
BlackBox2-RF	Machine learning model: RF Features computation version: Rwh Training set: the complete benchmark. Class partitioning timeout: 900 seconds.
BlackBox1	Machine learning model: RF Training set: a bigger benchmark that includes repeated instances. Class partitioning timeout: 3600 seconds.

Table 15: Description of the algorithm portfolios tested.

based on RF.

6.3.3 Empirical Performance

The empirical performance of an algorithm portfolio will be based on the number of solved instances considering the actual execution of the model rather than considering an estimate of the runtime. The algorithm portfolio is completely executed, including the execution of the Riss solver using the selected configuration.

In Table 15 the implemented portfolios are described. BlackBox1 [AM14b] algorithm portfolio is based on the best model obtained in [AM14a] that was trained for participating on the 2014 SAT competition. The configurations of the Riss solver used for BlackBox1 are different from the ones used in this thesis for BlackBox2-k-NN1 and BlackBox2-RF.

In the competition, BlackBox1 obtained the first place in the Crafted UNSAT track, second place in the Application SAT track and third place in the Application SAT+UNSAT track. The official timeout of the competition is 5000 seconds. The three algorithm portfolios were tested on the benchmark used in the 2014 SAT competition for timeouts of 900 seconds (Table 16) and 5000 seconds (Table 17). In Tables 16 and 17 the winning tracks are maked with the corresponding superindexes. The values that are greater than the BlackBox1 results are highlighted with bold style.

Table 16 shows that the BlackBox2 algorithms portfolios solved more instances than BlackBox1 for all the tracks of the Crafted benchmark and also for the Application UNSAT competition track. Furthermore, the total number of instances solved in the complete 2014 benchmark for the BlackBox2 portfolios was greater than the

Competition Track		BlackBox2-k-NN1	BlackBox2-RF	BlackBox1
Application	SAT[2]	68	67	76
	UNSAT	**89**	**90**	87
	SAT+UNSAT[3]	157	157	163
Crafted	SAT	**79**	**80**	69
	UNSAT[1]	**45**	**47**	39
	SAT+UNSAT	**124**	**127**	108
Totals		**281**	**284**	271

Table 16: Empirical performance on 2014 with 900 sec timeout.

Competition Track		BlackBox2-k-NN1	BlackBox2-RF	BlackBox1
Application	SAT[2]	87	88	108
	UNSAT	105	108	117
	SAT+UNSAT[3]	192	196	225
Crafted	SAT	**89**	**90**	88
	UNSAT[1]	**87**	**96**	79
	SAT+UNSAT	**176**	**186**	167
Totals		368	382	392

Table 17: Empirical performance on 2014 with 5000 sec timeout.

total for BlackBox1.

In the results shown on Table 17 for 5000 seconds timeout, the BlackBox1 portfolio solved more instances than the BlackBox2 portfolios in all the tracks of the Application benchmark. On the other hand, the results for BlackBox2 portfolios were still better than BlackBox1 results for all the tracks of the benchmark of Crafted instances. Nevertheless the greatest total of solved instances in the complete benchmark was achieved by BlackBox1.

The results of BlackBox2-RF and BlackBox2-k-NN1 were more similar than expected. The BlackBox2-RF portfolio solved only three more instances than BlackBox2-k-NN1 for a 900 seconds timeout. However, for 5000 seconds timeout the difference increased up to 14 instances supporting that the configurations selected by BlackBox2-RF were better.

7 Conclusion and Future Work

In this thesis the experiments were set out to study the performance on novel instances of different algorithm selection models based on machine learning techniques. Instance-specific Algorithm selection models allow the combination in a complementary way of different SAT solving techniques. Although the performance is upper bounded by the performance described by the virtual best solver, normally the best algorithm in the portfolio can be outperformed. In general, the cooperation will combine the strengths of the different algorithms allowing us a better usage of the available SAT solving algorithms. This thesis searched for the answer to the following questions:

1. What features computation versions have better properties to represent CNF instances for the used machine learning techniques?
2. Which machine learning prediction models provide the best performance on novel instances?
3. Which machine learning prediction models offer the best overall performance?

In Section 6 the empirical results were presented for all the prediction models tested.

The features computed provided very useful and non redundant information to represent CNF instances for the application of machine learning techniques. Furthermore, the computation was more efficient when the clauses and resolution graphs were not computed and the features remained with high values of information gain ratio.

The prediction models based on k-nearest neighbors and not in binary classifiers provided the best results for novel instances. They outperformed the best configuration and the Lingeling solver. The best model solved 95.20 % of the instances solved by the VBS which was only 1.13 % more instances than the best configuration. Therefore, there is still a lot of room for improvement. Models based on the same machine learning technique with different parameters did not differentiate too much.

The best prediction models on novel instances also resulted to be the best prediction models when they were tested on the whole benchmark, with the exception of the best model based on k-NN1, trained on 30 % of the instances. All the best models outperformed the best configuration and Lingeling solver on the whole benchmark. The best improvement percent obtained for k-NN1 model was 77 %, almost twice the best improvement percent of RF model, which was only 40 %. For the overall performance, the increases in the size of the training dataset improved the performance of the models but for the performance on novel instances it was not the case.

Still there is room for improvement in the the overall performance. Although the parameters of the prediction models played an important role, it is more important to focus on the structure of the prediction model and the machine learning techniques used. The configurations used in this thesis come from the SAT Competition 2013, the Configurable SAT Solver Challenge (CSSC) 2013 and some additional ones tuned for the 2014 SAT competition, and were not defined specifically to work in an algorithms portfolio cooperating between them.

