New CNF Features and Formula Classification

Enrique Matos Alfonso*and Norbert Manthey

Knowledge Representation and Reasoning Group
Technische Universität Dresden
Dresden, Germany

Abstract

In this paper we first present three new features for classifying CNF formulas. These features are based on the structural information of the formula and consider AND-gates as well as exactly-one constraints. Next, we use these features to construct a machine learning approach to select a SAT solver configuration for CNF formulas with random decision forests. Based on this classification task we can show that our new features are useful compared to existing features. Since the computation time for these features is small, the constructed classifier improves the performance of the SAT solvers on application and hand crafted benchmarks. On the other hand, the comparison shows that the set of new features also results in a better classification.

1 Introduction

Most modern SAT solvers, for example the ones that are used in international competitions [6, 7], have a good default configuration, which solve a high number of formulas on these benchmarks. Hence, many applications use these systems, for example in scheduling [15]. Still, these SAT solvers can provide more performance when tuned for a set of benchmarks, as shown recently in the Configurable SAT Solver Challenge 2013[1]. There exists methods to automatically tune a highly configurable SAT solver, such as ParamILS [20], GGA [4], or SMAC [19]. Given a benchmark that contains a set of problem formulas, one of these tools can be used to find a configuration of the used SAT solver to solve the benchmark best. Such a configuration requires many calls of the SAT solver, and a huge amount of computational resources. For an initial check whether using a SAT solver for a new problem is promising, we propose a simpler mechanism by classifying formulas and then selecting a promising configuration.

For selecting such a solver configuration, or a solver, there already exists tools. SATzilla [31] is such a tool that selects a SAT solver out of a portfolio of solvers based on a set of features, and internally learns a model to predict the run time of all the solvers in the portfolio. Then, SATzilla chooses the solver with the smallest predicted run time. More features have been proposed in [14], however, no empirical results on these features has been presented. The method *instance specific algorithm configuration*(ISAC) has been proposed by Kadioglu et al. [21, 23], and is based on the features that have been proposed by SATzilla, excluding run time based features. In ISAC the training set of instances is split into clusters based on the normalized features, and then for each cluster a good configuration is learned with GGA. To solve a a new instance, ISAC extracts its features, normalizes them, and then selects the cluster that is most similar to the instance, or if no such cluster is available in the set of known clusters a fall-back configuration is used. Again, the learning phase is done offline, so that a use of the ISAC method can use only the selection phase of the tool. For ISAC feature filtering has been proposed, to select only features that improve the creation of good instance clusters, and hence

*The author was supported by the European Master's Program in Computational Logic (EMCL).

[1] `http://www.cs.ubc.ca/labs/beta/Projects/CSSC2013/`

also selecting a good cluster. Further related work are sequential and parallel portfolio solvers, most prominently PFOLIO [29], which simply execute different SAT solvers in parallel, even with only a single computation unit.

In this paper we present three new CNF features, as well as a new classification model for SAT solver configurations. The computation of these features is light-weighted, so that they can be also extracted on nowadays large application formulas. On recent SAT benchmarks we show that the computation of the features can be done much faster than for the features of widely used SATZILLA. The quality of the feature is evaluated based on a formula classification, where we select a solver configuration based on the input formula. For a pre-defined set of configurations, we compute the run time for a large benchmark of application and crafted formulas. Then, we show that a very simple classification model, namely random decision forests, results in a good classification. When we train our classifier on the available competition instances of 2011 and 2012, and evaluate its performance on the 600 instances of 2013, then the best configuration can solve 229 instances, the classifier variant can solve 274 instances. When replacing our features by the features computed by SATZILLA the number of solved instances drop to 239. A reason for this smaller number is the time that is required to extract the features. Therefore, we try to compute all features with a very small run time.

The paper is structured as follows: First we present the state-of-the-art features, and next propose the new CNF features as well as the algorithms for their extraction in Section 2. Next, in Section 3, we discuss different possibilities for objective functions and how features can be used to classify CNF formulas. Then, Section 4 evaluates this classification. The conclusion of the paper is presented in Section 5.

2 Feature Extraction of CNF Formulas

The set of Boolean variable V is the set of natural numbers $V = \mathbb{N}^+$. A literal x is a positive Boolean variable x, or a negative Boolean variable $\overline{x}$. A clause C is a set of literals, and a formula F in conjunctive normal form (CNF) is a multi-set of clauses, because duplicate clauses can occur. In this paper we use the following notation: for a formula F and the literal x, F_x represents the set $\{C \mid C \in F, x \in C\}$. Furthermore, $\mathsf{vars}(F)$ represents the set of variables occurring in the formula. We assume the reader to be familiar with the basic concepts of propositional logic and SAT solving. More details about the used concepts can be found in [9].

In the following we present features that are already used in SATZILLA [31], before introducing new features. Some features in SATZILLA are based on the run time of an algorithm. We improve the concept of run time in seconds by a hardware-independent measurement: instead of measuring time, we maintain counters that are increased during the computation. After the computation, the value of the counter correlates to the run time of the algorithm, however, the value of the counter is independent of the used architecture. Hence, the dynamic feature run time is turned into a static feature.

Before we extract features from a formula, we apply *unit propagation* until a fixed point is reached. This way, the computational overhead is reduced, because all satisfied clauses, and all falsified literals are removed from the formula. Finally, the unit clauses are added back to the formula, to still work on the semantically equivalent formula. We do not apply more complex simplification techniques due to two reasons: (1) the different solver configurations use different simplification techniques and (2) for large formulas simplification techniques can consume a lot of time.

Table 1: Graph structures considered for features computation of a formula F with $|F| = m$, and n variables. The indexes i and j are elements of the corresponding vertexes sets.

