Diplomarbeit

Improving SAT Solvers Using State-of-the-Art Techniques

Norbert Manthey

15. Dezember 2010

Technische Universität Dresden
Fakultät Informatik
Institut für Künstliche Intelligenz
Professur für Knowledge Representation and Reasoning

Betreut von:
Prof. Dr. rer. nat. Steffen Hölldobler
Dipl.-Inf. Peter Steinke

Norbert Manthey
Improving SAT Solvers Using State-of-the-Art Techniques
Diplomarbeit, Fakultät für Informatik
Technische Universität Dresden, Dezember 2010

Aufgabenstellung Diplomarbeit

Name, Vorname:	Manthey, Norbert
Studiengang:	Informatik
Matrikelnummer:	3305366
Thema:	***Improving SAT Solvers Using State-of-the-Art Techniques***

Zielstellung:

Viele Probleme lassen sich als Erfüllbarkeitsproblem (SAT) kodieren. Innerhalb der letzten Jahre wurden dafür performante Solver entwickelt. Die Leistung von diesen SAT-Solvern übertrifft in vielen Fällen die Leistung der spezifischen Lösungsprogramme für die jeweiligen Probleme. Dadurch werden viele industrierelevante Anwendungen nach SAT kodiert und mit einem SAT-Solver gelöst. Dabei wächst die Größe der Probleme, wodurch die Anforderungen an SAT-Solver immer weiter steigen.

Um die Auswirkungen der letzten Entwicklung besser verstehen zu können, sollen in dieser Arbeit die einzelnen Techniken, welche in SAT-Solver Einzug erhielten, vorgestellt und analysiert werden. Durch diese Untersuchengen soll gezeigt werden, welche dieser Techniken zu dem Leistungssprung von SAT-Solvern innerhalb der letzten Jahre geführt haben. Durch die Kombination der besten Techniken soll ein SAT-Solver entstehen, der die Leistung eines state-of-the-art SAT-Solvers hat und auch für das Lösen industrieller Probleme geeignet ist.

Die aktuellen Techniken sollen in einem kompontenbasierten SAT-Solver implementiert werden, um die Auswirkungen der einzelnen Veränderungen auf die Leistung des Solvers nachvollziehen zu können.

Schwerpunkte:

- Analysieren der aktuellen Techniken in modernen SAT-Solvern
- Implementieren der neuen Komponenten in einen SAT-Solver
- Empirische Untersuchung der Auswirkungen der einzelnen Komponenten und deren Kombination

Betreuer:	Dipl.-Inf. Peter Steinke
verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. Steffen Hölldobler
Institut:	Künstliche Intelligenz
Lehrstuhl:	Knowledge Representation and Reasoning
Beginn am:	01.08.2010
Einzureichen am:	01.02.2011

Abstract. This work discusses modern techniques of state-of-the-art SAT solvers. Since most techniques are published without regard to the effect to recent techniques of other developers, this interference is studied in this work. Therefore, configurations for SAT solver components are combined and the performance of these combinations is analyzed. The SAT solver is divided into the following components: decision heuristic, unit propagation, conflict analysis, removal, restart strategy and preprocessor. The components of the CDCL-based SAT solver *riss* are extended by state-of-the-art techniques. The performance of these components is analyzed by using the application benchmark of the SAT Competition 2009 with its 292 instances from various industrial applications. The performance is measured by the number of instances that can be solved in a timeout of 3600 seconds each. Based on the performance of the configuration per component, the best performing configurations are combined to a major configuration, increasing the number of solvable instances by 10%. By analyzing the effect of the single component configurations on the major configuration, it has been shown that the performance of two components interfere with each other and consequently finding the best configuration for a SAT solver is not trivial. Furthermore, a component based SAT solver implementation has a slower run time. The best configuration that was found during the work by combining state-of-the-art SAT solving techniques is able to solve 13% more instances than the baseline configuration. The new techniques show that activity based components have a higher performance than components that are based on the clause length.

Contents

1. Introduction

Solving a Sudoku can be done in many different ways. The easiest way is to guess the values for all the free squares and verifying the solution afterwards. The most sophisticated method is to reason about the fields that are already filled and to assign numbers to free cells without guessing. Thus, solving a Sudoku by a program can be done by implementing a naive algorithm, namely checking every single solution candidate, or by inventing a special purpose solver that is made to solve Sudokus efficiently. Developing and implementing such a special purpose solver is time consuming because all the rules that are used for reasoning to be implemented. Another approach to solve the Sudoku is to transform the Sudoku into another domain and solve the new problem in this domain. Thus, only a transformation from the original problem, the Sudoku puzzle, into the new domain has to be found and an efficient solver for the new domain is needed. Sudokus can be transformed into propositional formulas [ILO06]. Afterwards, the propositional formula can be checked for a satisfying assignment for the involved variables. This step is called satisfiability testing (SAT) and is done by so called SAT solvers. If an assignment is found, this assignment can be transformed back into the domain of the Sudoku and the free cells can be filled.

The example of Sudoku puzzles shows that propositional logic can be used to describe other problems. This feature is not the only reason why SAT is used to solve real world problems. The performance of SAT solvers increased significantly in the last 15 years. The worst case execution time to solve a problem with n variables is still exponential in the number of variables. Fortunately, in average most problems can be solved much faster by modern SAT solvers enabling users to successfully tackle formulas with millions of variables [IB10]. A main reason is the introduction of conflict-directed clause learning (CDCL) [SS96] that enabled solvers to cut of big parts of the search tree. In addition, studies showed that restarting the search from time to time also helps to decrease the average runtime significantly [Hua07]. Searching a solution is also not done naively any more. Choosing the next variable that has to be assigned is based on properties of this variable. Thus, variables that seem to be important are picked more frequently [MMZ$^+$01]. Adding these heuristics to SAT solvers also increased their performance. Furthermore, the implementation of these solvers has recently been improved. Improved data structures are used to take advantage of properties of the algorithm. The two watched literal scheme for example avoids unnecessary work during unit propagation [MMZ$^+$01]. There have also been attempts to adjust the implementation more to the provided resources of the computing hardware and to utilize the provided hardware units, for example the cache and the prefetching unit, as well as possible [HMS10].

All the mentioned improvements turned SAT solvers into powerful tools that can solve large formulas efficiently. Simple problems like Sudokus can be solved without much effort, but SAT solvers are also used to tackle real world problems from many domains resulting in much more complex formulas. Applications that are converted into SAT come hardware and software verification, planning, scheduling, configuration, termination analysis and many more (see e.g. [BM00, FGM$^+$07, KS92, LMS06]). There also are other applications that are usually solved in the domain of the constraint satisfaction problem (CSP). Most of the problems that can be solved in the domain of CSP can also be transformed into the domain of SAT [TTKB09, Gen02], increasing the number of fields that use SAT solvers even further. All the problems that are solved by using a SAT solver can also be solved by a single special purpose solver. However, it seems to be more efficient to use a SAT solver whose development is pushed by annual competitions and increasing requirements of the industry. Using SAT solvers seems to be the most efficient known way to solve these large problems. Conveniently, as shown in Figure 1 only an encoder and a decoder have to be implemented and afterwards highly optimized SAT solvers can be used to find a solution as a black box. This scheme shows that using a SAT solver can also be the faster solution to solve a problem, because the conversion is more easy to implement than inventing an efficient special purpose solver if the problem can be converted into SAT easily. This work will focus on the solving the formula only.

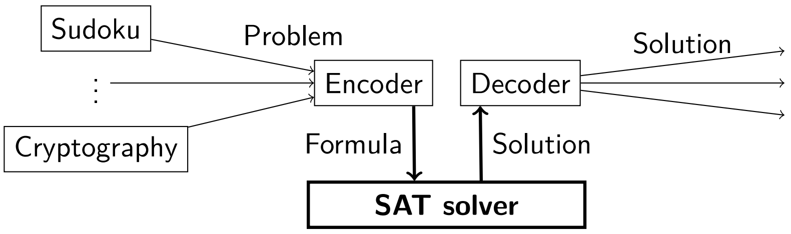


Figure 1: Using a SAT solver as black box

Applications for SAT solvers became larger because industrial problems have been transformed into SAT. Formulas containing millions of variables have to be solved. Due to new applications for SAT solvers, new algorithms are also developed and integrated into SAT solvers [Soo10]. Furthermore, SAT techniques are improved to be able to solve recent problems faster. These improvements and new techniques are presented at international conferences and their performance is tested in annual competitions. Unfortunately, most SAT solvers are implemented in a fixed configuration. Thus, it is not possible to compare the effect of a new technique with respect to another configuration of the solver easily. The effect of a single new technique is presented, but an overall overview of current techniques and their interference with the new development is missing.

This work tries to fill this gap. Techniques that have been presented recently are analyzed and their performances are compared. Most recent SAT solvers are described briefly in their competition configurations. In these descriptions only the main techniques are enumerated. The most recently used techniques in leading SAT solvers are collected and combined in the SAT solver *riss* [Man10] because it is implemented by using components that are easily exchangeable. The solver is also easily extendable. The performance of the baseline version of the solver, called rissR0, is comparable to MiniSAT 2 [SE09], a well known SAT solver that is not easily extendable. The techniques that are implemented in MiniSAT 2 are also implemented in rissR0. These techniques are conflict driven clause learning [SS96] with minimization [SB09], the VSIDS decision heuristic [MMZ$^+$01], restarts [GSCK00] and the two watched literal unit propagation [MMZ$^+$01]. Furthermore, both SAT solvers implement variable elimination [EB05] as the main preprocessing technique before the search is started. For comparing the performance of the solver configurations, the application benchmark of the last SAT Competition 2009 with 292 instances of several industrial applications is used. The most interesting measure for a user, namely the execution time, has been chosen as the criterion to compare two configurations. The configuration that is able to solver more instances within a certain time for each instance is considered to be the more powerful configuration. The timeout is set to 3600 seconds. Additionally, a memory limit of 2 GB has been added, similarly to the competition. The majority of industrial applications can be solved efficiently by SAT solvers that are based on the CDCL algorithm instead of using other SAT techniques. Thus, only techniques for these solvers are analyzed in this work. Stochastic and look-ahead SAT solvers are not considered.

Based on recent publications, new techniques have been implemented into *riss* and their performance has been compared. These techniques have been divided into the main components of a SAT solver, namely decision heuristic, unit propagation, conflict analysis, removal, restarts and the preprocessor, where the preprocessor is also an inprocessor because it is able to simplify the formula also during the search. For each of these components, rissR0 has been modified and all the single modifications have been compared. Recent techniques are for example phase-saving for the decision heuristic [PD07], restarting the search according to the Luby series [LSZ93] or applying on-the-fly self subsumption during conflict analysis [HS09]. Afterwards the best configuration per component has been picked and a new configuration for the solver has been created based on these single configurations. The effect of single component configuration improvements has been analyzed while all other components are already configured to the most powerful configuration.

The baseline configuration rissR0 is used as a reference for all the modifications. It is able to solve 147 instances of the benchmark within the specified limits. After all components have been improved separately, the best performing configurations have been combined. This configuration is able to solve 175 instances, which is 10% more instances of the benchmark than the baseline configuration can solve. The best found configuration of this work is able to solve 183 instances by setting a single component, the inprocessor, back to its standard configuration. The performance of the solver on the specified benchmark has been improved by 13%. During the analysis the comparison of the possible configurations per component do not show only the overall performance, but also the performance for lower timeouts. Although the performance is compared using the specified timeout, the performance analyses can also be used to find the best configuration for a lower timeout. This work yields another result, namely a SAT solver that can easily be trained to perform well on a certain kind of instances without changing the implementation, because it provides a very large set of component configurations.

This work is structured as follows. In Section 2 basic notations of SAT and the main algorithms that are used in modern SAT solvers are given. In Section 3 recent techniques for the components of a SAT solver are introduced and their performance is compared. Combining these configurations and analyzing the interference among the components is done in Section 4. Finally, the results are summarized in Section 5 and an overview on the remaining work is given in Section 6.

2. SAT Solving

SAT solving is widely used, because it uses a simple way to state a problem, namely the propositional logic. In this logic any variable can be assigned only true or false and it provides basic connectives, which combine the value of two variables to a new value. Given this simplicity, it is easy to encode many problems into SAT and solve them using SAT solvers.

In the following Section 2.1 propositional logic is introduced and the restricted alphabet that is used by SAT solvers is explained. Afterwards, techniques to solve the SAT problem are explained in Section 2.2. Recent approaches to compare SAT solvers are presented in Section 2.3.

2.1. Propositional Logic

Propositional logic, in general, consists of a countable infinite set of propositional variables, connectives and special characters. Since SAT solvers handle only formulas that are in conjunctive normal form (CNF), only the necessary terms for this form will be given. Propositional logic in general is described in [H09̈]. The following Section 2.1.1 will introduce terms that belong to the syntax of the formula. The semantics of propositional logic is briefly introduced in Section 2.1.2 to understand how SAT problems can be solved. Techniques to modify a propositional formula in CNF are presented in Section 2.1.3.

2.1.1. Syntax

A propositional formula F is a formula over an alphabet Σ that consists of a countable infinite set V of propositional variables, the set of connectives $\{\neg, \wedge, \vee\}$ and the special characters "(" and ")". The connectives are called *negation*, *conjunction* and *disjunction* respectively. The negation is unary, whereas conjunction and disjunction are binary connectives. Propositional variables are either numbers 1, 2, ...or small letters v, v', ... The negation for the number representation is stated as $\neg 1$ whereas the negation for the letter representation is stated $\overline{v}$.

Definition 1 *An atom is a propositional variable v or its negation $\overline{v}$.*

Definition 2 *A literal is an atom l or its negation $\bar{l}$.*

Definition 3 *The polarity of a literal is negative, if the literal is an negated atom $\bar{l}$. Otherwise the polarity of the literal is positive.*

Definition 4 *A clause is a disjunction of literals.*

Definition 5 *A formula in CNF is a conjunction of clauses.*

A clause C that represents the disjunction of literals $((l_1 \vee l_2) \vee (l_3 \vee l_4)) \vee l_5$ will be written as $[l_1, l_2, l_3, l_4, l_5]$. The literals of a disjunction can be freely ordered because the disjunction is commutative. The number of literals in a clause is also called the *size* of the clause. Special clauses are the *unit clause* or *unit* of size one and the *binary clause* of size two. A formula F that represents the conjunction of clauses $(C_1 \wedge C_2) \wedge C_3$ is written with diamond brackets $\langle C_1, C_2, C_3 \rangle$.

An example formula is the formula

$$F_{exp} = \langle\ [1,\ 3],\ [\neg 2,\ \neg 5,\ \neg 6],\ [\neg 1,\ \neg 4,\ 6],\ [\neg 1, \neg 2, \neg 4,\ 5],\ [\neg 1,\ 2] \rangle$$

This formula F_{exp} is used during the following sections to demonstrate presented techniques. The formula contains five clauses. Each of the clauses are named C_i where i is the index of the clause in the formula. For example, C_3 =[¬1, ¬4, 6].

2.1.2. Semantic

The domain of a propositional variable has two elements, namely *true* and *false*. Thus, the value of a propositional formula can be either true or false, depending on the values of its variables and the connectives between these variables.

Definition 6 *An assignment α of a set V of Boolean variables is a mapping α: V→{false, true} that assigns a value true or false to every variable of V.*

Definition 7 *A literal is satisfied if it is an atom that is mapped to true or if it is a negated atom that is mapped to false.*

Definition 8 *A clause is satisfied if one of its literals is satisfied, where an empty clause is unsatisfied.*

Definition 9 *A formula is satisfied if all its clauses are satisfied, an empty formula is always satisfied.*

An assignment of the formula is also called interpretation. Interpreting the formula F under an assignment α is written as $F|_\alpha$. Calculating this interpretation is done by applying the following two rules:

1. Remove all clauses from F that contain a satisfied literal.
2. Remove all unsatisfied literals from the remaining clauses.

Definition 10 *A partial assignment is an assignment α that does not contain a mapping for all variables of the given formula F. An assignment that contains a mapping for all variables is called complete assignment.*

Definition 11 *A variable that is not assigned by a partial assignment α is called undefined. A literal whose variable is undefined is also called undefined.*

In this work a partial assignment is called assignment only. Extending a partial assignment by a mapping that satisfies a literal l is denoted by αl. Interpreting a formula F by an assignment α is done by applying the same steps as for applying a complete assignment to a formula.

Definition 12 *If an assignment exists that evaluates the formula to true this formula is satisfiable. If there does not exist such an assignment the formula is unsatisfiable.*

Applying an assignment $\alpha = [1, 2, 3, 4, 5]$ to the example formula F_{exp} results in the following formula $F_{exp}|_\alpha \equiv \langle[\neg 6], [6]\rangle$. The clause C_1 is satisfied because it contains the literal 1. In C_2 the negative literals ¬2 and ¬5 are removed so that the remaining clause is [¬6]. In C_3 only [6] is left because the other negated literals are removed. C_4 and C_5 are removed from the formula because they are satisfied by 5 and 2, respectively.

Definition 13 *An implication of the form $(l_1 \wedge \cdots \wedge l_n) \to l'$ can be represented by the clause $[\overline{l_1}, \dots, \overline{l_n}, l']$.*

In general, any propositional formula can be converted into CNF. For this work, only the transformation of implications is needed, so that the general transformation rules are not given. The interested reader can find more on this conversion in [HÖ9].

2.1.3. Formula Modification Techniques

There are several techniques that can be applied to clauses of a formula F that do not change the satisfiability of F. These techniques either add or remove clauses or remove literals from clauses in F.

Definition 14 *Let $C_1 = [x, a_1, \dots, a_n]$ and $C_2 = [\overline{x}, b_1, \dots, b_m]$ be two clauses that share a common variable x with different polarities, a new clause called resolvent $C = [a_1, \dots, a_n, b_1, \dots, b_m]$ can be obtained by removing all positive occurrences of literal x in C_1 and all occurrences of $\overline{x}$ from C_2 and adding all remaining literals to C. This operation is called resolution.*

The first useful technique is the *resolution* of two clauses resulting in a new clause, called resolvent. The satisfiability of a formula F does not change when a resolvent $C = C_1 \otimes C_2$ of two clauses $C_1, C_2 \in F$ is added to the formula [BHvMW09, p. 138]. The resolution using the variable x is denoted by $\otimes_x$. The resolution operation can be lifted to sets of clauses [EB05]. Let S_x be the set of clauses that contain the literal x and $S_{\overline{x}}$ the set of clauses that contain the literal $\overline{x}$. The resolution of the two sets $S_x \otimes_x S_{\overline{x}}$ is defined as

$$S_x \otimes_x S_{\overline{x}} = \{C_x \otimes_x C_{\overline{x}} \mid C_x \in S_x, C_{\overline{x}} \in S_{\overline{x}}\}$$

The second technique is called *subsumption* and can be used to remove clauses from the formula.

Definition 15 *A clause $C_1 = [a_1, \dots, a_n]$ subsumes another clause $C_2 = [a_1, \dots, a_n, b_1, \dots, b_m]$ if C_2 contains all literals of C_1.*

To satisfy a formula F, all its clauses have to be satisfied (compare definition 9). If C_1 subsumes C_2 and F is satisfiable, then C_1 has to be satisfied and thus contains a satisfied literal l. The features of subsumption ensure that this literal is also part of C_2 and thus whenever C_1 is satisfied, C_2 is also satisfied and can be removed from the formula. For an unsatisfied formula F it is obvious that C_2 can be removed.