The empirical performance of the implemented portfolios still needs to be improved in order to achieve a performance that is closer to the estimated one. Nevertheless, the portfolios BlackBox2 performed better than BlackBox1 for the 2014 SAT competition track in which it obtained the gold medal.

As future work, some interesting ideas can be considered.

First, a partition of the instances in the benchmark can be defined according to how “easy” or “difficult” are they solved by the algorithms in the portfolio. Such partition can be used to build different algorithm selection procedures for easy and difficult instances.

One can also build a classifier to predict whether the instance is satisfiable or not. Then, use that information as another feature.

Additionally, the runtime of the algorithms can participate more in the algorithm selection procedure to try to select faster algorithms and not just any algorithm that solves the instance before the timeout.

Moreover, we can use the proposed features and perform *instance-specific algorithm configuration* (ISAC) [MS12]. ISAC can optimize the configurations of the Riss solver according to the information provided by the features. The resulting configurations would be better suited for using them in algorithm portfolios.

In this thesis, the best prediction model combined a very simple features computation (None version) and a selection algorithm based on k-nearest neighbor method and the “relative” class partitioning. Moreover, no binary classifier was used in the best model.

Acknowledgments This thesis would not be possible without the help of many organizations and people. The author would like to thank the ZIH of TU Dresden for providing the computational resources to produce the experimental data for the empirical evaluation and also the European Master's Program in Computational Logic (EMCL) for their support and education. Gratitude to the Erasmus Mundus Programme for their economical help. Special thanks to Norbert Manthey for his efficient work as a supervisor and his contagious motivation in the field. At last but not the least, thanks to the Friends who gave emotional and scientific support and also to the Family for believing.

References

[AM14a] Enrique Matos Alfonso and Norbert Manthey. New CNF features and formula classification. In Daniel Le Berre, editor, *POS-14*, volume 27 of *EPiC Series*, pages 57–71. EasyChair, 2014.

[AM14b] Enrique Matos Alfonso and Norbert Manthey. Riss 4.27 blackbox. *Proceedings of SAT Competition 2014*, pages 68–69, 2014.

[ARMS02] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult SAT instances in the presence of symmetry. pages 731–736, New York, NY, USA, 2002. ACM.

[Bie13] Armin Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. *Proceedings of SAT Competition 2013*, pages 51–52, 2013.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[CWZ11] Xiang Chen, Minghui Wang, and Heping Zhang. The use of classification trees for bioinformatics. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 1(1):55–63, 2011.

[Gab] Oliver Gableske. https://www.gableske.net/dimetheus. `https://www.gableske.net/dimetheus`. Accessed: 2014-04-14.

[GHM$^+$12] P. Großmann, S. Hölldobler, N. Manthey, K. Nachtigall, J. Opitz, and P. Steinke. Solving periodic event scheduling problems with SAT. volume 7345, pages 166–175, Heidelberg, 2012. Springer.

[Her06] P. Herwig. Using graphs to get a better insight into satisfiability problems. Master's thesis, Delft University of Technology, Department of Electrical Engineering, Mathematics and Computer Science., 2006.

[HR76] Laurent Hyafil and Ronald L Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[HW10] Marijn Heule and Toby Walsh. Symmetry in solutions. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[KMS$^+$11] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In Jimmy Ho-Man Lee, editor, *CP*, volume 6876 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.

[KSHK07] D. Kaiss, M. Skaba, Z. Hanna, and Z. Khasidashvili. Industrial strength

SAT-based alignability algorithm for hardware equivalence verification. pages 20–26, Washington, 2007. IEEE Computer Society.

[Kul99] O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96–97(0):149 – 176, 1999.

[LJN10] Tero Laitinen, Tommi A. Junttila, and Ilkka Niemelä. Extending clause learning DPLL with parity reasoning. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 21–26. IOS Press, 2010.

[Man14] Norbert Manthey. Riss 4.27. *Proceedings of SAT Competition 2014*, pages 65–67, 2014.

[MRT12] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning.* The MIT Press, 2012.

[MS12] Yuri Malitsky and Meinolf Sellmann. Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson, editors, *CPAIOR*, volume 7298 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2012.

[NLBH$^+$04] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 438–452. Springer, 2004.

[NMJ09] Mladen Nikolić, Filip Marić, and Predrag Janičić. Instance-Based Selection of Policies for SAT Solvers. pages 326–340. 2009.

[NMJ11] Mladen Nikolic, Filip Maric, and Predrag Janicic. Simple algorithm portfolio for SAT. *CoRR*, abs/1107.0268, 2011.

[Rou12] Olivier Roussel. Description of ppfolio 2012. In *Proc. SAT Challenge 2012; Solver and Benchmark Descriptions*, page 46. Univ. of Helsinki, 2012.

[VGLBB$^+$05] Allen Van Gelder, Daniel Le Berre, Armin Biere, Oliver Kullmann, and Laurent Simon. Purse-based scoring for comparison of exponential-time programs. Poster, 2005.

[WFH11] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, Amsterdam, 3 edition, 2011.

[XHHLb07] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-brown. Satzilla-07: The design and analysis of an algorithm portfolio for sat. In *In Proc. of CP-07*, pages 712–727, 2007.

[XHHLB12] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 228–241, Berlin, Heidelberg, 2012. Springer-Verlag.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 10. September 2014

Enrique Matos Alfonso