Graphs	Vertices	Edges	Weights	Considered sequences
CV^+ (CV^-)	$B = \{1, \ldots, m\}$ $W = \{1, \ldots, n\}$	$\langle i, j\rangle$ and $\langle j, i\rangle$ iff $j \in C_i$ ($-j \in C_i$)	-	degrees
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}	degrees weights
Clauses	$V = \{1, \ldots, m\}$	$\langle i, j\rangle$ iff $C_j \cap C_i \neq \emptyset$	$\lvert C_j \cap C_i\rvert$	degrees weights
Resolution	$V = \{1, \ldots, m\}$	$\langle i, j\rangle$ iff $\lvert C_i \cap \overline{C_j}\rvert = 1$	$2^{-(\lvert C_i \cup C_j\rvert - 2)}$	degrees weights

2.0.1 Graphs and Sequences

Most of the following features are based on *graphs* $G = (V, E)$, or *bipartite graphs* $G = (W, B, E)$, where V, W and B are vertices, and the set of edges is $E \subseteq V \times V$ or $E \subseteq (W \times B) \cup (B \times W)$, respectively. The edges in a graph can be *weighted* by a function $w : E \to \mathbb{R}$. Furthermore, for each vertex v, the *degree* is defined as $\deg(v) = |\{w \mid \langle v, w\rangle \in E\}|$.

Given a graph, the degrees of all vertices v can be collected as a sequence. A degree sequence of values $S = \langle x_1, \ldots, x_n\rangle$ is characterized with common statistical measures. First, a preprocessing step is used, which removes all elements $x_i = 0$ from the sequence S, and that sorts the elements in S, to obtain an invariance against shuffled formula, similarly to rotation invariance in shape recognition. The number of zero elements $x_i = 0$ is used as a statistical measurement of a sequence S.

Then, the following properties of such a normalized sequence are considered as features: the mean $mean(S)$, the standard deviation, minimum, maximum,and mode, which is the smallest number that appears most frequently in the sequence. Furthermore, the quantiles for 25 %, 50 % and 75 % are used, as well as the values rate $rate(S) = \frac{1}{n}|\{x_1, \ldots, x_n\}|$ and the entropy $entropy(S) = \ln(n) - \frac{1}{n}\sum c_j \ln(c_j)$, where each c_j represents the number of occurrences of for a value $x_i = j$ in the sequence. Finally, for a sorted sequence, also its derivation $S' = \langle x_2 - x_1, \ldots, x_n - x_{n-1}\rangle$ is computed, and all the above measures are used as features of this new sequence as well. This way, a more global view on the values in the sequence is represented, for example whether a sequence S increases slightly, or whether there are only a few high increases.

2.1 Graph Features

The following graphs are then computed for the CNF formula: (i) The *Clause-Variable graph* (CV^+ for positive literals, CV^- for negative literals) that connects literals with the clauses in which they appear. (ii) The *Variables graph* in which two variables are connected when they appear in the same clause without considering the polarity. (iii) The *Clause graph* connects clauses that share literals and (iv) the *Resolution graph* connects clauses when they produce a non-tautological resolvent. Table 1 defines this structural information more formally and also presents the considered sequences.

For the literal graph the sequence $deg^{\pm}(i) = \frac{max(deg_{CV^+}(i), deg_{CV^-}(i))}{deg_{CV^+}(i) + deg_{CV^-}(i)}$ is used, which represents the major polarity of each clause. For some of the graphs weighted variants are used, and

hence the calculation of the weights is given. For the variable and the resolution graph, the weight reflects how strong the search space of the problem is influenced by the corresponding variable or resolvent.

2.2 Features Aiming at Formula Structure

CNF formulas of applications, or difficult hand crafted CNF formulas contain a structure [2, 3], that is assumed to be exploited by modern SAT solvers [9]. The following sections present features that are motivated by this structure. This way, we try to distinguish better between application formula families and crafted formula families.

2.2.1 Binary Implication Graph

A simple graph of a formula is the *binary implication graph* $\text{BIG} = (V, E)$, which contains all literals as vertices, $V = \mathsf{lits}(F)$, and all edges in the graph correspond to the binary clauses in the current formula: $E = \{\langle a, b\rangle, \langle \overline{b}, \overline{a}\rangle \mid \{\overline{a}, b\} \in F\}$. For this graph, a sequence with the degree of each node is used.

2.2.2 Naive Encoded Constraints

CNF formulas come from different applications, for example hardware verification [22], or scheduling [15]. Whereas verification problems contain gates, for example AND-gates, scheduling instances usually contain EXACTLY-ONE-constraints. A usual circuit is encoded based on the Tseitin encoding [30] with AND-gates $x \leftrightarrow (a_1 \wedge \cdots \wedge a_k)$, that results in $k+1$ clauses. Due to the Plaisted-Greenbaum encoding of circuits [28], not all these $k+1$ clauses are required of each gate to represent the full circuit in CNF. Then, the partial representation $x \rightarrow (a_1 \wedge \cdots \wedge a_k)$, or $x \leftarrow (a_1 \wedge \cdots \wedge a_k)$ is only present. Although there exists research on extracting more complex high level constraints, we do not focus on those constraints here, because we want to keep the computational effort for the feature computation as small as possible.