The third technique is a combination of the two previously described techniques resolution and subsumption. Let $C_1 = [x, a_1, \ldots, a_n]$ and $C_2 = [\overline{x}, a_1, \ldots, a_i]$ be clauses of a formula F and $1 \leq i \leq n$, then the resolvent is $C_3 = C_1 \otimes C_2 = [a_1, \ldots, a_n]$. This resolvent subsumes the first clause C_1 that has been used for the resolution. By adding the resolvent C_3 to the formula F, the clause C_1 can be removed. These steps can also be seen as removing the literal x from clause C_1. The operation is called *clause strengthening*, *self subsumption* or *self subsuming resolution* in the literature.

2.2. SAT Solving Techniques

The numbers of variables in SAT instances are quite high in recent instances. There are formulas that contain more than 10 million variables and even more clauses [IB10]. Still, solutions for these formulas can be found by SAT solvers. Following the naive approach, all solutions would have to be tested. If n is the number of variables of a formula F, then 2^n solution candidates have to be checked. To improve finding a solution for a formula, partial assignments are used. These partial assignments can be represented in a binary tree, the *search tree*, which is described in section 2.2.2. To process this tree faster, the Davis Putman Logemann Loveland procedure (DPLL) has been introduced [DLL62]. This algorithm is described in section 2.2.3. Modern SAT solvers implement a modification of the DPLL algorithm that is also based on search trees. It is called Conflict Driven Clause Learning and is explained in section 2.2.4. The two algorithms can be seen as a depth first search in the search tree. Some of the properties of the propositional logic are implemented weakly in SAT solvers. Thus, these restrictions are explained in section 2.2.1 before the solving techniques are introduced.

2.2.1. Implementation Restrictions

To simplify and accelerate the work of a SAT solver, certain assumptions are added to the formula without changing the satisfiability or the model of the formula. The first assumption is that any literal occurs only once in a clause. Duplicate literals are simply removed from a clause. The implementation of the algorithms in a SAT solver ensures that this assumption is true during the whole search. Furthermore, it is assumed that all clauses do not contain a literal l and its negation $\bar{l}$, because a clause with this property is always satisfiable and thus does not constrain the search space. Therefore, these clauses are also removed before the search is started.

During solving an instance in theory falsified literals are removed from the clauses. This step is usually not done by the implementation because these literals might be needed later on again. Thus, the unsatisfied literals are kept in the clause, but they will not be regarded any more if the size of the clause is discussed. A unit clause C under a partial assignment α might contain a single undefined literal l and a certain number of unsatisfied literals. The same effect applies for binary clauses and clauses of any other size.

2.2.2. Search Tree

In SAT, a search tree is a binary tree. Each node n has two child nodes n_l and n_r. The two edges from n to its children are labeled with a literal. Let the edge $n \to n_l$ be labeled with l, then the edge from $n \to n_r$ to its other child is labeled by $\bar{l}$. The variable used for the literals on the two edges is not allowed to occur on the path from the root of the tree to the node n. Every node in the tree is assigned a *level*. This level is the number of branches on the path to the node.

The path from a node n to the root of the tree represents a partial assignment that contains all the literals on this path. Since the aim of the search tree is to find a satisfying solution, a path can be closed if the according partial assignment evaluates a clause C of the formula F to false. Since all the literals of C are already on the path and consequently C does not contain any undefined literal nor a satisfied literal, it cannot be satisfied by adding more literals to the current path and thus extending this path can be stopped. On the other hand, if a branch contains all variables of F and cannot be closed by any clause of F, then this path represents a satisfying assignment for the formula. Therefore, a search tree has to be extended by adding new child nodes to existing nodes until a path containing all variables is found or if all paths are closed by a clause. If a satisfying assignment is found, the formula is satisfiable by exactly this assignment. Otherwise, if all paths are closed, the formula is unsatisfiable.

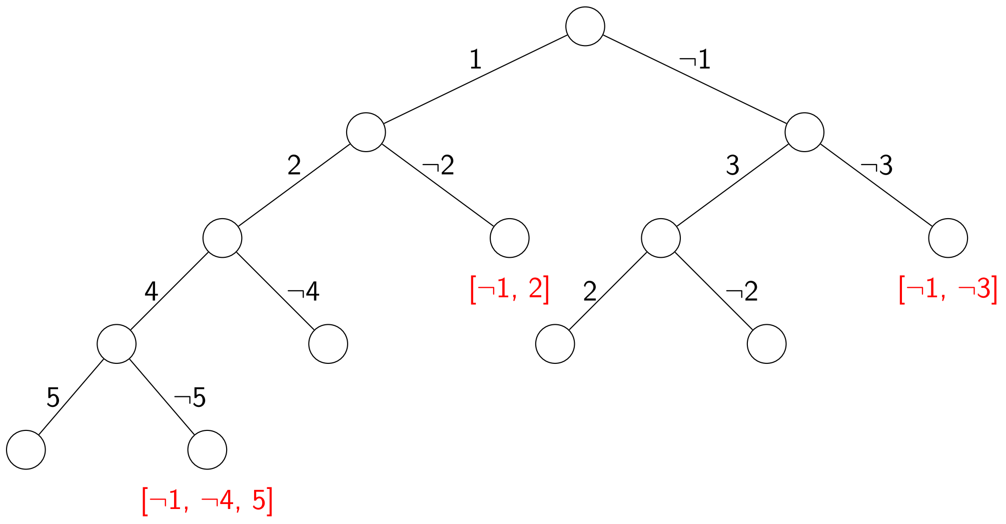


Figure 2: A search tree that is not completely extended

The search tree in Figure 2 shows a sample search tree for the formula F_{exp} that is not fully extended. An important property of the clause size can be seen. A short clause cuts off a big part of the search tree. For example, the clause $C_5 = [\neg 1, 2]$ forbids any assignment where variable 1 is assigned true and variable 2 is assigned false. Thus, one out of the four possible combinations is constrained so that this clause cuts off a quarter of the whole search tree. This statement can be generalized. A clause of size n cuts off up to the 2^n-th part of the search tree. When the number of leaves of a search tree that belongs to a formula with m variables are counted, a clause of size n cuts up to 2^{m-n} leafs. Thus, shorter clauses tend to a faster search because they may prune the search tree more than longer clauses.

2.2.3. Davis Putman Logemann Loveland Procedure

The Davis Putman Logemann Loveland [DLL62] (DPLL) procedure uses the search tree and sets up some effective rules to create the search tree. The algorithm is given in Algorithm 1 in its recursive version. It can be split into several rules. The first rule is called *SAT* and returns SATISFIABLE, if the current formula is empty with respect to the current assignment (lines 1-3). The second rule is called *UNSAT* and returns UNSATISFIABLE, if the current formula contains an empty clause (lines 4-6) or an unsatisfied clause with respect to keeping the literals in the clauses when an assignment is applied. The corresponding clause is called *conflict clause* or *conflict*. In literature it is also called *conflicting clause*. The next rule is one of the most important rules of the algorithm. Instead of always branching at a node, the DPLL procedure looks for unit clauses in the current state and satisfies them by extending the assignment by the literal of the unit clause (lines 7-9). Thus, this rule is called *unit*. The corresponding unit clause will be called *reason clause* or *reason*. The unit rule cuts half of the current sub tree. By applying the unit rule, visiting a sub tree that can always be closed is avoided. Thus, a node is not extended as in the search tree having two children. The current node gets only a single child and the edge between these two nodes is labeled with the literal of the unit clause. The new child node is at the same level of the tree than its parent node because the parent node does not branch the tree.

Definition 16 *A clause is a conflict clause with respect to an assignment if it is unsatisfied under this assignment.*

Definition 17 *A clause is called reason clause with respect to an assignment if it contains only a single undefined literal and all the remaining literals are unsatisfied.*

The next rule is usually not implemented in SAT solvers because it is expensive to check. It checks the current formula for *pure literals*, namely literals that only occur in one polarity. Satisfying a pure literal removes only satisfied clauses, but does not remove unsatisfied literals from other clauses so that no empty clause can be generated. The rule is called *pure* (lines 10-12). The last rule extends the current node by an unassigned literal as in the search tree (lines 13-17). Afterwards, it checks the first sub tree for a satisfying assignment (line 13). If this attempt fails, this sub tree does not contain a satisfying solution and the other sub tree is examined (lines 15-17). The satisfiability result of the second sub tree is the result for the whole formula and is returned. This rule is called *split* because it is the only rule that splits the underlying search tree and adds nodes of higher levels than the current level to the tree. It can also be understood as backtracking rule because, after the first search tree had been analyzed and no solution has been found, the search tracks back to the current node and proceeds with the other half of the tree.

Algorithm 1 DPLL(F, α)

```
1: if F$|_\alpha$ empty then
2:     return SATISFIABLE
3: end if
4: if F$|_\alpha$ contains an empty clause then
5:     return UNSATISFIABLE
6: end if
7: if F$|_\alpha$ contains an unit clause $[l]$ then
8:     return DPLL(F$|_{\alpha l}$)
9: end if
10: if F$|_\alpha$ contains a pure literal $l$ then
11:     return DPLL(F$|_{\alpha l}$)
12: end if
13: if DPLL(F$|_{\alpha l}$) = SATISFIABLE then
14:     return SATISFIABLE
15: else
16:     return DPLL(F$|_{\alpha \bar{l}}$)
17: end if
```

2.2.4. Conflict Driven Clause Learning Procedure

The Conflict Driven Clause Learning (CDCL) procedure was introduced in [SS96] and is an extension of the DPLL procedure. It tries to take advantage of the usage of the conflict clause better than the DPLL procedure, which simply performed backtracking. Instead, CDCL analyzes the conflict clause, learns a new clause and performs *backjumping* by undoing several decisions. When multiple decisions are undone, the order of the variables in the search tree can also change. Whenever the CDCL algorithm or a SAT solver based on this algorithm is discussed, backtracking will also refer to backjumping and thus can undo multiple levels. Two advantages of the CDCL algorithm with respect to the DPLL procedure arise: backjumping escapes faster from sub trees with no solution and the learned clause avoids entering a similar sub tree again, which does not yield a solution. The CDCL algorithm cannot be presented in a recursive version easily. Thus, its iterative version is given in Algorithm 2 using more variables that represent the current state of the search in the search tree. The given algorithm is similar to the implementations of modern SAT solvers.

Definition 18 *The trail is a stack that stores the assigned literals in the order they have been assigned.*

The algorithm starts with initializing necessary variables. Firstly, the assignment is cleared and the current level of the search is set to 0 (line 1) because the algorithm starts at the root of the search tree where no partial assignment exists and the level of the root node is 0. As next step, the reference for the conflict clause is set to *NO_CONF* because no conflict has been found yet. The current decision variable (the variable for the split rule) is set to a constant *NO_LIT*, representing that this literal has no value. The level

Algorithm 2 CDCL(F)

```
1: $\alpha \leftarrow \{\}$, $current\_level \leftarrow 0$;
2: $conflict \leftarrow NO\_CONF$; $decision \leftarrow$NO_LIT; level[$|V|$]; reason[$|V|$];
3: while $true$ do
4:    $conflict \leftarrow$ propagate(F, $\alpha$);
5:    if $conflict = NO\_CONF$ then
6:       $decision \leftarrow pick\_literal()$;
7:       if no decision possible then
8:          return SATISFIABLE;
9:       end if
10:      $current\_level \leftarrow current\_level + 1$;
11:      $\alpha \leftarrow \alpha decision$;
12:      $level[decision] \leftarrow current\_level$;
13:   else
14:      if $level = 0$ then
15:         return UNSATISFIABLE
16:      end if
17:      $clause \leftarrow$analyze($conflict$);
18:      $literal \leftarrow$ single literal from current level of $clause$;
19:      $current\_level \leftarrow max\{level[x] : x \in clause - \{literal\}\}$;
20:      backtrack($\alpha$, $current\_level$);
21:      $\alpha \leftarrow \alpha literal$;
22:      $level[literal] \leftarrow current\_level$;
23:      $reason[literal] \leftarrow clause$;
24:      $F \leftarrow F \cup clause$;
25:   end if
26: end while
```

of each variable is set to 0 because no variable has been assigned yet. Furthermore, the reason clauses for the variable assignments have to be initialized to zero (line 2).

The algorithm executes a while-loop (line 3-26) that terminates only if a solution for the formula, namely SATISFIABLE (line 7-9) or UNSATISFIABLE (line 14-16), is found. Thus, the following steps are repeated until a solution is found. Firstly, the current partial assignment is propagated (line 4). This step corresponds to the unit rule of the DPLL procedure. If a unit clause is found, the according literal is added to the current assignment and this variable is assigned the level which it has been assigned on. The unit clause is set as the reason for this clause and the literal is also put to a stack that stores the order of the assignments. This stack is called *trail*. In Algorithm 2 this trail is implemented in the assignment to keep the algorithm simple. SAT solver implementations usually implement an extra data structure for the trail because they need to access the assignment fast. Accessing an ordered structure is linear such that the assignment usually is implemented using an array with constant access time. The propagation step is repeated until no more unit clauses can be found or until a conflict clause is found. If a conflict clause is found, this clause is returned by the propagate method. Otherwise NO_CONF is returned to indicate that no conflict has been found.

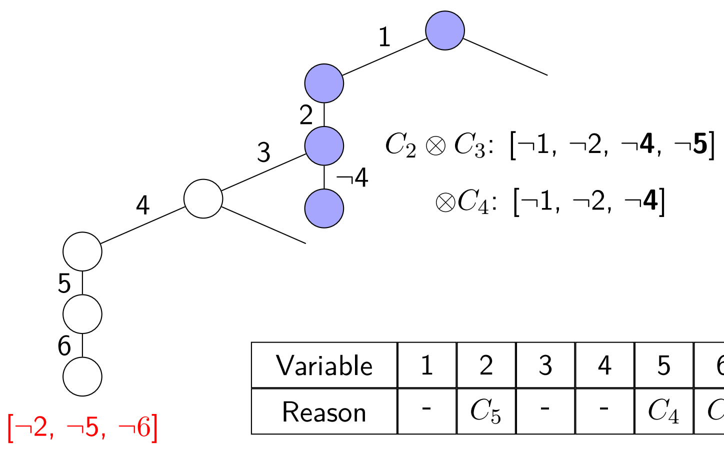
If no conflict is found (line 5-12) an unassigned literal has to be picked. This literal is stored in the variable decision. The decision literal is used to branch the search tree at the current node (line 6). If no unassigned literal can be picked, because all variables have already been assigned, a complete satisfying assignment has been found and the formula can be satisfied by this assignment because no conflict has been found during propagation. Otherwise the current node branches and one of its child nodes is examined. Note that the CDCL algorithm does not really branch the tree and processes both child nodes. As can be seen in Figure 3 most of the time the branched node is extended by more nodes. Still, there are cases where the algorithm examines both child nodes. Since the new chosen child node becomes the new current node with a higher level, the level has to be incremented and the decision literal has to be added to the assignment (line 10-11). Furthermore, as in the propagate method, the level for the decision variable has to be set to the current level (line 12). Then the while loop is repeated and the propagate method is called again to find new unit clauses. Whenever no conflict is found, the level of the current node will be increased and the search tree is extended by another branch.

If the propagate method finds a conflict clause, this clause is analyzed and *backjumping* is performed. If the current level is already the lowest level possible, no backtracking can be performed and thus no decision can be undone. Thus, the given formula F cannot be satisfied and UNSATISFIABLE is returned (line 14-16).

Otherwise the conflict can be analyzed (line 17) and a clause is generated. The analysis is explained in more details after the CDCL algorithm has been described. The generated clause is called *learned clause* and contains only a single literal from the current level (line 18). In literature it is also called *conflict clause*. The latter term is not used in this work to indicate the learned clause but the conflicting clause. After the learned clause is generated, backjumping is performed to a level on which the learned clause becomes a unit clause (19-20). During backjumping, the assignments of all literals above the backjumping level are reset and the level and reason information is also deleted. The learned clause becomes unit at the backjumplevel, because all of its literals except the one from the conflicting level are still unsatisfied. Since the all the variables from the conflicting level have been unassigned, there is a single undefined literal in the clause. This literal, which is the literal with the highest level in the learned clause, is added to the assignment (line 21). Furthermore, the level and reason information for this literal are also stored (line 22-23) and finally the learned clause is added to the formula before the new assignment is propagated on the formula again.

To explain the conflict analysis, certain properties of the reason clauses and the assigned literals have to be discussed. A literal that has been decided during the CDCL procedure is called *decision literal* and has no reason clause that forced its assignment. All other assigned literals are called *implied literals*. They do have a reason clause that has been unit clause before the literal was assigned. Thus, a reason clause C has only a single literal l that is satisfied whereas all the other literals are falsified with respect to the current assignment. Whenever a conflict clause $C_l = [l, o, a_1, \dots, a_n]$ is found, this clause contains only falsified literals. Note that at least one literal l has to be an implied literal. Otherwise this literal l would have been implied during the last propagation step on a lower level. Thus, the level of this literal *level*$[l]$ has to be the current level. Furthermore, there needs to be at least one other literal o of the current level that has been falsified. This literal o can be either an implied or a decision literal. Since l is an implied literal, there also needs to be a reason clause $C_{\bar{l}} = [\bar{l}, p, b_1, \ldots, b_m]$ for the negated literal $\bar{l}$ (p can be the same literal as o). The literal l will be called *conflicting literal*. The clause $C_{\bar{l}}$ also contains at least one other literal from the current level because otherwise this clause would have become a unit clause on a previous level and the literal $\bar{l}$ would have been assigned on that level. The aim of the conflict analysis is to produce a learned clause $C_{learn} = [s, c_1, \ldots, c_t]$ so that a part of the current partial assignment exists that makes the clause C_{learn} become unit. Thus, the learned clause should contain only a single literal s from the current level so that after backtracking to another level all literals from lower levels are still falsified and s is the only undefined literal so that C_{learn} is a unit clause with respect to the part of the current assignment. Note, that backtracking is always done such that the last variable that is unassigned due to backtracking is a decision variable and not an implied variable.

The learned clause C_{learn} is derived by applying resolution to the conflict clause C_l and the reason clause $C_{\bar{l}}$ of the conflicting literal l. The resolvent of these two clauses $C_r = C_l \otimes C_{\bar{l}} = [o, s, a_1, \ldots, a_n, b_1, \ldots, b_m]$ contains all the other literals and all of them are falsified. Thus, this clause can also be seen as the current conflict clause because it has the same properties as the conflict clause, except that it could contain only a single literal from the current level. If there is only a single literal from the current level, the analysis can be stopped and the current resolvent C_r is the learned clause. Otherwise, one of the literals s or o is chosen as the new conflicting literal and the procedure is repeated, until the resolvent contains only a single literal of the current level. Reaching such a clause is ensured because removing all the clauses of the current level by resolution cannot be done. The decision literal of the current level does not have a reason clause and therefore this literal cannot be resolved. Thus, the analysis always terminates. To ensure that this literal can be reached, the trail is traversed reversely and all variables that are element of the current resolvent C_r are used as variable for the next resolution step.


