TU Kaiserslautern

Development and Implementation of a Modular SAT-Solver Framework

Bachelor's Thesis in Computer Science

Author
Albert Schimpf
——

Supervisors
Prof. Dr. Roland Meyer
M.Sc. Sebastian Muskalla

August 6, 2016

Contents

I. Introduction **5**

1. Abstract **5**

2. Introduction **5**

3. Related Work **6**

II. History and Current State of SAT Solvers **7**

4. Preliminaries **7**

5. History **7**

6. Depth-First Backtrack-Based SAT Solvers **8**

6.1. The Recursive Davis-Putnam-Logemann-Loveland Algorithm 8
6.2. Non-Chronological Backtracking 11
6.2.1. Iterative DPLL 11
6.2.2. Implication Graphs 13
6.2.3. Resolution Calculus 20

7. Efficient Techniques **22**

7.1. Efficient Data Structures 22
7.1.1. Counter-Based Formula Representation 22
7.1.2. Two-Watched-Literals 24
7.1.3. Other Approaches 26
7.2. Variable Selection Heuristic 27
7.2.1. Accurate Variable Selection 27
7.2.2. Efficient Variable Selection 28
7.3. Other Techniques 28
7.3.1. Restarts 28
7.3.2. Clause Forget 29
7.3.3. Preprocessing 30

8. Overview **31**

III. Modular SAT Framework and Analysis 32

9. State Transition Systems for SAT 32
9.1. DPLL, DPLL State, DPLL Rules 32
9.1.1. Basic DPLL 33
9.1.2. Complete DPLL 35
9.2. DPLL Overview 39

10. Framework Analysis 40
10.1. Example Module Implementations 42
10.1.1. Counter-Based Formula 42
10.1.2. MiniSAT Decision Heuristic 45
10.1.3. Other Modules 47

11. Results 47
11.1. Seminar Results 47
11.2. Seminar Benchmarks 48

12. Conclusion and Future Work 54

IV. Appendix 55

A. Counter-Based SetOfClauses Implementation 55

B. Forget Heuristic Implementation 56

C. Restart Heuristic Implementation 57

D. Selected Benchmark Instances 58

Listings 59

Bibliography 59

Statutory Declaration

I hereby declare that I have authored this thesis independently and that I have only used sources listed in the bibliography, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. The thesis in this form or in any other form has not been submitted to an examination body and has not been published.

Date	Albert Schimpf

Part I. Introduction

1. Abstract

Propositional satisfiability is one part of the more general satisfiability testing field. We introduce recent research results concerning the efficiency of solvers and a theoretical approach to propositional satisfiability testing. We combine these two efforts to develop a framework for solving propositional formulas. The framework we developed has two major goals. The first one is to implement an efficient modular solver based on the given theoretical model. The second goal is to use this framework as a teaching tool.

2. Introduction

Propositional Satisfiability (referred to as SAT) is the problem of determining whether a propositional formula can be satisfied or not. Today, satisfiability testing has gained a firm ground and stands among other acclaimed research fields. Propositional satisfiability, which we will discuss, is one part of this field.
Although it is known that SAT is NP-complete [Coo71], efficient solvers have established themselves in specialized domains. There are currently three popular branches which are used for classification of SAT instances. Depending on whether a problem instance is derived from either the random, hard-combinatorial, or the application domain, different solvers may behave differently on each of the categories. We will concentrate on the application domain and on industrial SAT solvers in general. Concerning the industry, SAT solvers have established themselves in many areas. It is not uncommon to use solvers to solve problems in processor verification, planning, and computer encryption among other fields.
Recent research has introduced two abstract descriptions of the solving process for modern SAT solvers with the goal to extend it to Satisfiability Modulo Theories [KG07][NOT06]. We will focus on the model provided by Sava Krstić and Amit Goel [KG07].
In the first half, we will talk about the history of SAT solving. Our goal is to introduce state-of-the-art techniques used in modern SAT solvers. In the second half, we define the abstract transition system and use it to develop and implement a modular Java SAT solver framework. That solver was then used as a teaching tool in a seminar. Afterwards, we ran several benchmarks which we will use to analyze the framework. We will discuss the effectiveness and its use as a teaching tool as a conclusion.