To extract these AND-gates, all three patterns need to be detected. Furthermore, for scheduling instances, the constraint EXACTLY-ONE can also be found, because its clause pattern is very similar:

$$(x \rightarrow (\overline{a_1} \wedge \cdots \wedge \overline{a_k})), \wedge \cdots \wedge (a_k \rightarrow (\overline{a_1} \wedge \cdots \wedge \overline{x}).$$

The undirected graphs AND-gates(V, E_{AND}), BLOCKEDAND-gates(V, E_{BAND}) and EXACTLY-ONE-gates(V, E_{EXO}) can be extracted from the formula with a single method, which is given in Figure 1. The sets of edges for the corresponding graphs present for a pair of literals whether this pair is present in a gate of the above types. The algorithm first computes the binary implication graph (line 1), and initializes the graphs (line 2). Next, for each clause C it is tested, whether this clause is part of an exactly one constraint, or an AND-gate (line 3). First, for all literals $l \in C$ it is tested whether all remaining literals l' of the clause are implied by l negatively, $l \rightarrow \overline{l'}$ (lines 5–8). If not, then C cannot form an exactly one constraint, and the literal l cannot build an AND-gate with the clause C (line 8). The literal l and the clause C can still build a blocked AND-gate, namely if all required binary clauses could be added by blocked clause addition [24]. An easy check for this situation is to test whether the clause C is the only clause that contains the literal l in the formula. If this check succeeds for all literals (lines 12–13), then C and l build a blocked AND-gate, and the corresponding edges are added to E_{BAND} (lines 14–15). If the current clause C and the literal l build an AND-gate, then the edges are added to E_{AND} (lines 8–9), and the blocked gate is not added (line 11). Finally, if C represents an exactly one constraint, the clique of edges is added (lines 16–19). Given the

```
    ExtractConstraintFeatures (CNF formula F)
 1  let BIG be the binary implication graph for F
 2  let E_AND = ∅, E_BAND = ∅, E_EXO = ∅             // initialize edges
 3  for C ∈ F, |C| > 2                               // for all large clauses
 4    let exo = ⊤                                    // might be an exactly one
 5    for l ∈ C
 6      for l' ∈ C \ {l}                             // check for clique
 7        if ¬l' ∉ BIG(l) then                       // l implies ¬l'
 8          let exo = ⊥, goto 12                     // no AND-gate
 9      for l' ∈ C \ {l}                             // found AND-gate
10        let E_AND = E_AND ∪ {(l', ¬l)}
11      goto 5                                       // jump over blocked AND-gate
12      for l' ∈ C \ {l}                             // check blocked AND-gate
13        if {C | C ∈ F, l ∈ C} > 3 then goto 3      // not obviously blocked
14      for l' ∈ C \ {l}                             // found blocked AND-gate
15        let E_BAND = E_BAND ∪ {(l', ¬l)}
16    if exo = ⊤ then                                // add E_EXO clique
17      for l ∈ C
18        for l' ∈ C \ {l}
19          let E_EXO = E_EXO ∪ {(l', l)}
```

Figure 1: Extracting AND-gates, blocked AND-gates and exactly one constraints from a formula.

edges for the three graphs, weighted variants are added for the AND graph, and the blocked AND graph, where each edge that is added (line 10, 15) is weighted with $2^{-|C|}$, so that this weight represents how strong the clause constrains the search space.

2.2.3 Symmetries

As reported for example in [1], problems originating from routing can also contain symmetries. Similarly, scheduling instances can contain many symmetries [18]. Hence, a feature is added, which reflects how symmetric an instance might be, which can be calculated iteratively. The underlying idea for the features originates in the coloring scheme of [1] to detect symmetries. For each variable v and each clause C, two measures depending on the iteration i are used: $s_v(i)$, which corresponds to the color, and $s_C(i)$. For the initial iteration, both literal and clause measures are set to 1. Then, they are calculated as follows: $s_C(i) = \sum_{v \in C} s_v(i-1), i > 0$ and based on the clause measures $s_C(i)$, the variable measures can be calculated: $s_v(i) = \sum_{C \in F_v} s_C(i)$. Following the coloring scheme in [1], different numbers $s_v(i)$ for two variables v and w mean that these two variables are not in the same symmetry group. A sequence for each iteration i is build based on the different values for variables $s_v(i)$, adding the number of occurrences for each value. Hence, for the initial iteration a sequence with one value, namely the number of variables, is added. For our feature computation we use the sequences after the first, second and third iteration. Since only sums are used, this scheme is only an approximation so that the generated symmetry classes cannot be used for symmetry breaking.

2.2.4 Recursive Weight Heuristic

Another well known feature for literals of a formula is the *recursive weight heuristic* (RWH). Initially, this heuristic was proposed for solving random 3-SAT formulas with a look-ahead solver [27]. Then, RWH has been generalized to formulas with arbitrary clauses [5]. The heuristic provides a score $h_i(x, F)$ for each literal x that represents the tendency whether x is present in a model of the formula F. By increasing the number of iterations, the precision of the tendency can be increased. This score is initialized to $h_0(x, F) = 0$ for all literals. For each iteration i, a scale factor $\mu_i(F)$ is added, which represents the average of the current iteration: $\mu_i(F) = \frac{1}{2\,|\mathsf{vars}(F)|} \sum_{x \in \mathsf{vars}(F)} (h_i(x, F) + h_i(\overline{x}, F))$. The value $h_{i+1}(x, F)$ represents the tendency of x being part of the model, by testing for each clause C with $x \in C$ how likely the remaining literals $l \in (C \setminus x)$ will be falsified by the model. $h_{i+1}(x, F) = \sum_{C \in F_x} \left(\gamma^{k-|C|} \mu_i(F)^{|C|-1} \prod_{l \in (C \setminus \{x\})} h_i(\bar{l}, F) \right)$, The values of each iteration are weighted with the mean of the previous iteration $\mu_i(F)$, and is furthermore weighted by a factor γ, which is set to $\gamma = 5$, both in the literature and our implementation. For the iterations $i = 1$, $i = 2$ and $i = 3$ the values of $h_i(l, F)$ are used as a sequence as formula features.