1
2
3
¬4
4
5
6
$C_2 \otimes C_3$: [¬1, ¬2, ¬**4**, ¬**5**]
$\otimes C_4$: [¬1, ¬2, ¬**4**]
[¬2, ¬5, ¬6]

Variable	1	2	3	4	5	6
Reason	-	C_5	-	-	C_4	C_3

Figure 3: Backjumping after finding a conflict using the CDCL procedure

Figure 3 shows an example search tree of a CDCL search including a learning and backjumping step for the example formula F_{exp}. The first path that is examined is the white one on the left side. The decision heuristic simply picks the next variable and assigns it positive. The table in the figure shows the assignment and the according reason clauses. After all variables have been assigned to true, the conflict clause C_2 is found during propagation. Now the procedure applies the conflict analysis and does the two resolution steps that are given next to the search tree. Note that the boldly printed literals are the literals of the current level of the search tree. As shown in the figure, resolution is applied until the resolvent contains only a single literal of the current level. Afterwards, the backjumping level is calculated. The level of the remaining literal is level 3. The second highest level in the clause is level 1. Thus, backjumping can be done to level 1. As the last step of the conflict analysis branch in the CDCL algorithm, the learned clause [$\neg$1, $\neg$2, $\neg$4] is added to the formula. Since this clause is unit under the current assignment (after backtracking), the literal $\neg$4 is added to the trail. This state can be seen in the path with the filled nodes. Differently to the CDCL procedure, the DPLL algorithm backtracks only to level 2 where the other branch on literal $\neg$4 is indicated. Thus, the example shows that CDCL is able to jump back further then the DPLL procedure and thus saves more time during inspecting the search tree.

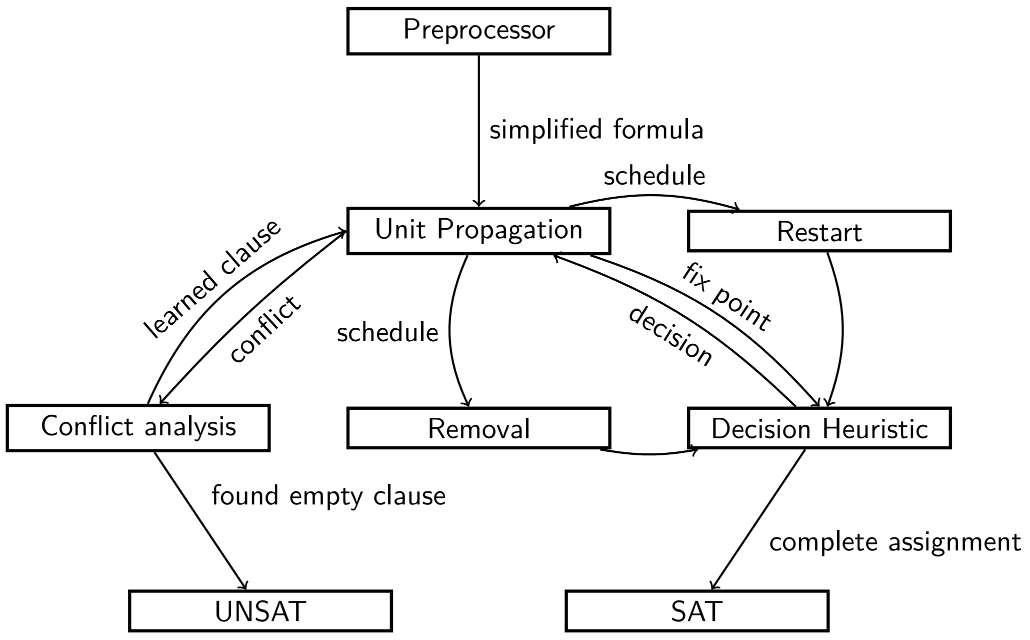


Figure 4: Flow diagram of the CDCL procedure in a component base SAT solver

A flow diagram of the CDCL procedure is given in Figure 4. This diagram also contains additional rules of the algorithm. The restart rule resets the current partial assignment by jumping back to level 0 and starting the search again. The removal deletes some of the learned clauses again from the formula. Finally, the preprocessor is run before the search is started to simplify the formula as much as possible. More details on these extensions are given in the related sections 3.4, 3.5 and 3.6 respectively. The used SAT solver *riss* is component based and splits the CDCL algorithm into the components *Decision Heuristic*, *Unit Propagation*, *Conflict Analysis*, *Removal*, *Restart* and *Preprocessor* as shown in the diagram.

2.3. Benchmarking

Comparing developments in SAT solvers is difficult. Depending on the encoded problem a chosen strategy might perform well or not. Thus, it is hard to determine in advance whether a certain SAT solver is good at solving an instance. To push the development of SAT solvers, annual competitions, namely the SAT Race and the SAT Competition, take place. These competitions usually limit the runtime by a certain time threshold. The number of solved instances determines the best solver. Thus, these competitions use a large set of encoded problems to compare the solvers. Furthermore, they strive to have a large variety among the instances so that all the areas where encoded problems might come from are considered. The benchmark that is used for this work is the application benchmark of the last SAT Competition 2009. This benchmark contains 292 instances that come, for example, from termination analysis [FGM$^+$07] and bounded model checking [BHvMW09].

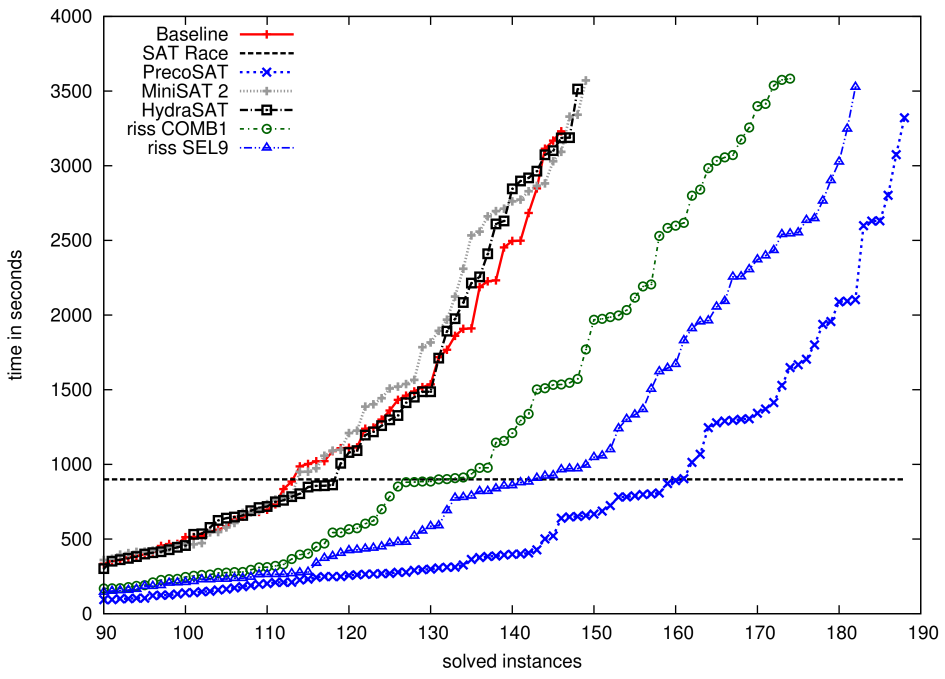


Figure 5: Snake plot to compare SAT solvers

To compare the performance of SAT solvers, a *snake plot* is used. An example plot is given in Figure 5. A line in the plot represents a SAT solver as labeled in the key in the top left corner. The points that are connected by the line represent a number of solved instances. The x-axis shows the number of solved instances and the y-axis shows

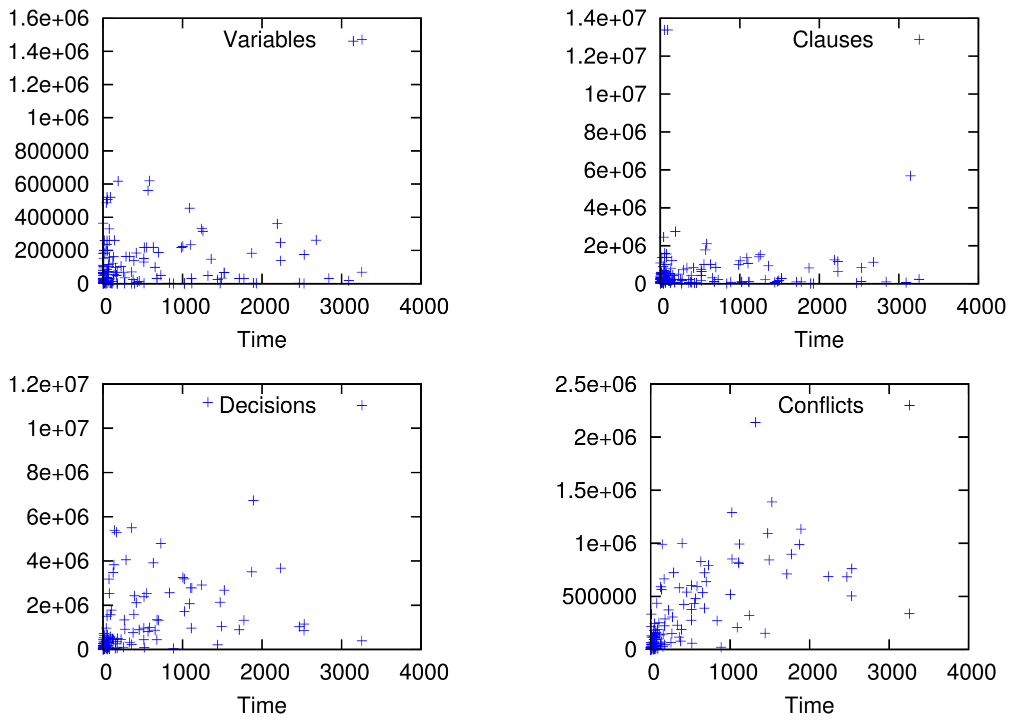


Figure 6: Correlation between solving time and other measures. Each point in the diagram corresponds to an instance of the benchmark. The diagrams show the number of variables, number of clauses, the decisions of the search and the number of conflicts with respect to the solve time of the instance.

the needed time. Thus, a point (x, y) in the diagram states that the solver is able to solve y instances of the benchmark by using less than x seconds per instance. The chosen timeout for the benchmark is set to 3600 seconds. Furthermore, an additional horizontal line label with SAT Race has been added to the diagram. This line represents the timeout of the SAT Race 2010, which has been set to 900 seconds. Looking at the diagram, the best solver according to the solved instances measure is the solver whose line is the most right line of all solvers because this solver is able to solve most of the instances. If a lower timeout is considered, for example the timeout of the SAT Race, then the best performing solver is the solver whose line is the most right line in the diagram on the horizontal line of the according timeout. All the comparisons of SAT solvers in the following sections are based on snake plots that always contain the baseline solver rissR0 [HMS10], which is the solver with its standard configuration. Figure 5 also shows some other points of the development of *riss* and MiniSAT 2.0 [SE09], a solver that is well known in the SAT community. HydraSAT [BGH$^+$09] is the solver that was used as a reference for the development of *riss* and has also been developed at the ICCL. Two versions that have been developed during this work are also presented. The setup *riss* COMB1 is a configuration that combines the well performing settings for the single components of the solver. The configuration *riss* SEL9 is the best configuration for the solver that has been found during the work. The last solver PrecoSAT is the winner of the SAT Competition 2009 and is added to the comparison as a reference.

Usually the user is interested in the total runtime that is needed by the solver to solve a certain instance. Unfortunately, this runtime is hard to predict and therefore lots of instances are solved in order to compare solvers. Figure 6 shows a plot of solved instances combined with measures that can be extracted from the instance or the search of the solver. The top left diagram in the figure contains a dot for every solved instance and the number of variables of the file. In theory, SAT can be solved in a worst case time of $O(2^n)$ if the instance contains n variables. However, the diagram shows that the real runtime is far away from this upper limit. The distribution of the variables of the instances does not correlate with the runtime. The same effect applies for the other chosen static measure, the number of clauses. This measure is printed in the upper right diagram. For the dynamic measurements the plot looks nicer, but still no correlation occurs. The lower left diagram in Figure 6 shows that there is no dependency between the time that is needed to solve an instance and the number of decisions that have been made to find a solution. The lower right diagram shows the found conflicts and the runtime. Again, there is no correlation between these two measures. The four plots exemplary show that the runtime of a SAT solver is not easy to predict and that there is no correlation of the runtime and the number of variables or clauses. Thus, to compare two configurations of a SAT solver, many benchmarks have to be run to be able to make statements about the performance. In the current work these runs are done using an AMD Opteron 285 CPU with a clock frequency of 2.66 GHz. During the runs, the SAT solver was the only application that was run on the CPU. The L2 Cache of this CPU can store 1 MB. The memory limit for solving the instances has been set to 2 GB, because all the computing nodes are equipped only by this amount of main memory.

3. Improving Solver Components

The baseline SAT solver that is used as a reference for the techniques is called rissR0. This solver is a very basic SAT solver that implements all the necessary components but not the most recent ones. This solver is the comparison base for all the experiments. It is also used as a reference to compare the recent development of SAT solvers. Most of the ideas and approaches used in rissR0 are taken from HydraSAT [BGH$^+$09]. HydraSAT is another component based SAT solver that was implemented in 2008 following the algorithms in MiniSAT v1.14.

The execution order of the components in a SAT solver is almost the same as in the CDCL procedure. To avoid unnecessary work, the formula is simplified before the search is started. The preprocessor component is responsible for the simplification. It ensures that obvious simplifications are removed from the formula before it is processed. Since unit clauses are removed by the preprocessor, the first rule of the CDCL procedure that can be applied is the decision rule. Afterwards, the implications of the made decision are propagated. In case of a conflict, the conflict analysis would learn a new clause and the search will jump back in the search tree.

The following sections are structured according to this execution order. At the beginning the major components of a CDCL SAT solver, namely decision heuristic, unit propagation and conflict analysis, are described and recent modifications are presented. Afterwards, the remaining state-of-the-art components, namely the removal heuristic, the restart heuristic and the preprocessor, are described with the choices to improve these components. The last component presented is the preprocessor because it is not really related to the CDCL search. The implementation of the components is done by using rissR0. After the techniques have been presented, the configuration of rissR0 is given. Finally, the old configuration is compared to the possible modifications of the respective component. After all the components have been analyzed separately, the best choices are combined into a single configuration in Section 4 to be further analyzed.

3.1. Decision Heuristic

The decision heuristic is an important component because it decides the part of the search tree that has to be examined in the next step. Whenever no other rule of the DPLL rules can be applied, the decision heuristic makes two decisions.

The first decision is the variable that will be assigned next. Various schemes have been developed in the last years but only one approach seems to be applied to leading SAT solvers. The basic idea comes from the VSIDS heuristic [MMZ$^+$01]. Every variable is related to an activity indicating how often this variable has been involved in recent conflict analysis. The more often the variable has been involved recently, the higher is its activity. The decision heuristic is sometimes also called *Activity Heuristic* because of this feature.

The second decision, that is made by the decision heuristic is to choose the polarity of the picked variable. The decision heuristic could simply choose a literal instead of a variable, but most of the implementations record the activity per variable. Choosing the polarity is done in different ways. One could set the polarity for the variable according to a ratio of positive and negative, use the ratio of the occurrences in the formula or use the last assigned polarity of the variable. Furthermore, one could randomly assign a polarity.

The decision heuristic is essential for finding a solution for a SAT problem fast. If the heuristic always leads to an unsatisfiable subtree before it finally decides to visit the satisfying subtree, the performance of the solver would be low. Thus, current SAT solvers use heuristics that perform well combined with the CDCL approach and use information about conflict analysis to improve the decisions.

3.1.1. VSIDS Implementations

Most of the modern SAT solvers follow a simple scheme to implement the VSIDS heuristic. Every variable is assigned to an activity $a(v)$. Furthermore, the conflict analysis stores an increment value *inc* that is initialized to 1. A parameter *decay* $= 1/0.95$ is set before the search. Updating the activity of a variable is done after the conflict analysis step. The activities of all variables that were touched during the analysis is increased by the current value of the variable *inc*. Afterwards, *inc* is increased by the $decay$ parameter to prefer variables that have been touched in recent conflict analysis steps (*inc* $=$ *inc* $\cdot$ *decay*). The value of the *decay* parameter determines how strong recent conflict analyses are weighted compared to previous ones. The higher its value, the stronger variables participating in recent conflict analysis steps are picked for the next decision.

The original scheme, introduced in Chaff [MMZ$^+$01], stored a score per literal and initialized the activities of the literals according to the occurrence in the original formula. Without the initialization, the very first decision is always the first variable because the priority queue is filled with the variables in ascending order. Modifications of the VSIDS heuristic also increase the activity of a variable if that variable is an element of the learned clause [PD09]. In that case, the activity of this variable might be increased twice per conflict analysis step. Most of the VSIDS implementations also pick the variable randomly with a certain probability. This approach should help the SAT solver to avoid visiting a similar search tree multiple times. Randomly picked variables lead to different branches that lead to another part of the search tree. Picking the variable with the highest activity according to the VSIDS scheme performs well, but the effect of randomly picked variables shows that it is only a heuristic. On the other hand, picking variables only randomly would result in a worse performance of the SAT solver. Therefore, most of the variables are picked heuristically instead of randomly.

3.1.2. Phase Saving

Choosing the polarity of the picked variable can be done by reassigning the previous polarity again, which was unassigned during backjumping. This approach was introduced in [PD07] and is called *phase saving*. The main goal of phase saving is to avoid redoing work. During backjumping, lots of variable assignments get lost and have to be revealed again during search. Keeping these polarities and assigning them again seems to save work for the search and results in finding a solution faster. When the previous polarity of a variable is chosen, this polarity has to be stored before it can be assigned. Therefore an additional data structure, the *backup assignment*, is introduced, which stores these polarities. Whenever a variable is assigned a new polarity by the decision heuristic or during unit propagation, the assignment and the backup assignment of the variable are set to this polarity. During backjumping only the information in the assignment is deleted. Since polarities are only removed from the assignment, but also set in the backup assignment, the backup assignment always contains the most recent polarity that a variable has been assigned to.

To avoid searching in a similar part of the search tree over and over, the backup assignment can be deleted [Bie09]. Using the backup polarity again results in a similar search path after backjumping because all the picked variables are assigned to the same polarity. Only the order of these variables on the path is different. Thus, the visited subtree is similar to the previous one. Only looking in a certain subtree might take a lot of time and does not necessarily result in a solution so that escaping might help to find a solution in another part of the search tree. To analyze this idea, two approaches have been implemented. The first approach deletes the backup assignment for all variables of the conflicting level when a conflict is found. The second approach resets the whole backup assignment when a restart is done. If no backup assignment is stored, because it has been deleted or the variable has never been assigned before, one of the other polarity heuristics, e.g. picking randomly or according to a certain ratio between positive and negative polarities, is applied to assign a polarity.

3.1.3. Jeroslaw-Wang Heuristic

The *Jeroslaw-Wang Heuristic* is an approach to weight polarities. For every occurrence of a literal in a clause of the formula the weight of this literal is increased by the following value. Let l be a literal that occurs in a clause of length n, then the value of this literal will be increased by 2^{-n}. This value is exactly the part of the search tree that is constrained by this clause. The values for all the literals are initialized using the described method before the search is started and they can be updated if a new learned clause is added or after a removal step is applied. If a polarity has to be picked for a decision variable, the polarity with the higher value is chosen emphasizing that choosing this polarity satisfies a set of clauses that prunes the search tree more than using the other polarity and the corresponding clauses.

3.1.4. Choices in rissR0

The reference version rissR0 implements the following configuration. The activity of variables is incremented only if the variable has been touched in a conflict analysis step. Increasing the activity addend is done according to the VSIDS heuristic. The decay is increased by $1/0.95$ after every conflict.