3. Related Work

The main goal is to implement a state-of-the-art modular SAT solver framework. As such, we use the extensive knowledge provided by researchers in the field. We start with the basic recursive algorithm first introduced by Davis and Putnam [DP60] and the extension by Logemann and Loveland, namely DPLL [DLL62]. We introduce the non-chronological backtracking and conflict-driven learning proposition for SAT solvers by [SS96] and [BS97]. We will also mention popular implementations of this proposition by modern solvers, namely CHAFF [Mos+01] and SATO [ZS96]. Out of the 2 modular theoretical frameworks provided, we will use the one defined by Krstić and Goel [KG07].

All techniques, which are not mentioned here, will be cited when they are further described in their appropriate chapter.

Part II. History and Current State of SAT Solvers

4. Preliminaries

In this section, we introduce necessary notations used throughout the work. Definitions for *formulas, assignments and satisfaction* are taken from [NOT06].
Let P be a fixed finite set of propositional symbols. If $p \in P$, then p is an *atom* and p and $\neg p$ are *literals*. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals $l_1 \vee ... \vee l_n$. A CNF *formula* is a conjunction of one or more clauses $C_1 \wedge ... \wedge C_n$, also written as $C_1, ..., C_n$ if there is no ambiguity.
A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p. A literal l is *true* in M if $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* in M otherwise. A literal is *defined* in M if it is either true or false in M. The assignment M is *total* over P if no literal of P is undefined in M.
A clause C is *true* in M if at least one of its literals is in M. It is *false* or *conflicting* in M if all its literals are false in M, and it is *undefined* in M otherwise. A clause is *unit*, if all except one literal l are false in M and that literal l is undefined in M.
A formula F is *true* in M, or *satisfied* by M, denoted $M \models F$, if all its clauses are true in M. In this case M is a *model* of F. A formula F is *satisfiable* if there exists a model of F. A formula F is *satisfiable with (an assignment) M*, if there exists a model M' of F, such that $M \subseteq M'$. If F has no models then it is *unsatisfiable*.

5. History

Propositional satisfiability lies at the heart of complexity theory as the first problem to be classified as NP-complete by Cook in 1971 [Coo71]. Before its classification in terms of complexity, one of the first algorithms to solve the problem is shown by Davis and Putnam in 1960. The second phase of the algorithm described in [DP60] was needed to solve satisfiability of propositional formulas based on resolution. The runtime behavior was not good enough for real-world applications because of the memory blow-up of the resolution based approach.
The algorithm had many problems, one of which was memory consumption. This was solved by Logemann and Loveland by relying on a backtrack-based depth-first tree search instead of a resolution-based approach [DLL62]. Since then, attempts to

implement a solver for sufficiently complex problem instances stagnated and propositional satisfiability became largely an academic exercise.
About 20 years ago, Silva, Sakallah, Bayardo, and Schrag made a breakthrough by porting the concept of non-chronological backtracking (first proposed in the Constraint Satisfaction Problem domain [Pro93]) and conflict-driven learning to SAT solvers. These techniques led not only to more efficient solvers but also sparked interest in propositional testing again in general. Consequently, efficient solvers are now used in the industry domain. Examples of such solvers are CHAFF, SATO, and Siege [Rya02].

We begin to describe the features of modern SAT solvers in the following chapters. Our approach is as follows. First we describe how to engineer SAT solvers, then we describe the theory around it. We will introduce techniques chronologically and from high-level improvements to low-level ones. As such, we start with the most basic recursive DPLL algorithm. We expand it with the high-level concept of non-chronological backtracking, low-level efficient techniques, heuristics, and other techniques until we get to the point where we have a modern SAT solver comparable to CHAFF. This will be the foundation for the theory framework, which we will use to develop our own solver.

6. Depth-First Backtrack-Based SAT Solvers

Boolean satisfiability is classified as a NP-complete problem. It is believed that not all problem instances can be solved efficiently. However, experience has shown that there are certain areas where SAT solvers have become efficient enough to solve large problems. One of these areas is industry application.
We will focus on industrial SAT solvers. Therefore the goal of this chapter is to understand efficient SAT solvers which will be able to solve real-world industry instances. We begin with a basic recursive algorithm and improve it along the way until it is comparable to a modern solver on a high-level basis.

6.1. The Recursive Davis-Putnam-Logemann-Loveland Algorithm

In this section, we introduce the recursive DPLL algorithm with its deduction rules. As mentioned before, the first algorithm which solved propositional formulas was part of a two-phase proof-procedure for first-order logic. We skip this algorithm and start with the first backtrack-based depth-first tree search which we will use as a basis for improvement.