2.2.5 Implementation Details

During the computation of a feature, for example the resolution graph, the graph is built while iterating over the clauses of the formula. Let each clause C_i of the formula be represented by its index i. Then, a required property of the graph is that each index appears at most once in the adjacency list of another index. Hence, during the construction of the graph, when the index i should be added to the adjacency list of index j, then first a check needs to be performed so that i is not added twice. For fast random accesses and a compact representation, an array-based data structure is used. Then, the inserting procedure can result in a quadratic algorithm, because for each element i, the whole list has to be tested. Even if the list is sorted, the procedure remains quadratic, because in the array, after adding each element, needs to remain sorted. In the worst case, the inserted element is always the smallest element. Then, all remaining elements need to be pushed by one position.

To avoid the quadratic overhead, we do not check for the presence of the index i, but simply add i to the adjacency list. After the whole graph has been constructed, and before its features are computed, each adjacency list is sorted, and then from the sorted list duplicates can be removed in linear time. A drawback of this approach is that during constructing the graph much more memory is consumed.

3 Instance Classification

A machine learning approach is used to select a configuration for a given instance. Such an approach is visualized in Figure 2. First, the features of the instance are extracted. Next, the features are given to a classifier, which selects a configuration that is supposed to work well on the given formula. The aim of the classification is to identify good configurations for a given formula. Here, many questions have to be answered: (i) Which configuration should be available as candidates? (ii) What happens if one of the components fails, for example due to exhausting resources. (iii) Which features should be used? (iv) How should the classification procedure work?

An answer to the question (i) is difficult: if the classification algorithm always selects the best configuration, then many very diverse configurations should be available, so that a range

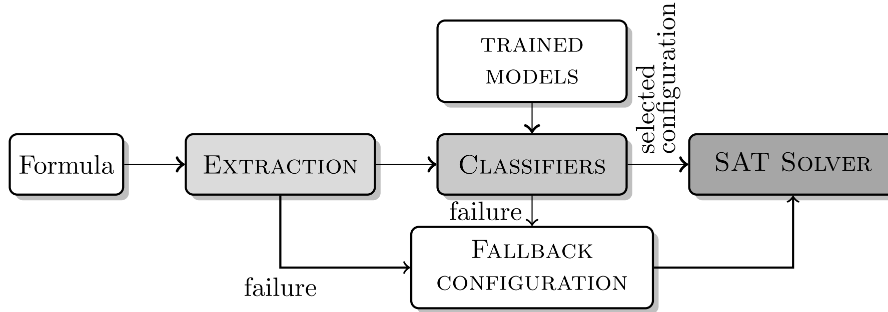


Figure 2: Classification of a formula given a SAT solver with many configurations.

of different formulas can be covered. In available tools, like ISAC, classification is interleaved with configuration optimization. This way, new configurations are created that have a high performance on a certain class of benchmarks.

The illustration already answers question (ii): If the feature extraction or the classification produce a failure, which happens for example by reaching resource limits, then a *fallback configuration* is used. For the computation of the features a trade off has to be made. Usually, the more features are available, the better the classification result, which can be measured in the prediction precision. The run time of the features computation has also to be taken into account. If computing the feature already consumes more time than there is available for solving the formula, then such an expensive feature should not be considered. There might also exist feature that do not help to classify a formula. Depending on the used classification algorithm, which might limit the number of used features, such features might be dropped to be able to consider better choices.

Finally, given the configurations $\{C_1, \ldots, C_j, C_k, \ldots, C_n\}$ and the features, the classifier needs to be trained on a set of formulas $\{F_1, \ldots, F_i, \ldots, F_m\}$ so that given a new formula a good configuration can be selected. First, for each pair of a formula F_i from the training set and each configuration C_j a decision has to be made whether C_j should be selected for solving the formula F_i. We label C_j for F_i with `good`, if C_j should be selected for F_i. Otherwise, the label `bad` is assigned. Now, given a certain resource limit, several schemes to assign the label `good` are possible. For this paper we focus on the time resource, denoted as `timeout`, for which we can define the labelling. $Time_f(F_i)$ represents the time used to compute the features of F_i, $time_{C_j}(F_i)$ is the time in which configuration C_j solves the instance F_i, b is a constant (5 seconds for our experiments) estimating how long the execution of the classification model for an instance can last. For the labellings Relative and Complement we define C_k as the fallback configuration. Then, the three labellings are:

Name	**Condition for `good`**
Global	$time_{C_j}(F_i) + time_f(F_i) + b < timeout$
Relative	$time_{C_j}(F_i) + time_f(F_i) + b$ $< \min(timeout, time_{C_k}(F_i) + time_f(F_i) + b)$
Complement	$time_{C_j}(F_i) + time_f(F_i) + b < timeout$ $\wedge\ (j = k\ \vee\ label_{C_k}(F_i) = bad)$

Then, based on the given labelling schemes, a machine learning algorithm can be selected that is trained on training data and afterwards used for classifying new formulas. Possible algorithms are *neural networks* [16], *support vector machines* [13][12] or *random decision forests* [10]. Furthermore, the classifier can be set up as a multi-class classifier to select one configuration, or

for each configuration there is a classifier that predicts whether this configuration should be selected for the current formula. In the latter case, the prediction with the highest likelihood is chosen.