The polarity is chosen according to a ratio between the positive and negative polarities that have been already picked. The implementation sets the polarity for every picked variable to negative. The solver does not initialize the activities according to the occurrences in the formula before searching. Instead, all the activities are initialized to 0. A random variable is picked every 1000 decisions. Such a random step tries to find an unassigned variable using ten attempts. If these attempts fail, because only variables that are already assigned have been picked, the decision variable is chosen by using the activities.

3.1.5. Comparison of Heuristics to Pick a Variable

The two decisions which have been discussed in section 3.1 are analyzed separately. Figure 7 shows the comparison of the presented variants for choosing the next branching variable. The first run *VAR1* increases the activity not only if a variable has been touched in the learning process, but also if the variable is part of the learned clause. The activity addend is the same for both situations. Furthermore, the activity might be increased twice for several variables during one conflict. The second run *VAR2* initializes the activity of the variables. If a variable occurs in a clause of size two, its activity is increased by 0.5. Only clauses of a small size are considered because they cut bigger parts off the search tree. The runs *VAR3*, *VAR4* and *VAR5* pick random variables with a probability instead after a fixed number of decisions. The tested probabilities are 2%, 5% and 10% respectively. The last two runs use different decays. Run *VAR6* uses $1/0.9$ like PicoSAT [Bie08b] and *VAR7* uses $1/0.85$, like the version of *riss* that was used in the SAT Race 2010.

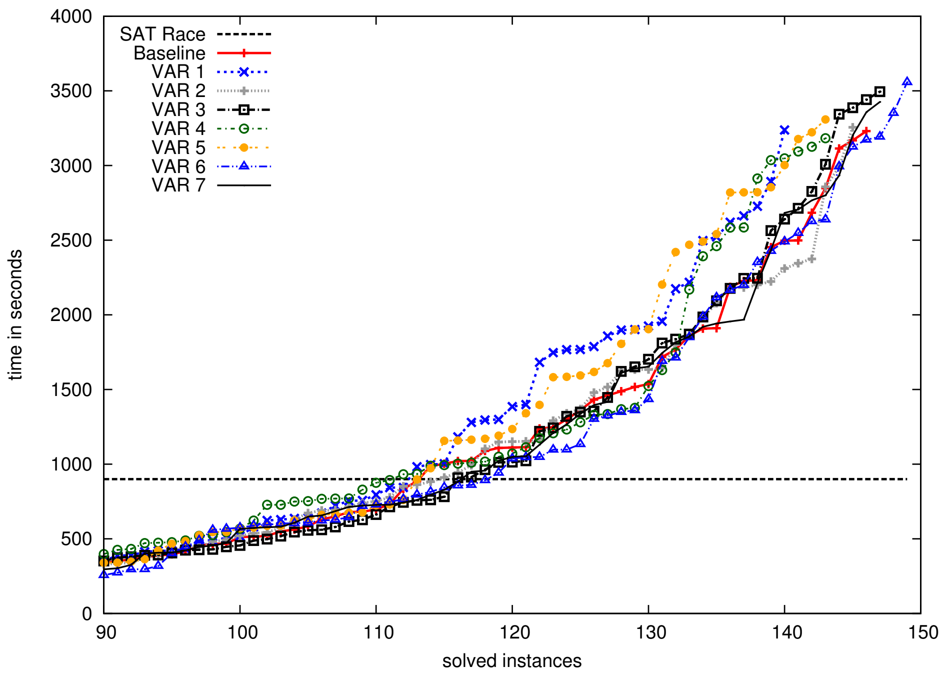


Figure 7: Comparisons of decision heuristic polarity options

The snake plot in Figure 7 shows the performance of these VSIDS implementations. The difference of solved instances between the best configuration *VAR6* and the slowest *VAR1* is only 9 instances. Increasing the activity of literals in the learned clause in *VAR1* or choosing variables randomly with a high probability as in *VAR4* and *VAR5* results in a slower search. Emphasizing recent conflict analysis steps more, as done in *VAR6* and *VAR7*, results in the best configurations for both the SAT Race timeout of 900 seconds and the experiment timeout of 3600 seconds. For the 3600 seconds timeout the best configuration *VAR6* can solve 150 instances.

3.1.6. Comparison of Polarity Heuristics

Figure 8 compares the decisions on choosing the polarity of the picked variable. Run *POL1* assigns the polarity using the phase saving scheme. The second run *POL2* assigns the polarity of the picked variable randomly using a uniform distribution. Run *POL3* assigns the polarity according to the Jeroslaw-Wang heuristic [JW90] of the clauses in the formula and the learned clauses. The last two runs also use the phase saving scheme but reset the backup assignment. *POL4* resets the backup assignment of the conflicting search level and *POL5* resets the whole backup assignment during a restart.

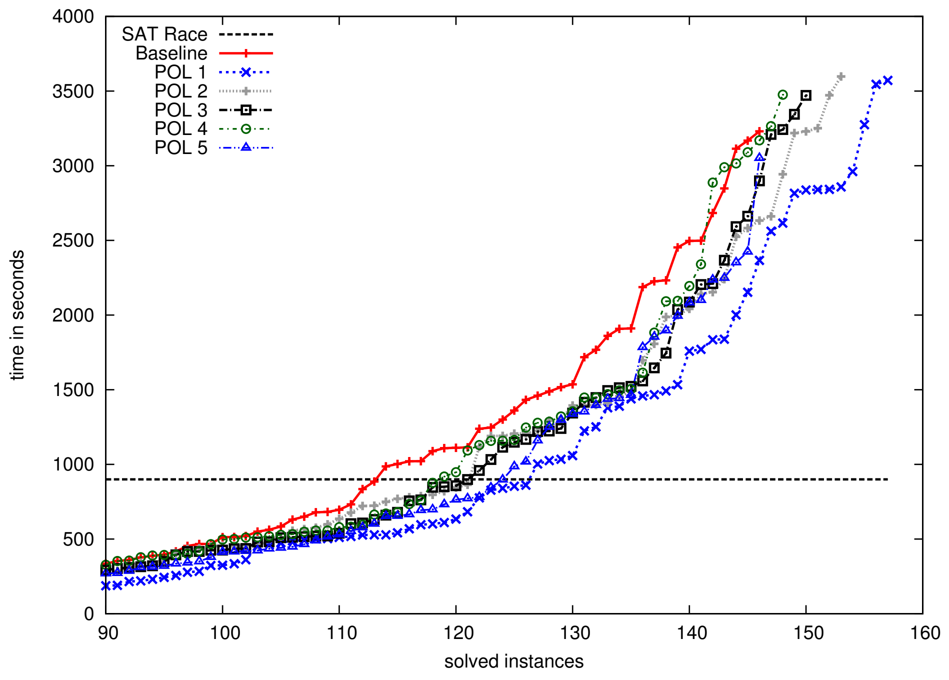


Figure 8: Comparisons of decision heuristic polarity options

The Experiment in Figure 8 shows that choosing a polarity can greatly influence the search. The difference between the best configuration *POL1* and the worst *POL5* is 11 instances. Surprisingly, choosing the polarity randomly in *POL2* performs almost as well as using the best configuration. The performance of the static schemes in *POL3* is clearly higher than the baseline solver. Furthermore, deleting parts of the backup assignment does not improve the performance of the solver but decreases it. Using phase saving without resetting the backup assignment solves 158 instances and performs best. This configuration is also used in most of the state-of-the-art SAT solvers [Bie09, Nik10].

3.2. Unit Propagation

Most of the runtime of a SAT solver is spent during propagating [HMS10]. The propagation is done by the Unit Propagation Component. This component is responsible to find all unit clauses under the current partial assignment and adds the according literals to the partial assignment. Furthermore, the unit propagation states whether the current partial assignment is inconsistent. If an inconsistent assignment is found, the conflict clause is returned and propagating the current implications is stopped (see Figure 4). Otherwise, all the literals of the current unit clauses are propagated until a fix point is reached when no other unit clauses can be found.

3.2.1. Two-Watched-Literal Unit Propagation

The only notable technique that is used in state-of-the-art SAT solvers to propagate the current partial assignment is the Two-Watched-Literal Propagation [MMZ^{+}01]. For this scheme a *unit queue* is used that contains the literals that are implied by the current partial assignment. Initially, this queue is empty. When a decision is made, the according literal is added to the unit queue. Afterwards, all elements of the queue are propagated until the queue is empty or a conflict has been found. Propagating a literal can add more elements to the queue, if adding this literal to the current partial assignment implies other literals. Whenever a literal is added to the unit queue, it is also added to the partial assignment and thus the variable is assigned to the according polarity.

Finding implications of a partial assignment can be done by touching all clauses of the formula during every propagation step. This approach would result in lots of clause visits. Less clause checks can be achieved if only clauses are analyzed that contain the negation of the currently propagated literal. Only these clauses can become a unit clause or a conflict clause because only in these clauses a literal is falsified and removed in theory. All the other clauses that contain the positive literal are satisfied and cannot become conflict or unit.

The Two-Watched-Literal scheme improves this step even more. It uses the fact that a clause can become unit only if one of the last two unassigned literals is falsified. If two literals of a clause are unassigned, the clause is not unit under the current partial assignment. Falsifying the second last unassigned literal in a clause results in a unit or a conflict clause. The algorithm knows that only the other watched literal does not need to be falsified, but it does not know whether this literal is already part of the unit queue. Thus, the state of this literal is checked before enqueuing it to the unit queue. If the literal is unassigned, it cannot be part of the unit queue and thus it is added to the queue and assigned the according polarity. If the literal is already satisfied, it is not necessary to enqueue it again, because it has already been added to the queue. If the second watched literal is falsified, all the literals of the clause are falsified and thus, the current clause is a conflict clause under the current partial assignment. If a literal is propagated, all the clauses in which the negation of this literal is one of the two last unassigned literals have to be visited. To achieve this quickly, watch lists per literal have been introduced. In every clause two unassigned literals, called *watched literals*, are chosen. The clause is added to the watch lists of these two literals. The two watched literals are usually stored at the first two positions in the clause. If another literal of the clause is falsified, the watched literals remain unassigned and the clause cannot become unit or conflict. If one of the watched literals is falsified, the remaining literals of the clause have to be analyzed. If there is another unassigned literal (except the other watched literal), this literal becomes a watched literal and is exchanged with the falsified literal. If there is a satisfied literal, the clause is satisfied and the next clause can be processed. If there is no unassigned literal, the other watched literal has to satisfy the clause. Depending on the assignment of the other watched literal, the clause becomes either a new implication of the current partial assignment or a conflict clause.

3.2.2. Probing

Probing is a technique that is usually used in the preprocessor instead of search. It tries to simplify the formula by comparing the partial assignment that has been created by propagating a variable with positive and negative polarity. Therefore, this step is only applicable if the current partial assignment does not contains any decisions, so that the created assignment represents all the literals that are implied by a single decision with respect to the current formula. During search, probing is only possible if the current level in the search tree is zero. This level can be reached by backjumping or because of a restart. Three interesting cases can occur if the probing approach is followed.

The first result of propagating both polarities can be that one of them might result in a conflict and thus it is clear that the other polarity has to be chosen. Finding that propagating one of the polarities fails is equivalent to finding a unit clause and thus half of the search space can be pruned. For the other two interesting cases the following example is used. It shows the implications of assigning variable *1* to positive and negative.

$$\begin{aligned} 1 &\Rightarrow 2,\ 3,\ 4,\ \neg 5,\ \neg 7 \\ \neg 1 &\Rightarrow 2,\ \neg 4,\ 6,\ 7 \end{aligned}$$

The second effect of probing can be seen looking for variable *2*. To create a complete assignment, variable *1* has to be assigned. However, assigning it to any polarity results in the implication of literal *2*. Thus, a new unit clause is found and the current partial assignment is extended immediately by this literal. The third case can be seen for variable *4* and *7*. Variable *4* is equivalent to variable *1*, because $1 \Rightarrow 4 \wedge \neg 1 \Rightarrow \neg 4 \Leftrightarrow 1 \equiv 4$. The same rule applies to *1* and *¬7*. The found equivalences can be used to reduce the number of variables in the formula by replacing all the equivalent variables by one representative.

3.2.3. Blocking Literals for Unit Propagation

The Two-Watched-Literals unit propagation is usually implemented as shown in Figure 9. As explained in Section 3.2.1, the watch list for literal 2 is processed when this literal is propagated. Every entry of this watch list usually contains only the reference to the clause but no literal. If a clause has to be processed, it is simply accessed and the assignment of its literals is evaluated. In more detail, each of the presented boxes is stored somewhere in the main memory. If a clause is satisfied by one of its literals, it is possible to avoid accessing the clause and checking its literals according to the Two-Watched-Literal scheme. Instead, the possibly satisfied literal can be stored in the watch list. Before a slow clause access is executed, it is checked whether the literal in the watch list entry is satisfied. In this case, the clause access is blocked and the next watch list entry is analyzed. Differently to prefetching the clauses [HMS10], the search path in the search tree can be changed by applying this technique. If the clause is accessed and an unassigned literal is found before

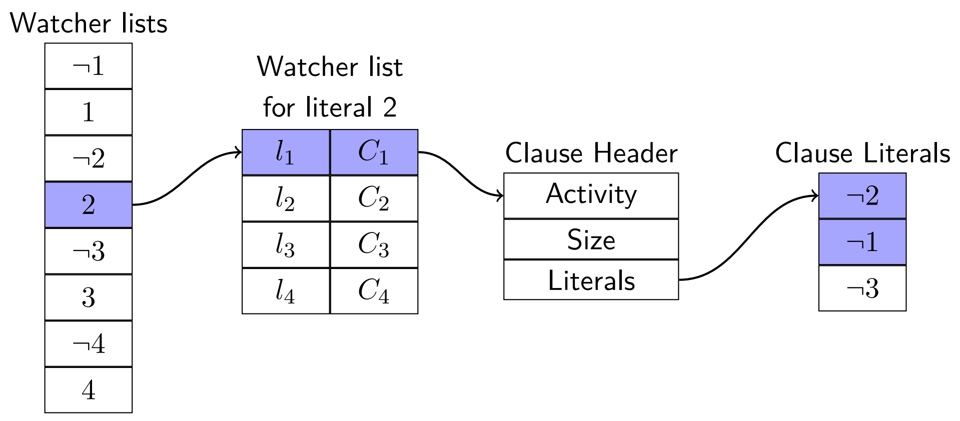


Figure 9: Implementation of the Two-Watched-Literal propagation using Blocking Literals l_1, l_2, l_3 and l_4

the satisfied literal, the order of the literals is changed. During a conflict analysis step this changed order results in a different resolution order and thus another clause might be learned.

Since it is not known in advance which literal will be satisfied first, the blocking literal is chosen statically. The blocking literal can also be updated during searching, but there is not any well performing heuristic known.

3.2.4. Choices in rissR0

The baseline solver implements the Two-Watched-Literal propagation. It furthermore treats binary clauses specially as introduced in [ES04]. Whenever a literal is propagated, all the binary clauses are checked and the transitive closure of the implications on the binary clauses is calculated and added to the queue. Afterwards, the literal is propagated using longer clauses. The propagation stops on the first conflict that has been found or if a fix point is reached. The solver rissR0 does not implement probing or blocking literals.

3.2.5. Comparison of Heuristics to Run Unit Propagation

Figure 10 shows the performance of three modifications on rissR0. The first run *UP1* implements probing. Whenever a literal is propagated on the root of the search tree, its negation is propagated first and checked for a conflict. Afterwards, the literal is propagated and the approaches in Section 3.2.2 are followed. Equivalences that are found are not used to simplify the formula. *UP2* prefers longer conflict clauses [Bie09]. Therefore, whenever a conflict during propagating binary clauses is found, the same partial assignment is propagated on the longer clauses. If a conflict is found among the longer clauses, this conflict is returned. Otherwise, the binary conflict is used for conflict analysis. The third run *UP3* implements the Blocking Literal scheme. The literal used for blocking the clause access is the first literal of the clause when the clause is added to the watch lists. The comparison of the three runs with the baseline solver in Figure 10 shows that

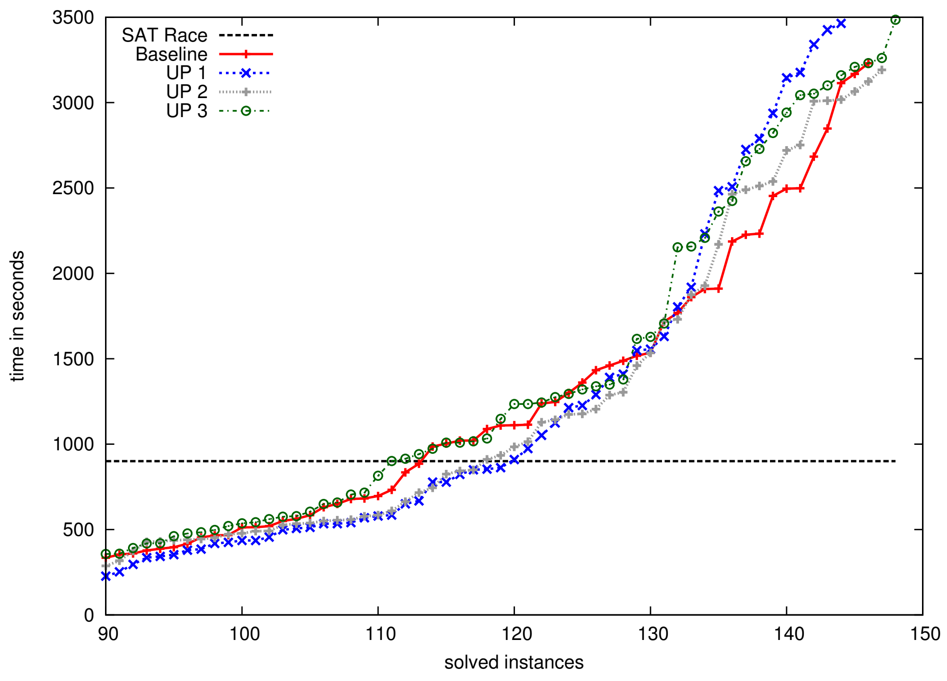


Figure 10: Comparisons of unit propagation approaches

preferring long conflict (*UP2*) and probing(*UP1*) seems to be good for short timeouts like the one for the SAT Race. If the timeout is set to 900 seconds, the difference of solved instances between these two configurations and the baseline solver is almost 10 instances. For longer run times the performance of *UP1* drops below all other approaches. Using blocking literals (*UP3*) during the search is the best configuration for very long timeouts and can solve 149 instances. Still, the best configuration among the four runs is hard to determine because the difference of solved instances within the timeout is only four instances.

3.3. Conflict Analysis

Analyzing the conflict is the main difference between the DPLL and the CDCL procedure. The advanced performance of the CDCL procedure comes with using the information of the conflict by performing backjumping and improving the information for the decision heuristic. Usually, learning a new clause is done by applying resolution to the conflict clause and the reason clauses of its literals [SS96]. After some resolution steps are applied, the resolvent is added to the formula (see section 2.2.4). Since the conflict clause and the resolvent, called learned clause, are both unsatisfied under the current assignment, some of the literals of the partial assignment have to be unassigned and backjumping is applied in the search tree. Usually the backjumping distance is chosen so that the newly learned clause becomes a unit clause and a unit propagation step can be applied instead of deciding the next variable.

3.3.1. First UIP Conflict Analysis

A *Unique Implication Point* (UIP) is a literal on a search path in the search tree that leads to a conflict. Together with the partial assignment before the UIP, the UIP implies the conflict. Thus, any literal on the current search path that implies both polarities of a variable is a UIP. During resolution a UIP is reached if the resolvent contains only a single literal of the conflicting decision level. The *first UIP* [ZMM01] is reached when only a single variable of the current level is found for the first time and when the resolution variables are taken from the trail reversely.. Implementing the resolution can be done efficiently [Rya04]. Not every resolution step is done separately. Only the necessary information is stored and processed. Furthermore, all the resolution steps share a single set of literals for the resolvent.

Original Formula:

$F = \langle\ C_1,\ C_2,\ C_3,\ C_4,\ C_5\ \rangle$

$= \langle$ [1, 3], [¬2, ¬5, ¬6], [¬1, ¬4, 6], [¬1,¬2,¬4, 5], [¬1, 2] $\rangle$

Current Search Path:

Assignment	1	2	3	4	5	6
Level	1	1	2	3	3	3
Reason	-	C_5	-	-	C_4	C_3

Conflict Analysis Resolution Steps:

Step	Operation	Result	Remarks
1.	$C_2 \otimes C_3$	[¬1, ¬2, ¬**4**, ¬**5**]	C_6
2.	$C_6 \otimes C_4$	[¬1, ¬2, ¬**4**]	C_7, learned clause, subsumes C_4
3.	$C_7 \otimes C_5$	[¬1, ¬**4**]	C_8, minimized clause

Resulting Formula: $F = \langle\ C_1,\ C_2,\ C_3,\ C_7,\ C_5,\ C_8\ \rangle$

Figure 11: Applying a state-of-the-art conflict analysis step

Conflict analysis can be illustrated using the running example as shown in Figure 11. In the current state all variables are assigned true. Under this assignment, clause C_2 is the conflict clause. According to the learning scheme, the trail is processed reversely and thus C_2 is resolved with clause C_3, resulting in the resolvent C_6. Since this clause still contains more than one variable of the conflicting decision level, the reason clause C_4 belonging to the next literal on the trail, namely literal 5, is resolved with C_6 in the second resolution step. The resulting resolvent C_7 is the learned clause according to the first UIP scheme because it contains only a single literal of the current level.

3.3.2. Conflict Minimization

Sometimes a learned clause can be minimized further. The minimization is achieved by applying self subsuming resolution, also called *local minimization*, or by applying recursive minimization [SB09]. Therefore, the learned clause is resolved with the reason clauses of its literals. If the size of the resolvent is greater than the size of the learned clause, the learned clause is kept. Otherwise the resolvent is used as the learned clause and the minimization is tested with the next literal.

Another techniques to determine whether a literal can be removed from the current learned clause is called *recursive minimization*. This minimization applies several resolution steps. Every literal l_i of the current learned clause $C = [l_1, \dots, l_n]$ is tested whether it is redundant. Let l_i be the literal that is currently tried to be removed. The first step is to resolve the current clause C with the reason clause for l_i. If new literals are added during a minimization step, these literals are resolved next. If a resolution step adds new literals from a new decision level, the literal l_i is not redundant and thus kept in the clause. If the size of a resolvent is smaller than the current learned clause then the current resolvent is kept as the learned clause. Properties of the minimized clause are that it is shorter than the learned clause and the minimized clause contains literals from the same number of decision levels as the learned clause. Local minimization can be seen as a special case of the recursive minimization, whereas only a single resolution step is necessary to remove a certain variable.

The local minimization can be seen in the example in Figure 11. Continuing the example in section 3.3.1 the learned clause can be minimized further. Therefore, it is checked whether the resolvent with any reason clause of a contained literal is smaller than the learned clause. The resolution of C_7 and the reason clause of literal 2, namely C_5, results in a smaller clause C_8. This clause is added to the formula instead of C_7.

3.3.3. On-the-Fly Self Subsumption

Conflict analysis uses the fact that adding resolvents to the formula does not change the model of the formula as shown in section 2.1.3. If a resolvent subsumes a clause of the formula, then this clause can be removed because, if the resolvent is satisfied, all clauses it subsumes have to be satisfied. Detecting whether a resolvent $C_r = C_1 \otimes_v C_2$ subsumes one of the two originating clauses can be done by using the size of the clauses. C_r subsumes C_1 if the size of C_r is exactly one element less than the size of C_1. This case occurs only if all literals of C_2 except v also occur in C_1. In this case, the size of the resolvent is less than the size of the larger participating clause. The only literal that does not occur in C_r, but in C_1, is a literal of variable v. Thus, C_r subsumes C_1. Keeping the shorter resolvent might increase the performance of the search because a shorter clause prunes the search tree more than a longer one. Applying this method has been introduced in [HS09].

The conflict analysis example in Figure 11 contains such an on-the-fly subsumption. The first UIP clause C_7 subsumes one of the two clauses involved in the resolution, namely C_4. Thus, C_4 can be replaced by this intermediate resolvent.

3.3.4. Assignment Stack Shrinking

After the conflict is analyzed and the learned clause is added to the clause database, backjumping is performed. Usually the backjump distance in the search tree is adjusted so that the learned clause becomes a unit clause under the new partial assignment leading to unit propagation. As proposed in [NR10], jumping further might help to escape from hard subtrees with no solution. This idea is realized by assignment stack shrinking, a technique that dynamically determines the backjumping level.

Assignment Stack Shrinking uses three heuristics and works as follows: After every conflict analysis, shrinking is applied if a certain condition, the *shrinking condition*, is fulfilled. If the shrinking condition is fulfilled, the literals in the learned clause are sorted according to a *sorting scheme*. Then the solver backtracks to the *shrinking backtrack level*. The next decisions that have to be made by the solver are done according to the ordered literals of the learned clause. The value of the picked literal is assigned false. After each assignment unit propagation follows as usual.

Algorithm 3 UpdateShrinkSize(x)

```
1: Initialize $y \leftarrow 95$;
2: mean ← mean of last $y$ learned clause lengths;
3: stdev ← standard deviation of last $y$ learned clause lengths;
4: center ← $mean + 0.5 \times stdev$;
5: ulimit ← $mean + stdev$;
6: if $x \geq center$ then
7:     $x \leftarrow x - 5$;
8: else
9:     $x \leftarrow x + 5$;
10: end if
11: if $x > ulimit$ then
12:     $x \leftarrow ulimit$;
13: end if
14: if $x < 5$ then
15:     $x \leftarrow 5$;
16: end if
```

The first shrinking condition that has been developed in [Nad04] is fulfilled if all the literals in the learned clause are from a different decision level. Another condition is the clause size exceeding a certain threshold x as in zchaff_rand [MFM04]. The threshold has been set dynamically, using statistics of the search, as explained in Algorithm 3. The first implementation in zchaff_rand of this dynamical setting initialized the parameter y to 600. Whenever and update is done,the algorithm calculates a new upper limit and a center value based on the mean and the standard deviation of the clause lengths of the last learned clauses. If the current threshold is below the calculated center value, the threshold is increased by 5, otherwise it is decreased by 5. The new value of the size threshold is bounded by a dynamic upper limit and a lower limit of 5. Besides the introduced conditions, another condition has been introduced in [NR10] that does not allow two shrinking steps in a row. Instead of counting the number of literals in a clause, the number of levels are counted and a threshold for this value is applied.

The literals of the learned clauses are sorted before they are reassigned again. Assigning these literals in the same order again would lead to a similar search path as if the backjump level is be set to the 1UIP level. If a different order is chosen, the order of the literals on the path is different and thus the order of resolution steps using the according reason clauses is different as well, resulting in a different learned clause. Sorting the literals of the learned clause can be done according to the decision levels of the literals, ascending or descending. Furthermore, the activity of the variables can be used to sort these variables before they are assigned again during the next decision steps. The second sorting scheme is motivated by the good performance of the VSIDS heuristic (compare section 3.1.1) and performs better than the first one according to [NR10] as the activity of a variable seems to be a better criterion than the level it has been assigned.

Choosing a lower shrinking backjumping level than the level where the learned clause becomes unit can be done in several ways. Published variants include unassigning the smallest part of the trail such that all literals of the current learned clause become unassigned or unassigning all levels of the current partial assignment until the first level that is not part of the learned clause [NR10].

3.3.5. Choices in rissR0

The baseline solver rissR0 implements the usual conflict analysis combined with conflict minimization. Although the minimization can be implemented recursively the standard implementation does not use this fact. Neither on-the-fly self subsumption nor assignment stack shrinking are used in rissR0.

3.3.6. Comparison of Several Conflict Analysis Techniques

Figure 12 compares several conflict analysis techniques. The first run *CON1* does not try to minimize the learned clause further. *CON2* applies the on-the-fly self subsumption during conflict analysis. Due to the component based structure of the solver, the subsumed clause is not removed from the data structures. Instead, the new subsuming clause is added to the formula. The clauses that can be subsumed are kept and thus introduce an overhead. Thus, the impact on the runtime of this configuration might be better, if the subsumed clauses were removed immediately. *CON3* and *CON4* apply different assignment stack shrinking configurations. *CON3* sets the backjumping level so that all literals of the learned clause are unassigned again. Shrinking is only applied if the learned clause contains only one literal per decision level and if its size is above a certain threshold, namely 95. The size threshold changes according to Algorithm 3. Assigning the literals of a conflict clause again is done according to the activity of the literals. *CON4* sets the backjumping level to the closest level in the search tree where none of the literals in the learned clause has been set. Shrinking is applied if the learned clause has more than 95 literals. To reassign, the literals of the currently learned clause are sorted ascending accordingly to their decision level. The last configuration *CON5* uses a recursive implementation of the recursive minimization as implemented in MiniSAT 2.2.

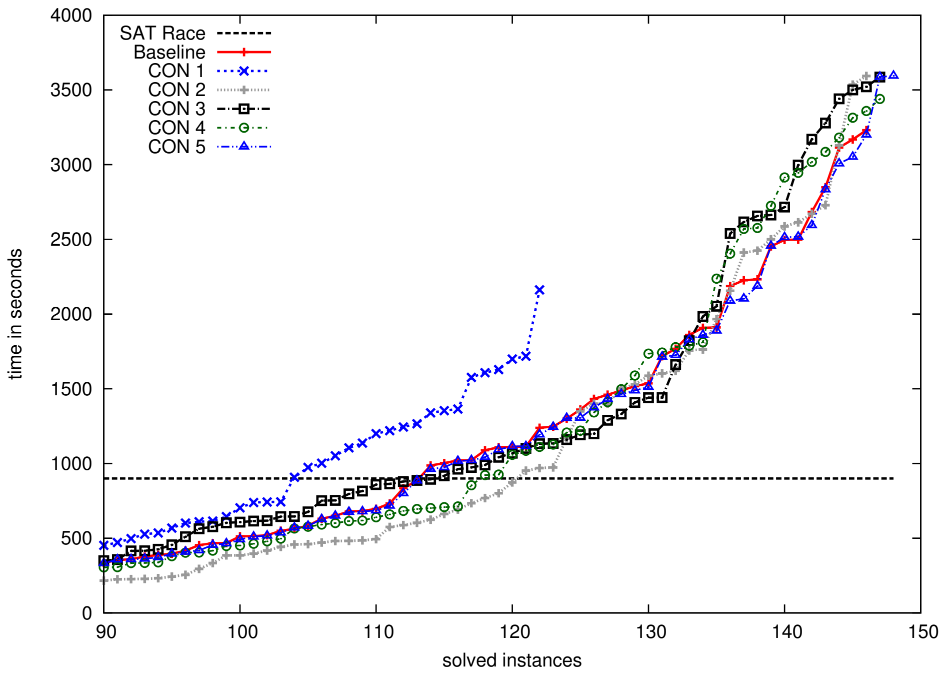


Figure 12: Comparisons of conflict analysis strategies

The result of the minimization is always the same as in the iterative implementation in rissR0. Thus, the search path of the two configurations is always the same.

Minimizing the learned clause is crucial for the performance of a SAT solver. This is the major result of comparing the configurations to the baseline solver. The baseline solver can solve 147 instances in the limits, whereas it is able to solve only 123 instances if the minimization is disabled. The snake plot in Figure 12 shows this very nicely. The performance of the solver is not influenced significantly if the minimization is implemented recursively instead of iteratively. Without minimization the solver can compete with all the other configurations only within the easiest 60 instances. Solving harder instances takes significantly more time if conflicts are not minimized. Applying assignment stack shrinking to the conflict analysis does not improve the performance of the solver significantly. Shrinking the assignment leads to a better performance than the baseline solver only for the 900 seconds timeout of the SAT Race. Applying on-the-fly self subsumption during conflict analysis results in the best performance for the SAT Race timeout.

3.4. Removal

Analyzing the conflict clause and conflict directed backjumping boosted the performance of DPLL search algorithms dramatically [SS96]. However, as mentioned in [SS96], the clause database has to be managed, because too many clauses slow down the performance of unit propagation [ES04]. Therefore, the clause database is separated into two sets of clauses, the original formula and the set of learned clauses. All state-of-the-art SAT solvers reduce the learned clauses according to some *removal heuristic* during the search to keep the performance of unit propagation considerably high. Most systems always keep clauses of size two because they prune large parts of the search tree and because propagating binary clauses is very cheap. Furthermore, a clause that is a reason for a current assignment of a variable cannot be removed because otherwise resolution on that particular variable fails during conflict analysis. The point in the search when a removal is scheduled is also chosen heuristically by a *removal event heuristic*.

Differently from the DPLL search, the CDCL approach needs the learned clauses to remember which part of the search tree it has already examined. Removing a learned clause allows the search to reach the same search path again that has been evaluated when the learned clause was generated. If the clause removal is not restricted, the search will visit the same conflicts again and again and will not terminate. To ensure termination, removals are scheduled in increasing intervals. If an interval is so large that all possible learned clauses can be generated within that interval, the search will terminate again.

3.4.1. Activity Removal

MiniSAT [ES04] stores one activity per clause. This activity is computed as the activity for the variables in the activity heuristic (compare section 3.1.1). The activity of a clause increases if the clause is touched during conflict analysis. Again, there is a current increment value $\mathit{inc} = 1$ and a decay value $\mathit{decay} > 1$. When a clause has been used in a resolution step in the current conflict analysis, its activity is increased by the current value of *inc*. After all touched clause activities have been updated, the increment value is increased by the value of the decay parameter $\mathit{inc} = \mathit{inc} \cdot \mathit{decay}$. To remove learned clauses, the clauses are ordered according to their activity. The clauses with the lower activity are removed because they have not been important during recent conflict analyses and so they might not be useful in the current search tree. The percentage of the learned clauses that is removed is another parameter that can be tuned.

3.4.2. Literals Blocks Distance (LBD)

Another activity for clauses has been introduced in [AS09b]. This value is called Literals Blocks Distance (LBD). It is computed as follows. The literals of a clause are partitioned into sets where all literals of a set have been assigned on the same decision level. The number of the sets is the LBD. The LBD for a clause can be set immediately after the clause has been generated during conflict analysis or during propagating the last unassigned literal of this clause. It has been shown in [AS09b] that clauses with a low LBD value are very important for conflict analysis steps. Thus, during removal, clauses with a low LBD are kept. Clauses with a LBD$= 2$ are never removed because a learned clause with this LBD value assigns all its literals on a single level after backjumping. The percentage of the learned clauses that are removed is a parameter.

3.4.3. Suffix Removal

To determine how important a certain clause might be for the current state of the search can be done by using activities as in the previous two sections. Another indicator of this importance might be the number of assigned or unassigned literals in a clause given the current partial assignment.

This approach is implemented in HydraSAT [BGH+09]. The literals of the learned clause are sorted according to their level. In the next step they are divided into a prefix and a suffix. Either the number of suffix or prefix literals can be fixed and the remaining literals belong to the other part. The level of the last prefix literal is stored and the clause is enqueued to a list related to that level. Therefore, per visited decision level a list, the *level-list* of clauses has to be stored. If the clause contains only prefix literals or is smaller than a certain threshold, it is stored in another data structure, the *prefix-list*. Important clauses, for example clauses of size two will never be removed and thus are enqueued to a third structure, the *kept-list*.

When a removal is scheduled, the current partial assignment is usually not empty and the current state of the search is not on the root level of the search tree. Let *level* be the current level of the search, then all clauses that are enqueued in level-lists at higher levels are removed. Furthermore, clauses from the prefix-list can be removed. Since new clauses are added at the end of the prefix-list, an aging mechanism can be applied by removing clauses from the front of the list. This aging mechanism ensures that the average age of the kept learned clauses does not increase significantly. Since new clauses have a low age and previously added clauses have a higher age, clauses that have been learned recently are kept and the older ones are removed. This aging keeps clauses that might be important in the current sub tree.

3.4.4. Dynamic Scheduling

Scheduling a clause removal can be done statically after a certain number of conflicts and added clauses or dynamically, depending on the size of the formula and other features of the search. MiniSAT 1.4 schedules a removal if the ratio between learned clauses and clauses of the formula reaches a certain threshold. At a restart (see section 3.5), the *factor* ratio is increased. Initially, the ratio is set to $1/3$. At every restart, it is increased by $\textit{factor} = \textit{factor} \cdot 1.1$. A removal is scheduled if the following equation is satisfied: $\textit{learned\_clauses} - \textit{assigned\_variables} \geq \textit{original\_clauses} \cdot \textit{factor}$.

3.4.5. Static Scheduling

The winning solver *glucose* of the UNSAT track of the SAT Competition 2009 was the first SAT solver that implemented the LBD as clause activity combined with a static removal scheduling. This removal follows a linear scheme depending on x, the number of removals that have already been scheduled. A removal is scheduled, if the number of conflicts since the last removal is greater than $20000 + 500 \cdot x$.

Another approach that is followed in the baseline solver rissR0 schedules removals according to a geometric series. The first removal is triggered after $\textit{limit} = 100$ conflicts. When a removal is scheduled, this limit is increased by $\textit{limit} = \textit{limit} \cdot 1.5$ so that after the next *limit* conflicts the next removal is scheduled.

3.4.6. Choices in rissR0

The baseline solver rissR0 uses the static geometric scheduling (compare section 3.4.5). Furthermore, it does not use any clause activity scheme to remove clauses but uses the clause size. All learned clauses with more than six literals are removed from the oldest 55% percent of the clauses. This behavior occurs because the removal is scheduled together with a restart and thus the suffix removal is not able to handle prefixes or suffixes of clauses. Thus, only the aging mechanism of the removal is applied.