3.1 Design Decisions – A First Classification Prototype

The fallback configuration is the configuration that can solve most of the instances of the benchmark. The resource limit for extracting the features as well as classifying is 125 CPU seconds and 6.8 GB of memory. The overall memory limit is 7 GB. As features we use all features of our tool except the clause graph and the resolution graph features (because their computation is too expensive, see Section 4.1). The configurations used for the evaluated classifier have been created during the *Configurable SAT Solver Challenge 2013* for the SAT solver Riss. To this set we add the default configuration of the solver. Hence, we do not tune the configurations on the benchmark, but use existing configurations only. As classification algorithm we use random decision forests, and use the implementation provided in the tool Weka [17]. Based on preliminary tests, we set up the classifier with the following settings: The configurations are labeled with `good` according to the Complement schema, selecting the default configuration as soon as the current formula is not very similar to known formulas. For the labelling procedure, a feature extraction timeout of 125 seconds is applied. Then for each configuration a random decision forest is trained, where the number of trees in the forest, as well as the depth of all trees is optimized for each configuration, using a local search algorithm and the training data. Starting with n trees and a depth of m, we perform a Hill Climbing algorithm [11] that attempts to iteratively improve m and n till the accuracy of the classifier can no longer be improved or till we reach a limit of 3 iterations. The number of trees value is changed by adding $\delta t \in \{-4, -2, 0, 2, 4\}$ and the depth by adding $\delta t \in \{-10, -5, 0, 5, 10\}$ where $\langle \delta t, \delta d \rangle \neq \langle 0, 0 \rangle$.

When a new formula is classified, then for each configuration a probability of being labeled with `good` is returned. This probability is calculated as the number of trees in this random decision forest that classify the formula as `good` for the configuration and normalizing this number by the number of trees in the forest. The configuration with the highest probability is selected. For tie-breaking, the configuration that can solve more formulas on the training data is selected if two configurations have the same probability.

4 Empirical Evaluation

For the empirical evaluation we use the SAT solver Riss [25], which is an extension of Glucose 2.2 mainly by incorporating the CNF simplifier Coprocessor [26]. The evaluation of the techniques is performed on all industrial instances and all crafted instances that have been used in the SAT Competitions and Challenge 2011 to 2013, including unselected instances. The full set contains 3213 instances.[2] The CPU time limit is set to 900 seconds and the memory limit is 7 GB. The used cluster uses AMD Opteron 6274 CPUs with 16 cores and 2 MB level 2 cache that is shared by two cores. The experiment uses one out of four cores for a SAT solver to stabilize the results.

[2] Duplicates benchmarks are kept. The source code and all used data is available at `tools.computational-logic.org/content/riss/blackbox.tar.gz`.

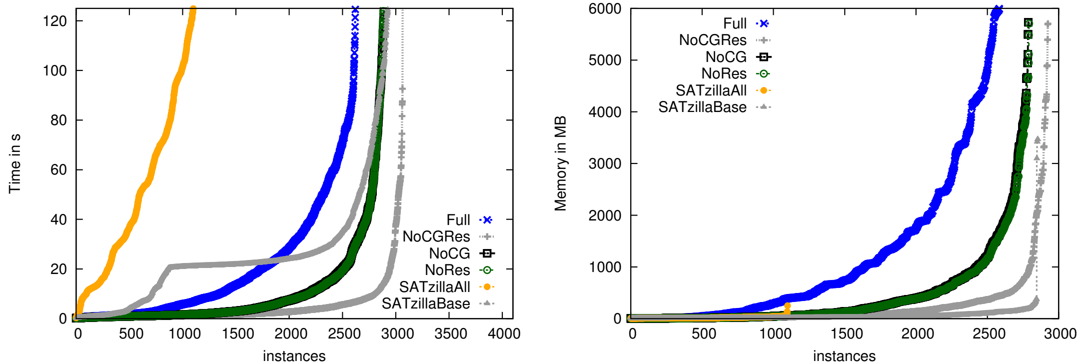


Figure 3: Run time distribution and memory consumption for the different feature computations.

4.1 Feature Computation

First, we compare the computation of our feature set to the widely used feature extraction algorithm of SATzilla. For SATzilla, the available feature sets All and Base are computed. We implemented our feature extraction into the SAT solver Riss. With this implementation we extracted the feature sets Full, which computes all the features including the clause graph and the resolution graph. Furthermore, we compute NoCG, which does not compute the clause graph features, NoRes, which does not compute the resolution graph and NoCGRes, which does not compute the clause graph features, nor the resolution graph features.

Figure 3 shows the distribution of the run time of the feature computation, as well as their memory consumption. The run time for SATzillaAll is very high, so that this feature set is too expensive for being used online. Next, the feature computation for SATzillaBase shows a plateau at 20 seconds, but can compute features for 2926 instances. Computing all features with Full can solve less instances, mainly due to the memory limitation. Since Full constructs all graphs, its memory consumption is very high. When either the clause graph or the resolution graph is not computed, then features for more instances can be computed. The least run time and memory consumption is obtained when the two graphs are not used in NoCGRes, which results in features for more instances. The comparison of run time and memory shows the effect of the implementation of our feature extraction (Section 2.2.5): by using more memory, the run time of the algorithm can be decreased significantly.

In machine learning, the *information gain* is defined as the change in the entropy of the information from a prior state to a state that takes some information as given. The information gain ration is then the ration of the information gain and the intrinsic value. To compute how important are the features defined in this paper, we calculate the *information gain ratio* with respect to each to the classes defined by each of the configurations and then build average. According to the values obtained we defined the value 0.02 as the threshold that defines the most important features. Table 2 shows the features groups that have computed features and give the number of important features per group. All the newly introduced features groups have features of high importance for the classification. Furthermore, more features with a high importance are present in this table. When looking at the features that are computed with SATzilla, then there are only 7 features that are more important than 0.02.

Table 2: Most important features with respect to the information gain ratio.