3.4.7. Comparison of Removal Heuristics

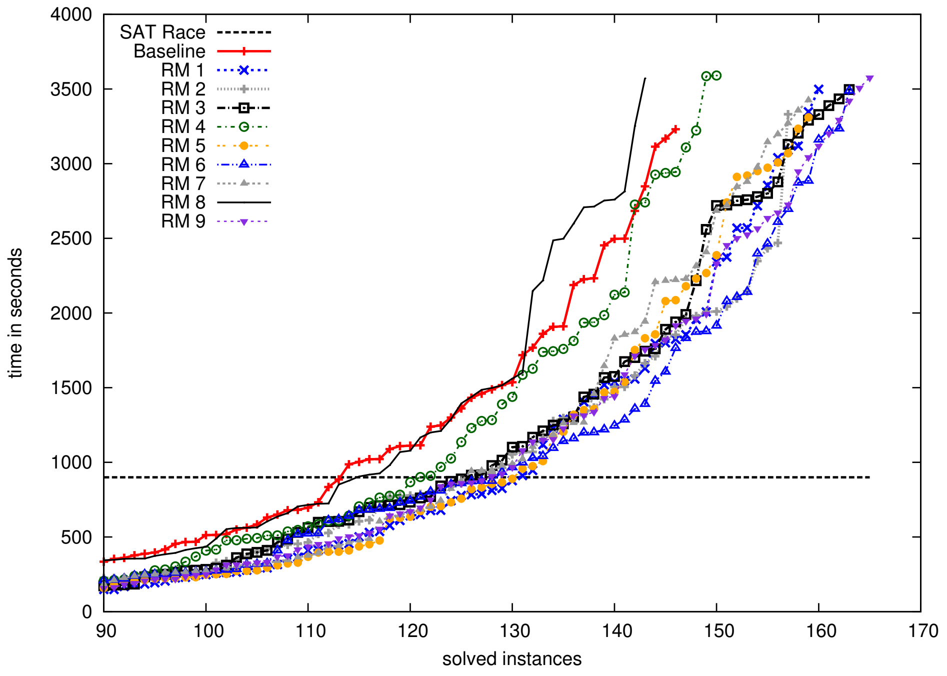


Figure 13: Comparisons of removal heuristic options

Figure 13 shows the runtime for some removal configurations. *RM1* implements the LBD removal with removing 50% of the clauses with the highest LBD value. The LBD value is set per clause only when the clause is generated because of the component based structure of *riss*. Updating the activity during unit propagation would result in a more complex structure of the solver. Furthermore, PrecoSAT [Bie09] implements an additional step into the removal. This SAT solver does not allow to remove learned clauses that had been learned directly before a restart. Whenever the activity removal is chosen, this scheme is also applied. *RM2* behaves like *RM1* except that scheduling a restart is done by using the dynamic scheduling. *RM3* implements the MiniSAT activity removal, combined with the standard geometric scheduling. *RM4* also uses the MiniSAT activity and also schedules removals according to MiniSATs dynamic scheduling. *RM5* uses the same configuration as *RM1* except the treatment of learned clauses that had been generated directly before a restart. Theses learned clauses are not treated specifically in *RM5*. Since all presented configurations keep 50% of the learned clauses and analyze only the remaining ones when a removal is scheduled, the next two runs analyze this behavior for two different values. *RM6* keeps only 25% of the learned clauses with the highest activity during a removal step and *RM7* keeps 75% of these clauses. Longer intervals between two removals are tested in the last two runs. *RM8* implements the same configuration as the baseline solver except that the initial limit for the geometric series that is used to schedule removals is set to 1000 instead of 100. *RM9* also extends the initial limit to 1000 but uses the activity removal as in *RM1*.

The major result of the comparison in Figure 13 is that all removals that are based on the LBD activity removal perform better than the aging removal implemented in the solver. Furthermore, configuration *RM4* with the dynamic removal scheduling and a dynamic clause activity performs very badly. The snake plot also shows that all the configurations that use the LBD activity perform better than the other configurations. Using the MiniSAT activity removal (*RM3*) seems to be the better choice compared to the static aging removal in rissR0. When it comes to the LBD value, the static geometric scheduling performs slightly better than the dynamic scheduling. Increasing the intervals for the removal using the LBD as activity results in the best configuration, which can solve 166 instances.

3.5. Restarting the Search

The backtracking SAT algorithm DPLL is a deterministic algorithm to find a solution for a SAT instance. This algorithm is known to behave heavy-tailed [GSCK00]. The heavy-tailed behavior leads to a very long run time for a certain instance although the instance can be solved very fast if another random seed is used. Therefore, restarting the search randomly is proposed in [GSCK00] to avoid the high variance of the runtime and to boost the average solving performance of SAT solver.

Restarting the search is done by unassigning the whole partial assignment (except found unit clauses). Afterwards, the decision rule is applied again and can pick a variable with a high activity. The effect of a restart is that the complete search tree is turned over and variables that had been assigned at very low levels become unassigned again. Restarting also helps to escape of subtrees that are difficult to escape from by backjumping. After a restart, the search starts again at the root of the search tree and can choose any variable. Thus, the probability that a different subtree is examined next is higher if restarts are enabled.

During the last years the frequency of restarts increased. It has been shown in [Hua07] that applying restarts according to any schedule performs better than not applying restarts. Instead of scheduling restarts randomly, they are triggered by a static schedule in recent SAT solver. As described in [Bie08b], the high frequency of restarts helps to find the solution instead of getting lost in a subtree that does not contain a solution or refutation. Combined with the phase saving polarity heuristic (see section 3.1.2), restarts only change the order of the variable assignments in the search tree instead of leading to different assignments and different subtrees. Whenever the next conflict is found in the similar subtree, a different conflict clause will be generated because the order of the variables on the lower level than the conflicting level have a different order. This conflict clause will lead to another subtree. Thus, for the subtree that has been examined another conflict clause is learned before the search escapes from this tree and visits another subtree. If the partial assignment is undone during the search, the whole algorithm might not terminate because a restart could be scheduled before a solution is found. Thus, most of the restart schedules have an increasing intervals so that the intervals become so large that all possible assignments can be tested between two restarts and thus the search always terminates.

In the following section, frequently used restart policies are introduced. Most of them follow a static schedule. The last one strategy uses some features of the current search state to determine whether the search should escape from the current state and a restart should be applied.

3.5.1. Geometric Series Scheduling

The first schedule that will be presented follows a geometric series and has been implemented in MiniSAT 1.4 [ES04]. The solver initializes the size of the first interval with *limit* $= 100$. When the number of occurred conflicts reaches the value of *limit*, a restart is triggered and the interval size is increased by *limit* $=$ *limit* $\cdot 1.5$. After the next *limit* conflicts the next restart is scheduled.

3.5.2. Luby Series Scheduling

The Luby series has been introduced in [LSZ93] and used for scheduling restarts the first time in [Hua07]. The limits for the restart intervals starts as follows: 1 1 2 1 1 2 4 1 1 2 4 8 1 1 2 $\dots$. Since it is very improbable to find a solution after a single conflict, this series is multiplied by a factor. This factor is a parameter of the Luby series schedule. For search algorithms without information about the search state, the Luby series is proved to be optimal in [LSZ93]. Thus, it is be the optimal choice if no information about the search is available and it is expected that this policy performs better than the geometric series if restarts do not interfere with the behavior of other components in the search.

3.5.3. Nested Series

The intervals of the geometric series increase very fast compared to the Luby series (compare Figure 14). The intervals become so large that it is hard for the solver to escape from hard subtrees after the first small restart intervals. Nesting two series can avoid this problem and has been implemented in PicoSAT [Bie08b]. The solver now maintains an inner limit and an outer limit. The outer series bounds the inner series. Whenever the inner limit i reaches the outer limit o, the inner series starts from its beginning and o is set to the next value of its series. The interval size for the number of conflicts between two restarts is represented by i. Whenever the number of conflicts since the last restart reaches the value of i, a restart is triggered and i is set to the next value of its series or reset because it has reached its bound o.

This procedure ensures two properties. The average size of the intervals grows such that the search still terminates. On the other hand, small intervals occur much more often than in one of the two used series.

3.5.4. Dynamic Scheduling

Scheduling restarts is mostly done depending on the number of conflicts. In the static schedules, a certain number is determined and after this number of conflicts a restart is scheduled. Scheduling restarts in [AS09a] depends not only on the number of conflicts, but the decision level is also taken into account to adjust the restart schedule to the current search state. Although the Luby series has been proved to be the best series for restarting a search without information, even better restart schedules can be created by using the provided information of the current search state.

Glucose [AS09a] watches the decision levels of the decisions that were made during the last 100 conflicts. The specified number of conflicts to average the decision level is called *windowSize*. The solver maintains the average *recent* of the decision levels. If decreasing of the average of this decision levels stalls, a restart is scheduled. To recognize this stalling, the global average *global* of all decision levels is maintained as well. If the value of *recent* gets below 70% of *global*, a restart is scheduled. Since $+$the value for *recent* has to be calculated after a restart again, the interval size between two restarts is at least as large as the window size for the calculation. Since the window size is not increased, termination cannot be guaranteed for this schedule.

3.5.5. Reject Scheduled Restarts Using Agility

Scheduling restarts dynamically was considered difficult until an effective adaptive way to reject scheduled restarts was found [Bie08a]. The used measurement to decide whether a restart should really be scheduled is decided upon the agility of the search. This agility is calculated by using the phase saving polarity heuristic (see section 3.1.2).

The agility is initialized with $\mathit{agility} = 0$. Whenever a polarity is assigned to a variable , this agility is updated. If the new polarity is different from the backup assignment of this variable, the agility should increase, if not, it should decrease. To achieve aging at every variable assignment, the activity is decreased by a decay $\mathit{agility} = \mathit{agility} \cdot d$ where d is a value between 0 and 1. Increasing the activity is done by using the same decay $\mathit{agility} = \mathit{agility} + d$. Thus, it is ensured that the value of *agility* is bounded by 0 and 1. If a restart should be scheduled by a static restart heuristic, the current agility can be compared to a threshold. If the agility is higher than the threshold, the restart is not scheduled. In [Bie08a] it is assumed that the solver might find a refutation if most of the variable assignments have a different polarity. If the agility is low, the restart is scheduled to help the solver escaping from the current subtree.

3.5.6. Choices in rissR0

The baseline solver implements a simple static restart schedule. As in MiniSAT 1.14, the first restart is scheduled after 100 conflicts and the increase factor for the series is set to 1.5.

3.5.7. Comparison of Restart Schedules

Figure 14 compares the restart intervals for a set of static restart schedules. Note the log scale on the y-axis that represents the interval size between two restarts. The x-value describes the number of the restart. A purely geometric schedule clearly increases the interval size exponentially. Modern SAT solver use rapid restarts using the Luby series or a nested geometric series. The plot shows that the intervals for a nested geometric series are smaller than the ones for the Luby series. PicoSAT uses a nested geometric series with an increase factor of 1.1 as shown in the plot. It schedules restarts with the highest frequency among the compared heuristics. Skipping some of this scheduled restarts leads to a performance boost again according to empirical results [Bie08a].

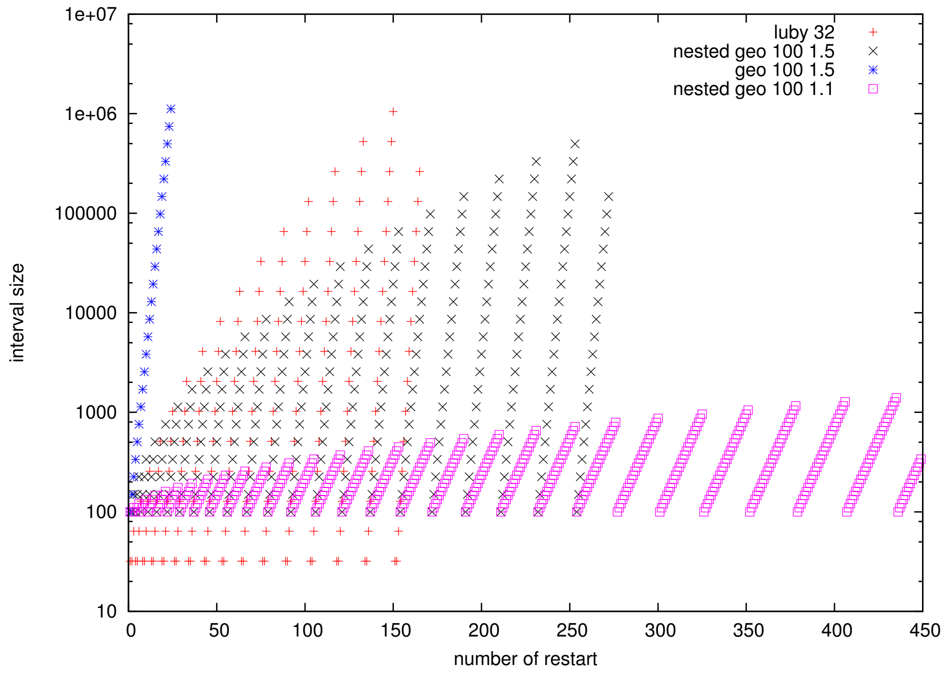


Figure 14: Comparisons of restart schedules

3.5.8. Comparison of Restart Heuristics

In Figure 15 restart schedules are compared. The first four runs *RES1*, *RES2*, *RES3* and *RES4* use a Luby series with the bases 2, 32, 64 and 1024, respectively. *RES5* and *RES6* use the dynamic scheduling with different window sizes. *RES5* uses a window size of 100 conflicts like glucose and *RES6* uses 1000 conflicts. The remaining two runs implement the ideas from PicoSAT. *RES7* restarts using two nested geometric series both with a base of 100 and an increment factor of 1.1. *RES8* enables the phase saving polarity heuristic to be able to calculate the agility of the current search state. The configuration rejects restarts if the measured agility of the search is lower than 22%.

Figure 15 clearly states that scheduling restarts often boosts the performance of the

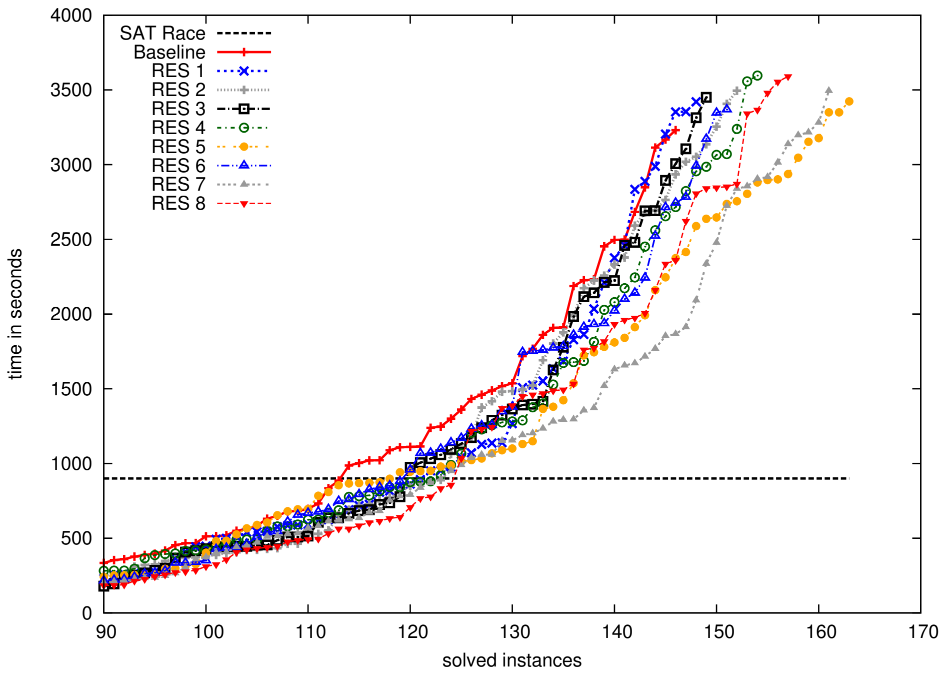


Figure 15: Comparisons of restart heuristics

SAT solver. The two schedules that schedule restarts most often are clearly the fastest configurations. The dynamic schedule in *RES5* schedules restarts after at least 100 conflicts and avoids scheduling restarts if the search seems to escape of a certain subtree because the decisions level decreases. This configuration is able to solve 166 instances of the benchmark. Enlarging the window size to 1000 conflicts (*RES6*) already results in a slow search. Using the nested schedule of two geometric series (*RES7*) is also quite powerful and the average distance between two restarts is still lower than 1000. In case of the Luby series, it can be seen that this series highly depends on the chosen factor by which it is multiplied. Choosing a large factor results in large interval sizes, even for the small numbers in this series. When too small values are chosen as factor, the Luby series schedules restarts too fast. For example for factor two, whenever the series starts from 1 again, a restart is scheduled after two conflicts although it is very improbable that the solver is in a difficult subtree after only two conflicts(*RES1*).

The comparison of the schedules shows that rapid restart are very powerful. Still, scheduling restarts too frequently slows the SAT solver down again, since the solver is not able to examine subtrees well enough before the next restart is scheduled. Furthermore, the dynamic schedule or the restart rejection does not outperform static schedules.

3.6. Preprocessor

The general SAT problem is NP complete [Coo71]. For a given formula with n variables, all known algorithms have a worst case execution time of $O(2^n)$. This execution time depends only on the number of variables and not on the number of clauses. In theory, reducing the number of variables results in a faster search according to this complexity. However, in practice the number of variables does not correlate with the runtime. The number of clauses highly influences the performance of the unit propagation and thus also the frequency of learning new clauses and performing backjumping. Thus, reducing the number of variables by increasing the number of clauses does not necessarily boost the performance of the SAT solver but slows it down. Preprocessing helps to reduce the size of the formula by removing variables and clauses that are not necessary for finding a model for the formula. There are different techniques that can be separated into two groups. The first group of techniques keeps the model and that is why these techniques are called *model preserving*. The other group does not keep the model and is called *non-model preserving*.

Simple techniques like propagating unit clauses or applying subsumption and self subsumption are model preserving [HJB10] and are not described in detail here. Nevertheless, they are part of any modern preprocessor [EB05] and are applied to any formula to remove redundant clauses and literals.

The following sections describe recently developed preprocessing techniques that reduce the size of the formula. The first two techniques in the following sections are non-model preserving whereas all the other presented techniques are model preserving.

3.6.1. Variable Elimination

Variable Elimination (VE) [EB05, SP05] is a technique to remove variables from the formula. After a variable is removed, only a partial assignment is left and thus this technique is non-model preserving. VE is based on resolution and thus it can still be used to trace the refutation of an instance, namely keeping track of the resolution steps.

Removing a variable from a formula is done by resolving the according clauses in which the variable occurs. Given a formula with a set of clauses S, this set contains clauses where the variable x occurs positive C_x and negative $C_{\overline{x}}$. Let F be the union of these two sets $F \equiv C_x \cup C_{\overline{x}}$. Resolving these two sets on variable x results in a new set of clauses F' where trivial clauses are not added to the new clause set. It is shown in [EB05] that F can be replaced by F' without changing the satisfiability of the formula. If the model is needed for the original formula, then the partial model can be extended using the original clauses F to assign variable x. Usually, applying VE to a variable results in a larger number of clauses. Thus, in state-of-the-art preprocessors VE is only applied to a variable, if the number of clauses does not increase.

The example in Figure 16 shows the application of VE to a formula. The resulting formula has fewer clauses than the original one and contains fewer variables. Solving the simplified formula F_{VE1} without knowing the original formula might result in a satisfying assignment like $\alpha = \{1, 2, \neg 3, 4, 5\}$. Obviously, this assignment does not satisfy the original formula F because clause C_4 is not satisfied. By using all clauses where the variable 1 occurs, the assignment for variable 1 can be set to the needed value without changing any other variable assignments. Simply flipping the assignment to the negative polarity results in a satisfying assignment for the original formula.

3.6.2. Blocked Clause Elimination

Blocked clauses are redundant clauses. However, removing a blocked clause from the formula does not keep the assignment and thus blocked clause elimination (BCE) is non-model preserving [HJB10]. A clause C is blocked if it contains a blocking literal l. The literal l is a blocking literal, if $\bar{l}$ is part of C, or for each clause $C' \in F$ with $\bar{l} \in C'$ the resolvent $C \otimes_l C'$ is a tautology [JBH10, HJB10]. Removing blocked clauses can be done until a fix point is reached. The order of the removal does not change the result because BCE is confluent [JBH10]. Not removing all blocked clauses from the formula does not seem to influence the runtime of the solver much so that BCE is not performed for literals with a high occurrence.