Group	Features	Group	Features
SymmTime2	13	variable-clause_degree	4
SymmTime1	9	Exactly1Lit	3
variables_graph_degree	6	SymmTime0	4
variables_graph_weights	6	Full_AND_gate_degree	3
RWH-1	6	variable-clause_polarity	2
RWH-2	5	bin_implication_graph_degree	1
RWH-3	5	Blocked_AND_gate_weights	1
Full_AND_gate_weights	5	resolution_graph_degree	1
clause-variable_degree	4	clauses_graph_degree	1
Other features	6		

4.2 Performance of Available Configurations

The used configurations of Riss are BMC08, CircuitFuzz, default, GI, IBM, LABS and SWV from the *Configurable SAT Solver Challenge 2013*[3], as well as the configuration Riss3g from the *SAT Competition 2013*.[4] Table 3 shows the number of instances that can be solved by each of the configurations on the benchmark, and furthermore includes the two state-of-the-art SAT solvers Glucose 2.2(Gl 2.2) and Lingeling(Lgl) [8]. Furthermore, for the configurations of Riss the number of *unique solver contributions* (UC) is presented, as well as the *virtual best solver*(VBS), which selects for each instance of the benchmark the fastest configuration. The UC states how many instances can be solved by this configuration only, but not by any other configuration of Riss.[5] Hence, UC shows a first potential of a classification approach, because by selecting the right configuration, all these instances can be solved as well. However, the unique solver contribution is only an approximation of the overall potential, because instances that can be solved by only two configurations are not considered. The difference between the best configuration Riss3g and VBS of 306 solved instances cannot be covered with the UC only, because this sum is 104.

Note that the table shows furthermore that, on the selected benchmark, the two state-of-the-art SAT solvers Lingeling and Glucose are more competitive than any configuration of Riss. Glucose can solve almost 100 instances more, mainly due to additionally solved satisfiable instances. Lingeling is especially good on solving additional unsatisfiable benchmarks.

4.3 Instance Classification

We compare the approach as presented in Section 3.1, which we call BlackBox, to the two state-of-the-art SAT solver Glucose 2.2 and Lingeling [8]. The training instances for BlackBox consists of instances that have been used for the competitions 2011 and 2012, and furthermore includes the unselected instances. The classifiers model is based on computing features with no Resolution or Clauses graphs and the labelling version is the complement one. This configuration is the one that performed better. Furthermore, for the configurations of Riss we compute the *virtual best solver* (VBS), which solves a formula as fast as the fastest configu-

[3]For details see `http://www.cs.ubc.ca/labs/beta/Projects/CSSC2013/results.html`.
[4]For details see `http://satcompetition.org/2013/results.shtml`.
[5]Lingeling and Glucose are not considered for this analysis.

Table 3: Performance of the Configurations

	BMC08	CIRCUIT FUZZ	DEFAULT	GI	IBM	LABS	RISS3G	SWV	VBS	LINGELING	GLUCOSE 2.2
solved instances	1483	1264	1403	1231	1533	1374	**1546**	1356	1852	1602	1649
SAT	859	814	851	747	**904**	901	**904**	807	1109	932	994
UNSAT	624	450	552	484	629	473	**642**	549	743	670	655
SAT UC	15	13	8	2	16	**33**	12	3			
UNSAT UC	3	2	5	**13**	3	9	9	0			

rations. Additionally, the *virtual worst solver* (VWS) is added. This solver behaves as well as the worst configuration, so that this solver solves only instances that can be solved by all configurations. These two virtual solvers are interesting, because they represent the best case of the configuration prediction (VBS) and the worst case (VWS). Finally, the configuration that solves most of the instances, RISS3G, is added to the comparison. The plot in Figure 4 shows the cactus plot. For the timeout of 900 seconds, the figure shows that the prediction is a nice trade-off between the best configuration RISS3G and the VBS. BLACKBOX is located in the expected area. The most trivial prediction would always select the best configuration RISS3G, so that this configuration forms the actual lower bound.

4.3.1 Comparing BlackBox to the Used Configurations

The comparison of the prediction in BLACKBOX to the fallback configuration RISS3G is also interesting, because BLACKBOX has to compensate the time used for classification with selecting a good configuration. As long as the default configuration is selected, this time can be consid-

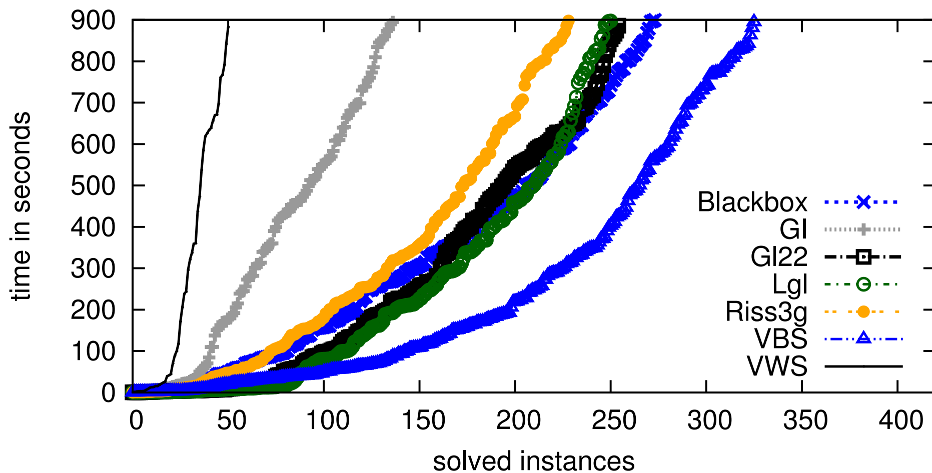
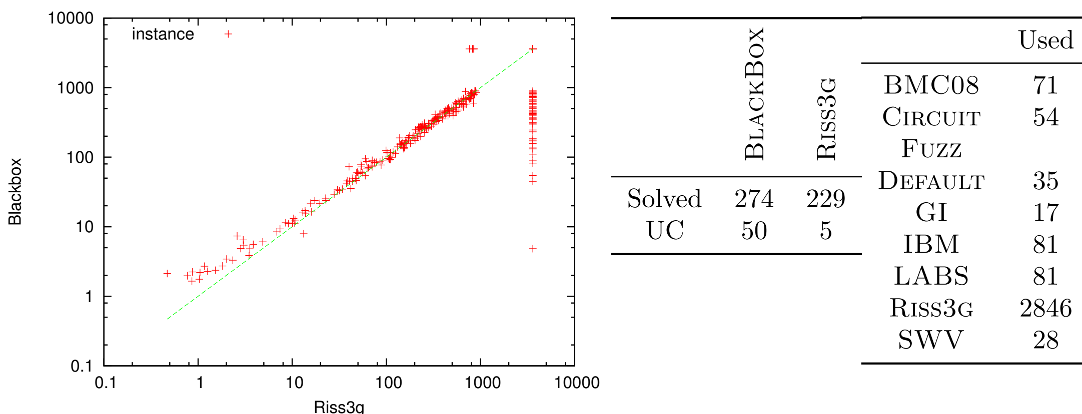


Figure 4: Comparing the performance of the prediction BLACKBOX to state-of-the-art solvers



	BlackBox	Riss3g
Solved	274	229
UC	50	5

	Used
BMC08	71
Circuit	54
Fuzz Default	35
GI	17
IBM	81
LABS	81
Riss3g	2846
SWV	28

Figure 5: Comparing the prediction BlackBox to the fallback configuration Riss3g (left: CPU time scatter plot, middle: number of solved instances and unique solver contributions on 600 instances from 2013, right: number of times a configuration has been chosen on all 3213 instances).

ered as overhead. Another interesting number is the number of unique solver contributions of the fallback configuration Riss3g, because this number show the number of instances where the classifier either consumed too much run time and afterwards selects this configuration, or where the classifier selected another configuration that could not solve the instance. These two comparisons are presented in Figure 5. The overhead of the feature extraction and classification seems to be negligible, most of the dots in the plot are very close to the identity line. For only five instances the overhead makes the difference between being solved and not being solved. These five instances are also present as unique solver contributions. On the other hand, for 50 instances BlackBox selects a different configuration that is capable to solve the instances that cannot be solved by Riss3g. Furthermore, the distribution for selecting configurations out of the available configurations is given. As expected, the fallback configuration is selected most of the time. Note that the number of being selected as a configuration correlates to the number of unique solver contributions (in Table 3). The configurations that have a high UC, namely BMC08, IBM and LABS are selected more frequently then the other configurations.

A final evaluation has to be made on the instances that have been added newly to the benchmark 2013. Usually, instances of previous competitions are used again, or previously unselected instances are selected in a latter year. When the benchmark of 2013 is reduced to the newly added instances[6], only 351 instances are left in the benchmark. As expected, the new files are not detected by the classification with the Complement labelling. Out of the eight available configurations the default Riss3g is selected 333 times, and only 18 times the configuration BMC08 is used. These numbers indicate that the decisions made in the design phase result in a classification that strongly relies on a robust default configuration. While this property is not bad per se, we consider relaxing that classification as future work, so that the resulting BlackBox considers more configurations on novel formulas. The potential for this future work is present: 40 times the configuration Riss3g is selected, and the solver cannot solve the instance, although there is another configuration that can solve the instance. On the other hand, for the 18 times where the configuration BMC08 is selected, Riss3g can solve exactly one instance, whereas BlackBox does not solve the instance, because BMC08 cannot

[6] This reduction is done based on the file name of the instances in the benchmark.

Table 4: Comparing BlackBox with theoretical combinations of features and labellings.

Features	Labelling	Solved	Novel Solved
BlackBox	Complement	274	121
NoCGRes	Complement	**276**	**123**
NoCGRes	Relative	242	97
NoCGRes	Global	264	111
SATzillaBase	Complement	239	98
SATzillaBase	Relative	224	91
SATzillaBase	Global	221	81
NoCGResNoNew	Complement	273	120

solve any of the 18 instances.

4.3.2 Comparing BlackBox to Using Different Features and Classifications

While the above experiments have been created by running BlackBox on the cluster, with the obtained data during computing the features for all feature sets, and for evaluating all configurations, we can furthermore perform theoretical experiments by using the pre-computed features and configuration run times. Hence, we first compare BlackBox to its theoretical variant that does not include the classification time. Furthermore, we compare the different labellings, and check how the new features influence the classification. Finally, we exchange the feature set and use the widely used SATzillaBase.

The results for the 600 instances of the 2013 benchmark are presented in Table 4. First, by executing the actual solver, two instances cannot be solved. Furthermore, the best labelling is the labelling Complement for both feature sets. Where for our implementation of the features the labelling Global gives better results then Relative, for the SATzillaBase features this effect is not present. When our feature extraction is compared to SATzillaBase, then the classifier for the two labellings Complement and Global we perform significantly better. A future work analysis has to show, whether feature filtering improves this picture, as we currently provide many more features than SATzillaBase. Currently, the new features help to classify only three more instances correctly.

5 Conclusion

In this paper we present a set of new features for CNF formulas. For the features we use the score of the RWH heuristic, a symmetry approximation, as well as a gate recognition that detects partially and fully encoded AND-gates as well as exactly-one constraints. With an efficient implementation we show that these features have the potential to replace the widely used feature extraction of SATzilla.

Furthermore, we show how features can be used to select a configuration for a highly configurable SAT solver. For preset configurations we present three objective functions how a configuration should be chosen, where relying on a strong fallback configuration seems to be very valuable on the used benchmarks. When combining such a labelling with the features, we built a classifier that chooses a solver configuration for a given formula, improving the perfor-

mance of the SAT solver Riss significantly. When using our features, the classification results are already better than for the SATzilla features.