Original Formula:

F=$\langle$ C_1, C_2, C_3, C_4, C_5, C_6 $\rangle$
=$\langle$ [1, 2], [1, 3, 4], [¬1, ¬3], [¬1, ¬5], [¬3, ¬4], [¬2, 4] $\rangle$

VE on F using variable **1**, remove tautology $C_2 \otimes_1 C_3$=[3, ¬3, 4]:

F_{VE1}=$\langle$ $C_{1\otimes3}$, $C_{1\otimes4}$, $C_{2\otimes4}$, C_5, C_6 $\rangle$
=$\langle$ [2, ¬3], [2, ¬5], [3, 4, ¬5], [¬3, ¬4], [¬2, 4] $\rangle$

BCE on F using C_2 with blocking literal **3** removes C_2:

$$F_{BCE} = \langle\ C_1, C_3, C_4, C_5, C_6\ \rangle$$

HTE on F using C_2:

Step	Operation	Remark
1.	HLA(C_2,C_1)=[1, ¬2, 3, 4]	C_2'
2.	HLA(C_2',C_6)=[1, ¬2, 3, 4, ¬4]	C_2'', tautology
3.	remove C_2	

$$F_{HTE} = \langle\ C_1, C_3, C_4, C_5, C_6\ \rangle$$

Figure 16: Preprocessing a formula

Reducing the original formula F in the example in Figure 16 by using BCE results in the formula F_{BCE}. Unfortunately, this reduced formula does not have a model that does not satisfy F as well (compare section 3.6.3). Thus, it does not show that it is possible to repair the model using the blocked clause. However, the example still shows that it is possible to remove redundant clauses by applying BCE to a formula.

3.6.3. Hidden Tautology Elimination

The Hidden Tautology Elimination (HTE) is based on a clause extension, the hidden literal addition (HLA). After the clause is extended by HLA, it is checked whether it is a tautology. If it is a tautology, the clause can be removed from the formula. HTE is model preserving [HJB10]. The HLA applied to a clause C with respect to a formula F is computed as follows: Let l be a literal of C and $[l', l] \in F \setminus \{C\}$. If such a literal l' can be found, C is extended by $C := C \cup \overline{l'}$. This extension is applied until fix point. HTE now removes an extended clause, if it is a tautology. Note that applying HLA or HTE to a formula is model preserving [HJB10]. HLA is the opposite operation of self subsuming resolution. An example HTE simplification is given in Figure 16. Firstly, in step 1 clause C_2 is extended by literal ¬2 using C_1 because both clauses contain literal 1. Afterwards, C_2' can be extended by using the newly added literal and C_6. The latter step results in C_2'', a tautology. According to the features of HTE, this clause can be removed from F still keeping all satisfying models for the original formula. This fact also shows why there is not any assignment for F_{BCE} that does not satisfy the original formula F.

3.6.4. Equivalence Elimination

According to the complexity theory, a SAT problem is solvable in exponential time with respect to the variables in the formula. Reducing the number of variables to speed up the search can be done by removing equivalent literals and only keeping one representative of the equivalence class. An equivalence of literals can be found by finding cycles in the binary implication graph. To visualize the implications of a formula, a *binary implication graph* can be generated by using all the binary clauses of the formula. The vertexes of the graph are all literals of the formula F. The edges of the graph show which literals l' are implied by another literal l. Since a binary clause represents an implication, for all binary clauses $C = [l, l']$ the following two edges can be added to the graph: $\overline{l} \to l'$ and $\overline{l'} \to l$. Assuming the cycle $a \to b \to c \to a$ has been found in the graph, then there have to be edges $a \to b$, $b \to c$ and $c \to a$ resulting from the clauses [$\overline{a}$, b], [$\overline{b}$, c], and [$\overline{c}$, a]. These clauses also force the following cycle $\overline{a} \to \overline{c} \to \overline{b} \to \overline{a}$ to exist in the graph. Given these implications, it can be shown that $a \equiv b$ for the given clause set. The definition of equivalence is $a \equiv b \Leftrightarrow a \to b \wedge \overline{a} \to \overline{b}$. Given the first implication, the left hand side of the assumption can be shown. The second implication gives the right hand side. This proof sketch can be adjusted for any two literals in the cycle resulting in the statement that all participating literals in the cycle are equivalent.

Another way of finding equivalences is to use probing and comparing assignments, as already discussed in Section 3.2.2. How probing is implemented in the preprocessor is described in the following section.

3.6.5. Probing

Probing is a technique to simplify the formula by propagating several single decisions, for example l, separately and comparing their implications BCP(l) [LMS03]. Since *riss* is component based, the preprocessor has to implement its own unit propagation. Currently, the unit propagation of the preprocessor in *riss* is only able to propagate assignments on binary clauses.

The preprocessor is able to apply the two rules explained in 3.2.2. The found necessary assignments are used to simplify the formula and equivalences are eliminated. The rule to find necessary assignments is called *failed literal probing*. Another rule to find necessary assignments, the *clause probing*, is implemented: For a clause $[l, l'] \in F$ all the literals BCP(l) $\cap$ BCP(l') are necessary assignments. The rule can be explained by the fact that the clause $[l, l']$ can only be satisfied by satisfying at least one of its literals. If both literals of the clause imply a same literal, this literal has to be set to true for any satisfying assignment of the formula. This rule can be extended to longer clauses.

Furthermore, as explained in [LMS03], binary clauses can be added to the formula. Whenever a decision l is propagated, for all literals l' that are implied by l new binary clauses $[\bar{l}, l']$ can be added to the formula. Naturally, not too many clauses should be added because the higher number of clauses slows down the search.

3.6.6. Vivification (Asymmetric Branching)

Vivification is a technique to reduce the length of clauses. Replacing a long clause by a shorter one results in more pruning of the search space and during search also to smaller learned clauses. Thus, small clauses should be more beneficial for the search than longer clauses. Reducing the length of a clause $C = [l_1, \dots, l_n]$ of a formula F can be done as follows: All the literals are sequentially propagated in ascending order of the index until one of the following three cases occurs.

1. BCP($\{\overline{l_1}, \dots, \overline{l_i}\}$) results in an empty clause for $i < n$.
2. BCP($\{\overline{l_1}, \dots, \overline{l_i}\}$) implies another literal l_j of the C with $i < j < n$
3. BCP($\{\overline{l_1}, \dots, \overline{l_i}\}$) implies another negated literal $\overline{l_j}$ of the C with $i < j \leq n$

In the first case, the assignment that unsatisfies the formula F is not allowed. Disallowing this assignment can be done by adding a clause $C' = [l_1, \dots, l_i]$. The clause C' is clearly shorter than C and also subsumes C. Thus, the longer clause C can be removed by the newly created clause C'.

Propagating the literals of C sequentially on the formula F resulting in the second case can also be understood by the constraint $\overline{l_1} \wedge \dots \wedge \overline{l_i} \rightarrow l_j$. Converting this constraint into a clause (compare Definition 13) results in $C' = [l_1, \dots, l_i, l_j]$. This clause subsumes C and thus C can be replaced by the shorter clause.

In the third case, the extracted relation is the following with respect to the formula F: $\overline{l_1} \wedge \dots \wedge \overline{l_i} \rightarrow \overline{l_j}$. This constraint can also be formulated in a clause $C' = [l_1, \dots, l_i, \overline{l_j}]$. Applying self subsumption to $C'' = C \otimes_{l_j} C' = [l - 1, \dots, l_{j-1}, l_{j+1}, \dots, l_n]$ results in a shorter clause C'' that subsumes C.

3.6.7. Inprocessor Simplifications During Search

The preprocessor techniques are based on the structure of the formula F. If one of these techniques is applied until a fix point, it is not possible to reduce the formula further by using the same technique. However, during the search unit clauses might be learned or found by probing. Applying this unit clause to the formula has the following consequences:

1. Satisfied clauses are removed from the formula.
2. Clauses are shortened because the unsatisfied literals can be removed.

The implications of the unit clause can also lead to more reduction according to the two rules. Thus, if a top level unit is found during the search and the search reaches level zero (for example during a restart), all formula simplifications can be applied again before the search will be continued [EB05].

3.6.8. Choices in rissR0

The preprocessor implemented in rissR0 uses three techniques. In the first step, pure literals are detected and set to true. Afterwards, variable elimination is applied. Since the implementation of the Satelite preprocessor [EB05] tightly couples subsumption and variable elimination, these two algorithms are also run tightly coupled in rissR0. Firstly, all subsumed clauses are removed. Afterwards, VE is run, using the variables of the formula in descending ordered according to the number of their occurrence in the formula. These two techniques are repeated until no more clauses can be subsumed and no more variables can be eliminated without increasing the number of clauses. Finally, all found unit clauses are propagated. Differently to the implementation of Satelite, propagating unit clauses is not done during subsumption or variable elimination.

3.6.9. Comparison of Preprocessing Techniques

Figure 17 shows the performance of preprocessing techniques combined with rissR0. To evaluate the preprocessing techniques, a new preprocessor is implemented that is able to run all the presented techniques separately. It is named *Coprocessor* because it is also able to simplify the formula during a restart. Applying unit propagation to the formula for probing or asymmetric branching is done by using only the binary clauses of the formula. This decision is a trade off between finding all implied unit clauses and running a fast unit propagation. Due to the fact that VE is not confluent, applying VE with this preprocessor results in a different formula than running the old preprocessor. Therefore, the first run *PP1* enables the same techniques as used in rissR0 but using a different implementation. This run is the baseline for all the other configurations and thus VE is executed in all the following configurations at least once. *PP2* applies BCE before VE is run on the formula. Running BCE exhaustively is expensive [JBH10] so that two limits have been chosen. BCE is performed only if the clause that is analyzed is smaller than 4 literals and if the literal does not occur in more than 20 clauses. These limits are set for all configurations that use BCE. *PP3* implements the combination of BCE and VE as well but repeats this combination a second time, which is the best combination of VE and BCE in [JBH10]. Vivification is used in *PP4*. Again, not all clauses can be checked for redundant literals in reasonable time so that only the largest 90% of the clauses are analyzed. *PP5* runs HTE after VE has been executed. HTE is also limited to the 10% most frequent variables.

The probing techniques have been analyzed in the configurations *PP6*, *PP7* and *PP8*. These configurations enable failed literal probing, clause probing and failed literal probing including creating binary clauses, respectively. After probing has been applied, clauses that can be subsumed are removed. Again, limits have been set to keep the runtime of the preprocessor reasonable. Failed literal probing is only executed for the 20 most frequent variables. Clause probing is done for clauses of size 3 and only for the 4 most

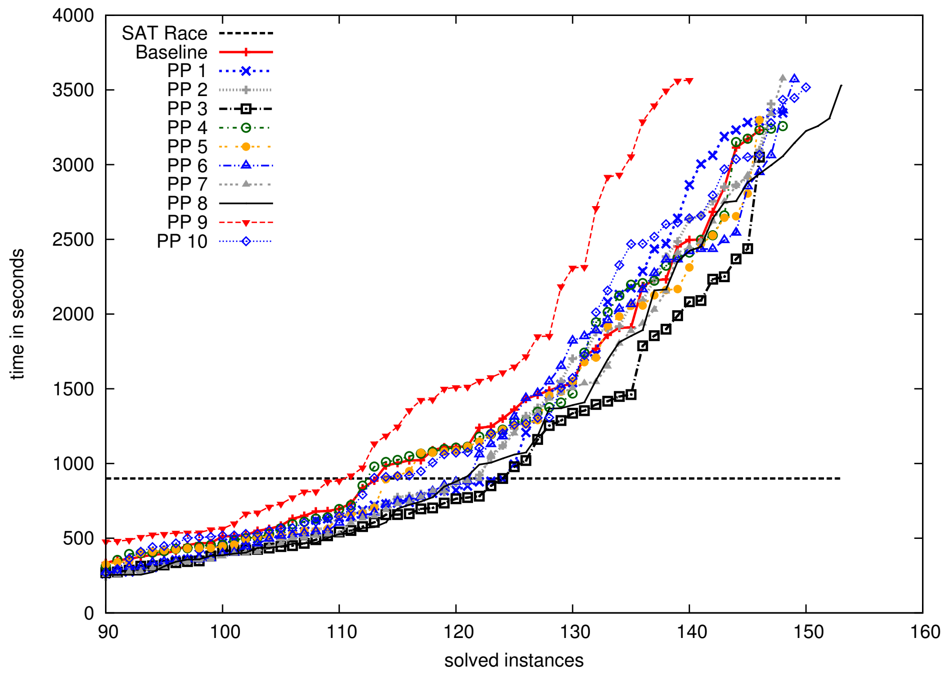


Figure 17: Comparisons of preprocessing algorithms.

frequent variables. *PP9* disables the preprocessor completely so that the search works on the original formula. Configuration *PP10* is an extension of *PP1* and enables the unit propagation during VE again. Thus, the effect between the standard implementation and the original algorithm in [EB05] can be compared.

In general, finding good parameters for the preprocessing techniques is hard because the techniques involved provide lots of parameters that can be tuned to preprocess a certain set of formulas very well. Since parameter tuning is not part of this work, finding good settings will be part of future work.

The major result of Figure 17 is that preprocessing is important for modern SAT solvers because less instances can be solved by *PP9* than by any other configuration. Furthermore, none of the presented techniques outperforms the VE implementation of rissR0. By applying unit propagation during the VE (*PP10*) as proposed in the original algorithm, only four additional instances can be solved. The vivification in *PP4* has almost the same performance as rissR0 and is only able to solve one more instance. Repeating BCE and VE in *PP3* solves the same number of instances as rissR0 but faster. Applying probing results in more solved instances. Creating binary clauses during probing boosts the overall performance the most and enables the solver to solve 154 instances. This result emphasizes that short clauses are more important for the search than longer ones.

3.6.10. Comparison of Inprocessor Techniques

Analyzing the effect of preprocessing techniques that are applied during search is done in Figure 18. Again, the used preprocessor is called Coprocessor because the preprocessor of rissR0 is not able to simplify the formula during a restart. Simplifications are not applied during all restarts. The number of new literals on the root level is counted. Only if the difference of this number between the last simplification and the current value is greater than two, a simplification is triggered. The same limits as in section 3.6.9 are applied for all the used techniques.

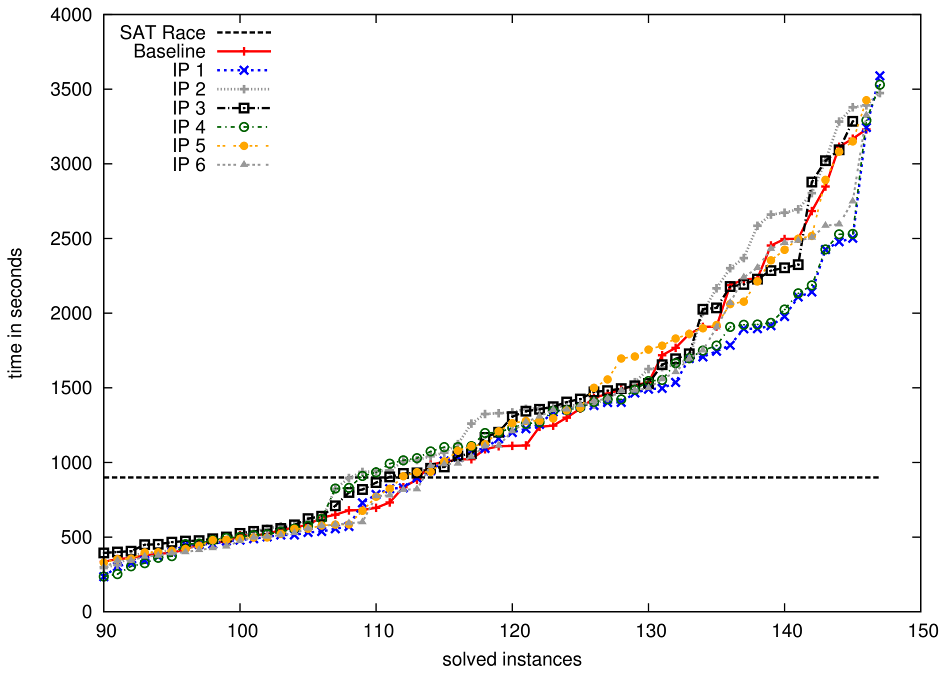


Figure 18: Comparisons of inprocessor algorithms

IP1 applies subsumption during a restart. Furthermore, satisfied clauses are removed from the formula and unsatisfied literals are removed from the remaining clauses. All the following configurations are based on *IP1*. Eliminating variables during a restart is done in *IP2*. *IP3* removes blocked clauses during simplification. *IP4* applies asymmetric branching. *IP5* combines all the previous configurations and applies VE, BCE and asymmetric branching during a restart. *IP6* tries to remove hidden tautologies if the formula is simplified during a restart.

The comparison in Figure 18 shows that none of the presented configurations speeds up the search significantly. Sometimes a configuration can solve a set of instances faster, for example *IP1* and *IP4* for the timeout of 2000 to 2500 seconds. Still, the overall performance of the SAT solver is not improved significantly by applying simplifications during the search. This effect might also be introduced by the component-based structure of *riss*. If the formula has to be simplified, the preprocessor has to be initialized again firstly and afterwards the structures of the search have to be filled with the simplified formula again. Thus, moving the formula from one component to the other can be another source for the missing performance boost.

4. Results

In the following two sections the combinations that seemed to have a big impact on the performance are combined and analyzed. In Section 4.1, some of the most powerful configurations for the single components have been picked and combined to analyze the interference of the performance of the components. In Section 4.2, the effect of disabling a well performing configuration for a single component, while all the other components keep their powerful setting, is analyzed.

4.1. Combined Runs

To find a good performing configuration for all the components of the solver, the most powerful configurations per component have been chosen and have been combined to analyze the effect of the combination. Figure 19 shows this comparison.

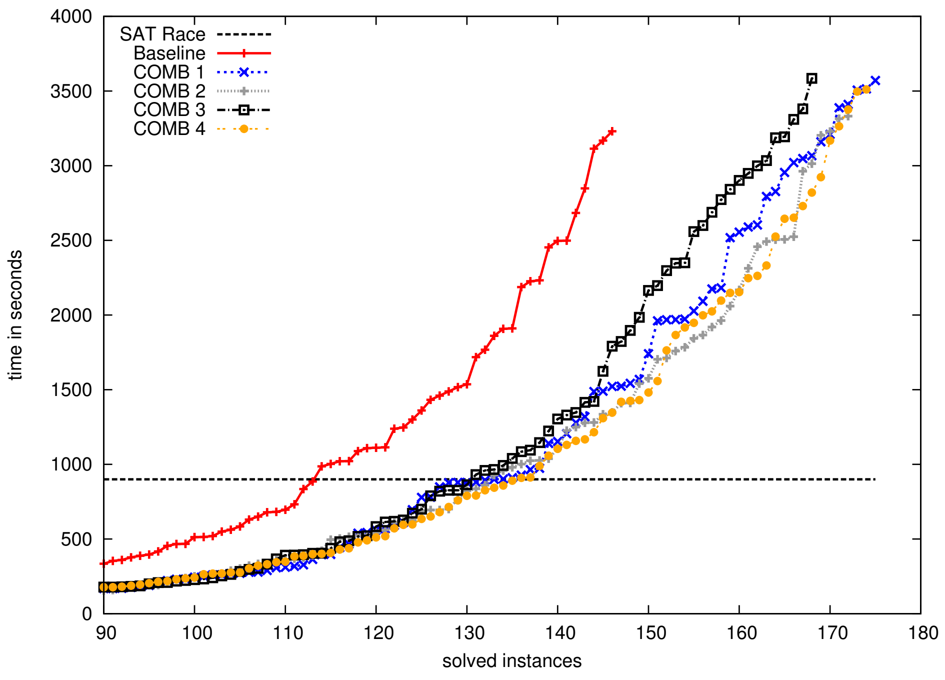


Figure 19: Combinations of winning components

The compared configurations are similar because they are based on a major configuration. Thus, this major configuration is described before the differences of the configurations are explained. The decision heuristic of the major combination uses a *decay* $= 1/0.9$ (*VAR6*) and uses phase saving without any resetting for picking a polarity (*POL1*). The used unit propagation uses blocking literals (*UP3*). The conflict analysis applies on-the-fly self subsumption during the analysis to shorten clauses (*CON2*). The removal heuristic uses the LBD value as activity and always keeps the 25% most active clauses when a removal is scheduled (*RM6*). Removals are scheduled according to the geometric series with an initial limit of 100 and an increase factor of 1.5. Scheduling restarts is done according to the dynamic scheduling with a window size of 100 (*RES5*). The preprocessor that is used in the major configuration implements the Blocked Clause Elimination, Variable Elimination, Failed Literal Probing with the creation of binary clauses of the found implications and asymmetric branching. These techniques are applied in exactly this order before the search is started. During a restart, the formula is simplified by asymmetric branching (*IP4*).

The first configuration *COMB1* that is shown in Figure 19 implements exactly the major configuration that is described in the previous paragraph. *COMB2* implements a different restart scheduling. Instead of the dynamic scheduling, two geometric series with an initial limit of 100 and an increase factor of 1.1 have been used. The remaining two configurations implement a different removal scheduling. The initial limit of the geometric series has been increased to 1000 (*RM8*) to see the influence of a frequent removal on a state-of-the-art solver. *COMB3* corresponds to the major configuration (*COMB1*) and *COMB4* corresponds to *COMB2*, except for the changed removal schedule.

Figure 19 shows that combining the well performing components increases the performance of the SAT solver significantly. The baseline solver was able to solve 147 instances whereas the best configuration is able to solve 175 instances. The relative improvement is 20%. Concerning the benchmark, *COMB1* can solve 10% more than the baseline solver. However, let x_{comp} be the performance difference between the baseline solver and the picked configuration for component $comp$. The performance difference between the baseline solver and the major configuration *COMB1* is not the sum of all the differences x_{comp} for all the components. This result shows that the configurations of the components interfere with each other concerning their performance. It is hard to determine the best configuration for a single component if the configuration for the remaining components is not fixed. The comparison in Figure 19 underlines this fact. The two configurations *COMB1* and *COMB3* share the same restart scheduling and implement a different removal strategy. The performance of the configuration with the dynamic restart schedule is higher than with nested scheme, namely 175 compared to 170 solved instances. However, if a different removal schedule is chosen as in *COMB2* and *COMB4*, the relation of the performance between these two configurations with respect to the restart scheduling behaves inversely. *COMB2* with dynamic restarts is able to solve only 172 instances, whereas *COMB4* with nested restarts can solve 176 instances. The leading configuration among the picked opportunities is *COMB4* with 176 solved instances. Since the number of possible configurations is quite high and the current comparison shows that not combining the best single components can result in a better performance, no more combinations of the components of the previous section have been analyzed.

4.2. Looking for Beneficial Components

In the previous Section 4.1, it has been shown that the performances of two components in a SAT solver interfere with each other. Therefore the major configuration has been analyzed further. For the seven choices that have been presented in Section 3, the components in the major configuration have been set back to the standard configuration of the baseline solver. Thus, the impact of switching a single component from well performing

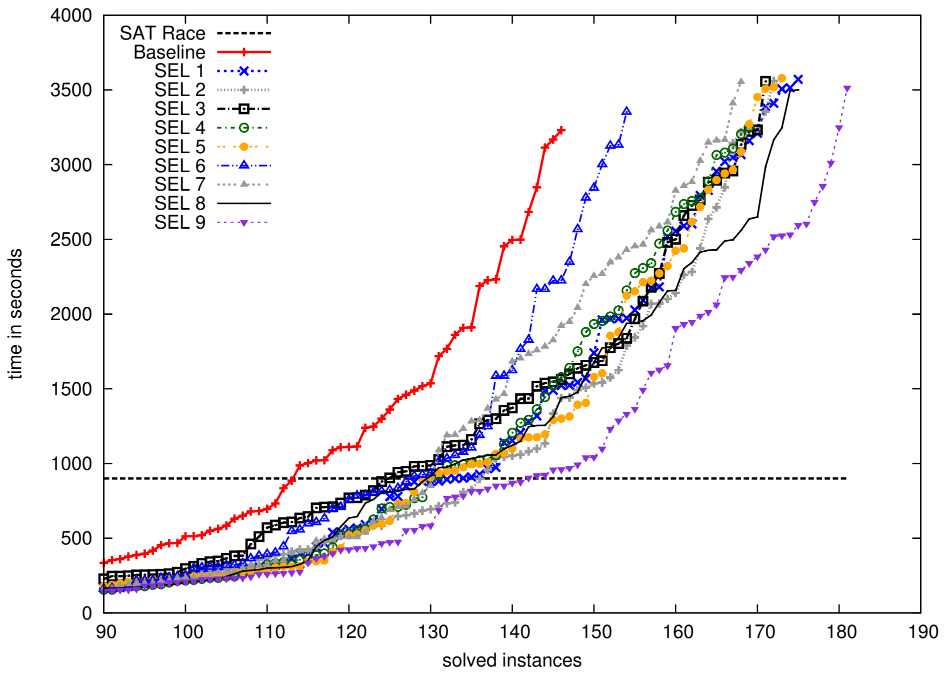


Figure 20: Comparison of combination except single components

back to standard can be analyzed.

The first configuration *SEL1* of the comparison in Figure 20 implements the major configuration *COMB1* of Section 4.1. All the remaining configurations implement the major configuration except a single component. *SEL2* resets the variable heuristic. Therefore, the decay value is set to $1/0.95$. *SEL3* sets the polarity of the picked variable always to negative instead of using phase saving. *SEL4* does not use blocking literals during unit propagation. The configuration *SEL5* does not use on-the-fly self subsumption during conflict analysis to shorten clauses. *SEL6* removes all clauses that contain more than 6 literals and 55% of the remaining learned clauses during a removal that is scheduled according to a geometric series with an initial limit of 100 and an increase value of 1.5. *SEL7* schedules restarts according to a geometric series with an increment factor of 1.5 and initial limit set to 100 instead of dynamically scheduling. *SEL8* uses the standard implementation of Variable Elimination as a preprocessor. Finally, *SEL9* does not use any simplification technique during a restart.

Figure 20 shows the modifications of the major configuration. It shows that resetting a single component back to its standard configuration still performs better than the baseline solver. The modification with the worst performance, *SEL6*, still solves 154 instances. Furthermore, the major configuration *SEL1* has a higher performance than any other modification, except the configurations that modify the preprocessor, namely *SEL8* and *SEL9*. Thus, the configuration of all the components is important to reach a high performance system. Furthermore, it can be seen that setting a single component back to its standard configuration does not lead to a worse SAT solver than setting all components back. There is not any component that is responsible for the overall performance of the SAT solver. Some configurations can solve more instances within a lower timeout than 3600 seconds. Still, combining all well performing configurations leads to the best search procedure. The highest impact on the performance can be seen by the removal strategy. Using the standard removal results in 21 instances that cannot be solved because the upper memory limit is reached. The major configuration cannot solve two instances because of the same reason. Thus, a good removal strategy is important to keep the used memory of a SAT solver low. Scheduling restarts has the second highest impact on the solver performance. If the geometric schedule with fast increasing interval sizes is chosen (*SEL7*), the solver is able to solve only 170 instances. The influence of picking a variable and polarity, allying unit propagation or on-the-fly self subsumption introduces only a variance of 5 instances compared to the major configuration.

Surprisingly, configuration *SEL9* solves significantly more instances than the major configuration, namely 183 instances in total. The chosen simplifications in *SEL1* remove satisfied clauses, falsified literals and try to shorten remaining long clauses by applying asymmetric branching. These techniques do not seem to have such a big impact on the search to compensate the overhead that is introduced for the simplification coming from the component based structure of *riss*. To simplify the formula during a restart, the inprocessor has to be initialized and after the simplification the unit propagation and the removal heuristic have to add the new formula to their structures again. Ignoring the benefit of a smaller formula seems to be the better solution for the implementation of the solver because the overhead of the simplification can be avoided. The effect that can be achieved by simplifying the formula could also be increased by tuning the parameters of the simplifications. As already discussed in Section 3.6.9, this tuning is difficult and yields enough work to be done in another project.

5. Conclusions

This work shows that recent modifications on SAT solvers do have a significant impact on their performance. Combining the improvements leads to 13% more solvable instances. However, for single components the influence varies. The VSIDS heuristic is a quite powerful heuristic that seems to be well tuned for the benchmark. Modifying leads only to slightly changes on the performance of the SAT solver. Choosing the polarity of literals has a greater influence. Using phase saving enables the solver to solve 11 more instances. Adding different schemes to speed up the unit propagation has also side effects. Blocking literals in the watch list also change the search. Still, adding these components to the search does not result in a significant performance boost. The conflict analysis is the main difference between a DPLL and a CDCL solver. The analysis can be improved significantly when it is extended by conflict minimization. When conflict minimization is used, all other suggested improvements for the conflict analysis do not improve the performance significantly. Since learned clauses are added to the formula, applying unit propagation slows down and thus some of the learned clauses are removed again. The frequency of removals and the number of removed clauses is very important. Removals that are based on an activity of clauses perform much better than length-based schemes. As shown in Section 4.2, choosing an appropriate removal boosts the performance of a SAT solver. Removing clauses also helps the SAT solver to limit the amount of memory that is needed. Restarting the search influences the performance of a SAT solver, as shown in [Hua07]. Scheduling restarts frequently results in a more efficient search than scheduling restarts by using large intervals. Furthermore, simplifying the formula before the search is started can improve the overall performance of the SAT solver because there are fewer variables to decide and fewer clauses to be checked during unit propagation. Still, the applied preprocessor techniques consume time and can be tuned to reach better results to compensate their runtime again. This effect applies even more when these techniques are used to simplify the formula during a restart because the search has to be stopped and updated afterwards.

Well performing settings for the inprocessor that simplifies the formula during a restart have not been found. All in all, the component based implementation is not the most efficient way to implement a SAT solver. Most of the components need to communicate with each other. For example, the activity of a variable needs to be updated during conflict analysis or the decision heuristic has to be informed that a certain variable becomes undefined again during backtracking. All these procedures perform better if the method of the component is called instead of passing the information to the other component. Implementing a fixed configuration is also not a solution because it is very hard to determine the best configuration because of the interference of the components among each other. As shown in Section 4.1, combining the best configurations according to a single component does not result in the most powerful solver. Furthermore, the effect of a single component on the whole search is not known. According to the experiments in Section 4.2, it has been proved that restarts and removal have the biggest impact on the overall performance.

The results of this work is that recent improvements on SAT solvers have been effective. Since there is a great variety to implement a component of the solver, a configuration has to be chosen by the developer. This work shows that this task is not trivial but very difficult. Finding a well performing configuration for a single application might be feasible, but fixing a configuration for solving a mixed benchmark efficiently is always a compromise. It is also hard to completely understand the work of SAT solvers because most of their decisions are done heuristically although the main runtime is spent during unit propagation [MS10]. Based on this property the development of better heuristics could improve the performance of SAT solvers more and push the major part of runtime from unit propagation to reasoning. Unfortunately, developing new heuristics is also difficult because there is no metric known that can compare two solvers without running them. Thus, properties of the search can, at most, indicate whether a certain configuration is more powerful than another one.

6. Future Work

The main improvements of SAT solvers that participate in annual SAT competitions are only published in short descriptions. Analyzing these single improvements has brought up lots of open tasks that should be considered in the future. Naturally, the general aim of the development of SAT solvers is to create a solver with the highest performance possible. At a first glance, this solver should be an efficient sequential solver. To achieve this goal, the single components have to be improved. On the other hand, SAT solvers are used as general purpose solvers. Lots of applications can be converted into CNF and SAT solvers are used to solve these problems. In the snake plots in Section 3 and 4, it can be seen that for a certain threshold a configuration solves more instances than for another one although the overall performance of the second configuration is higher. Furthermore, the solved instances do not have to be the same. To improve SAT solvers, this effect could be used. The input formula has to be analyzed first and major properties like the distribution of literal occurrences, clause sizes or more complex information have to be extracted. Afterwards a classifier can be applied that selects the category of the current instance and chooses an appropriate configuration to solve it. Therefore, properties of formulas have to be identified and a classifier has to be trained to classify instances. Furthermore, the best performing configuration per category has to be found, for example by using the parameter optimization tool paramILS [HHLbS09]. Similarly, the preprocessor techniques have to be tuned better to suite the requirements of modern SAT solvers better. The step between being able to use all introduced techniques and using them within appropriate limits and parameters can be quite big. Even the order of the chosen techniques can change the overall performance of a SAT solver significantly. Therefore, the influence of preprocessing a formula on the performance of the search has to be analyzed. Furthermore, tuning the simplification during a restart has to be done to achieve a performance boost of the search and to overcome the overhead of the component based implementation.

The architecture of modern computing resources becomes parallel and the frequency of CPUs is stagnating. Thus, developing parallel SAT solvers has to be considered. To create an efficient parallel SAT solver, an efficient sequential one has to be developed first. Furthermore, bottlenecks have to be found and reduced. According to [HMS10], the memory architecture is the major bottleneck of a SAT solver and plays a major role in the analysis. Since recent parallel architectures share a single memory bus among several CPUs, the effect of this architecture has to be analyzed. Since hardware related events, like branch prediction, might interfere with the memory bus, they also have to be considered. After the sequential implementation is improved with respect to all the bottlenecks, an appropriate algorithm has to be found that suits the modern multi-core architecture. After the algorithm is implemented the effect of the bottlenecks has to be analyzed again to be able to draw a conclusion on the scalability of the chosen implementation. In a few years, the number of cores on a CPU will be significantly higher than today [Cor10]. Scalable implementations of SAT solvers are needed to be still able to solve industrial instances efficiently and without consuming unnecessary resources.

A. List of Figures

B. List of Algorithms

References

[AS09a] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. SAT 2009 Competitive Event Booklet, `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`, 2009.

[AS09b] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solver. In *Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, pages 399–404, jul 2009.

[BGH$^+$09] C. Baldow, F. Gräter, S. Hölldobler, N. Manthey, M. Seelemann, P. Steinke, C. Wernhard, K. Winkler, and E. Zenker. HydraSAT 2009.3 solver description. SAT 2009 Competitive Event Booklet, `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`, 2009.

[BHvMW09] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[Bie08a] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 28–33, Berlin, Heidelberg, 2008. Springer-Verlag.

[Bie08b] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[Bie09] A. Biere. PrecoSAT system description. `http://fmv.jku.at/precosat/preicosat-sc09.pdf`, 2009.

[BM00] R. Béjar and F. Manyà. Solving the round robin problem using propositional logic. In *Procs. 17th National Conf. on Artificial Intelligence and 12th Conf. on Innovative Applications of Artificial Intelligence*, 2000.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Procs. 3rd Annual ACM Symposium on Theory of Computing*, 1971.

[Cor10] Intel Corporation. Intel's Teraflops Research Chip. `http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf`, 2010.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT'05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.

[ES04] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. 6th SAT, LNCS 2919*, 2004.

[FGM$^+$07] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Procs. 10th SAT, LNCS 4501*, 2007.

[Gen02] Ian P. Gent. Arc consistency in sat. In *Proceedings of ECAI 2002*, pages 121–125. IOS Press, 2002.

[GSCK00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24:2000, 2000.

[Hö9] S. Hölldobler. *Logik und Logikprogrammierung, Band 1: Grundlagen*. Synchron, 2009.

[HHLbS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. Technical report, 2009.

[HJB10] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for cnf formulas. In Christian Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer Berlin / Heidelberg, 2010.

[HMS10] Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware sat solvers. In Christian Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer Berlin / Heidelberg, 2010.

[HS09] Hyojung Han and Fabio Somenzi. On-the-fly clause improvement. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 209–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[Hua07] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2318–2323, 2007.

[ILO06] Ines Lynce Ist, Inês Lynce, and Joël Ouaknine. Sudoku as a sat problem. In *Proceedings of the 9 th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale*. Springer, 2006.

[JBH10] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer Berlin / Heidelberg, 2010.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990. 10.1007/BF01531077.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *Procs. 10th European Conference on Artificial Intelligence*, 1992.

[lB10] Daniel le Berre. The International SAT Competition Webpage. `http://www.satcompetition.org/`, 2010.

[LMS03] Inês Lynce and João Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '03, pages 105–, Washington, DC, USA, 2003. IEEE Computer Society.

[LMS06] I. Lynce and J. Marques-Silva. SAT in bioinformatics: making the case with haplotype inference. In *Procs. 9th SAT, LNCS 4121*, 2006.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47:173–180, September 1993.

[Man10] Norbert Manthey. riss 2010 solver description. `http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_15.pdf`, 2010.

[MFM04] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In *SAT (SELECTED PAPERS)*, pages 360–375, 2004.

[MMZ$^+$01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference*, pages 530–535, 2001.

[MS10] Norbert Manthey and Ari Saptawijaya. Towards improving the resource usage of SAT-solvers. In *Pragmatics of SAT Workshop*, 2010.

[Nad04] Alexander Nadel. Efficient algorithms for clause learning SAT solvers. Master's thesis, Hebrew University, 2004.

[Nik10] Niklas Sörensson. Minisat 2.2 and minisat++ 1.1. `http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf`, 2010.

[NR10] Alexander Nadel and Vadim Ryvchin. Assignment stack shrinking. In *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 375–381. Springer, 2010.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing(SAT)*, pages 294–299, 2007.

[PD09] K. Pipatsrisawat and A. Darwiche. RSat solver description for SAT competition 2009. SAT 2009 Competitive Event Booklet, `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`, 2009.

[Rya04] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers, 2004.

[SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.

[SE09] N. Sörensson and N. Eén. MiniSAT 2.1 and MiniSAT++ 1.0 - SAT race 2008 editions. SAT 2009 Competitive Event Booklet, `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`, 2009.

[Soo10] Mate Soos. Enhanced gaussian elimination in DPLL-based SAT solvers. In *Pragmatics of SAT*, Edinburgh, Scotland, UK, July 2010.

[SP05] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. Niver: Non-increasing variable elimination resolution for preprocessing sat instances. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 276–291. Springer Berlin / Heidelberg, 2005.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[TTKB09] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2):254–272, 2009.

[ZMM01] Lintao Zhang, Conor F. Madigan, and Matthew H. Moskewicz. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Erklärung

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

Dresden, den 15. Dezember 2010

Norbert Manthey