Consequently, future work is to improve the classifier, so that more novel formulas are classified correctly, but without loosing too much performance. Furthermore, the trade-off between memory consumption and run time should be adopted in favor of memory consumption, because for large formulas already 6 GB of memory are necessary. Finally, SAT-related formalisms, such as MaxSAT or QBF, can benefit from the classification, because for these tools there also exists many highly configurable solvers that can benefit from classification. Hence, the feature extraction will be adopted to the corresponding formats.

We hope that the presented approach is useful not only for selecting a configuration of an existing solver, but furthermore enables the research community to look into techniques that might be costly, but that are extremely helpful in solving formulas from a certain area. With the classification approach, those techniques can be selected in the right case, but do not reduce the overall performance of the SAT solver. Then, more techniques might be proposed that are valuable in increasing the diversity of applications for SAT solving.

Acknowledgments The authors thank the ZIH of TU Dresden for providing the computational resources to produce the experimental data for the empirical evaluation.

References

[1] Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult SAT instances in the presence of symmetry. In: DAC 2002. pp. 731–736. ACM, New York, NY, USA (2002)

[2] Ansótegui, C., Bonet, M.L., Levy, J.: On the structure of industrial sat instances. In: Gent, I.P. (ed.) CP. LNCS, vol. 5732, pp. 127–141. Springer (2009)

[3] Ansótegui, C., Giráldez-Cru, J., Levy, J.: The community structure of sat formulas. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 410–423. Springer, Heidelberg (2012)

[4] Ansotegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent, I.P. (ed.) Principles and Practice of Constraint Programming – CP 2009. LNCS, vol. 5732, pp. 142–157. Springer (2009)

[5] Athanasiou, D., Fernandet, M.A.: Recursive weight heuristics for random k-SAT. Tech. rep., Delft University of Technology (2010)

[6] Balint, A., Belov, A., Diepold, D., Gerber, S., Jrvisalo, M., Sinz, C.: Proceedings of SAT Challenge 2012; Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, University of Helsinki, Helsinki, Finland (2012)

[7] Balint, A., Belov, A., Heule, M.J., Järvisalo, M. (eds.): Proceedings of SAT Challenge 2013, Department of Computer Science Series of Publications B, vol. B-2013-1. University of Helsinki, Helsinki, Finland (2013)

[8] Biere, A.: Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In: Balint et al. [7], pp. 51–52

[9] Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press, Amsterdam (2009)

[10] Breiman, L.: Random forests. Machine Learning 45(1), 5–32 (2001)

[11] Brownlee, J.: Clever Algorithms: Nature-Inspired Programming Recipes. Lulu Enterprises Incorporated (2011)

[12] Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995)

[13] Cristianini, N., Shawe-Taylor, J.: An Introduction to Support Vector Machines and Other Kernel-based Learning Methods. Cambridge University Press, 1 edn. (2000)

[14] Gableske, O.: https://www.gableske.net/dimetheus. `https://www.gableske.net/dimetheus`, accessed: 2014-04-14

[15] Großmann, P., Hölldobler, S., Manthey, N., Nachtigall, K., Opitz, J., Steinke, P.: Solving periodic event scheduling problems with SAT. In: Jiang, H., Ding, W., Ali, M., Wu, X. (eds.) IEA / AIE 2012. LNCS, vol. 7345, pp. 166–175. Springer, Heidelberg (2012)

[16] Gurney, K.: An Introduction to Neural Networks. Taylor & Francis, Inc., Bristol, PA, USA (1997)

[17] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. 11(1), 10–18 (Nov 2009)

[18] Heule, M., Walsh, T.: Symmetry in solutions. In: Fox, M., Poole, D. (eds.) AAAI. AAAI Press (2010)

[19] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: LION. LNCS, vol. 6683, pp. 507–523 (2011)

[20] Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306 (October 2009)

[21] Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: Isac - instance-specific algorithm configuration. In: Coelho, H., Studer, R., Wooldridge, M. (eds.) ECAI. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 751–756. IOS Press (2010)

[22] Kaiss, D., Skaba, M., Hanna, Z., Khasidashvili, Z.: Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In: Jobstmann, B., Ray, S. (eds.) FMCAD 2007. pp. 20–26. IEEE Computer Society, Washington (2007)

[23] Kroer, C., Malitsky, Y.: Feature filtering for instance-specific algorithm configuration. In: ICTAI. pp. 849–855. IEEE (2011)

[24] Kullmann, O.: New methods for 3-SAT decision and worst-case analysis. Theoretical Computer Science 223(1–2), 1–72 (1999)

[25] Manthey, N.: The SAT solver RISS3G at SC 2013. In: Balint et al. [7], pp. 72–73

[26] Manthey, N.: Coprocessor 2.0 - a flexible CNF simplifier - (tool presentation). In: SAT. LNCS, vol. 7317, pp. 436–441 (2012)

[27] Mijnders, S., de Wilde, B., Heule, M.J.H.: Symbiosis of search and heuristics for random 3-sat. In: Mitchell, D., Ternovska, E. (eds.) LaSh 2010 (2010)

[28] Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation 2(3), 293–304 (1986)

[29] Roussel, O.: Description of ppfolio 2012. In: Proceedings of SAT Challenge 2012; Solver and Benchmark Descriptions. University of Helsinki, Helsinki, Finland (2012)

[30] Tseitin, G.S.: On the complexity of derivations in propositional calculus. In: Slisenko, A.O. (ed.) Studies in Constructive Mathematics and Mathematical Logic, pp. 115–125. Consultants Bureau, New York (1970)

[31] Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for sat. JAIR 32(1), 565–606 (Jun 2008)