DPLL can be seen as an algorithm divided in two phases.

The first phase, the deduction phase, tries to find variable assignments that have to be fulfilled in order for the formula to be satisfied. DPLL uses two of such rules: Pure and Unit. These rules are proven to not modify the satisfiability of the formula, and as such are used exhaustively, i.e. until no rule can be applied anymore, to avoid unnecessary branching. We will look into each rule shortly.
The second phase, the split phase, ensures that the algorithm is *complete* and *terminates*. It selects an until now free variable and explores both possible assignments recursively, depth-first. One can see that — without the first phase — the idea is to explore all possible assignments of all variables contained in the formula. The exploration stops if the formula is found to be satisfying in the current branch. If no satisfying branch exists, the instance is declared unsatisfiable.
The formula presented to the solver is in *conjunctive normal form*. This normal form has established itself in automated theorem proving and most solvers accept this format. However, this is not a limitation because there exist algorithms to transform any propositional formula into a CNF formula with the same satisfiability in polynomial time. One such algorithm is the *Tseitin transformation* [Tse83].
Instead of blindly picking one of the many available versions of DPLL and showing its pseudo code, we should think about what properties we desire. This is especially important for when we are about to implement the algorithm, and also in anticipation of the transition system we are going to introduce in Part III. Therefore, the pseudo code should fulfill these four properties:

1. *Recursive, depth-first, backtrack-based* pseudo code
2. *Complete and terminating* algorithm
3. Separation of *syntax* (formula itself) and *semantics* (variable assignments)
4. Abstraction of the *deduction phase*

The first two points should be fulfilled by every recursive DPLL code. More interestingly, the separation of syntax and semantics will help us to correctly separate the code base and helps us to understand the code better. Lastly, abstraction of the deduction phase is needed, because we will later describe new techniques which we want to easily incorporate into the solver. The algorithm shown in Listing 1 from [ZM02] fulfills all four properties. We will not proof correctness and termination, as this is proved for the transition system in Part III.
Property one follows from Line one, twelve and sixteen. Here, the algorithm calls itself recursively to explore both assignments of the chosen variable in Line ten, i.e. it alternates the phase of the chosen variable.
Separation of syntax and semantics is used throughout the whole process. On the one hand, the algorithm has to be called with two parameters representing the syntax and

the semantics (Line one). On the other hand, deducing new variable assignments and choosing a free variable requires both the formula and the current assignments to function (Line two, Line ten).

Listing 1: Recursive DPLL Algorithm

```
1  DPLL(formula, assignment){
2    necessary = deduction(formula, assignment);
3    newAsgnmnt = union(necessary, assignment);
4    if (isSatisfied(formula, newAsgnmnt)){
5      return SATISFIABLE;
6    }
7    if (isConflicting(formula, newAsgnmnt)){
8      return CONFLICT;
9    }
10   var = chooseFreeVariable(formula, newAsgnmnt);
11   asgn1 = union(newAsgnmnt, assign(var, 1));
12   if (DPLL(formula, asgn1)==SATISFIABLE){
13     return SATISFIABLE;
14   }else{
15     asgn2 = union(newAsgnmnt, assign(var, 0));
16     return DPLL(formula, asgn2);
17   }
18 }
```

Lastly, the deduction phase is abstracted in Line two. With the given information (formula, current assignments) every deduction rule can be implemented, if needed.
We will now look into the two deduction rules used in DPLL more closely, as we will model them as rules of a transition system in Section 9.1. The correctness of these deduction functions will be proven later as well[1].

6.1.1 Proposition (Pure Rule)
Let F be a formula, M a partial assignment, and p undefined in M. If atom p is contained in every undefined clause in F either as the literal p (or $\neg p$ respectively), or not at all, then:
F with current partial assignment M is *satisfiable* if and only if F with $M \cup p$ (or $M \cup \neg p$ respectively) is satisfiable.

The application of a pure rule removes all occurrences of a literal in a single phase. This halves the search space and potentially satisfies many clauses at once. By its nature, this rule does not cause clauses to become conflicting. This powerful effect comes at the cost of checking the condition of its application because it needs information about all clauses per literal.

6.1.2 Proposition (Unit Rule, Unit Literal, Antecedent)
Let F be a formula and M a partial assignment. If there exists a clause $C = (l_1, \dots, l_k, l)$ such that $l_1, \dots l_k$ are false in M and l is undefined in M, then:
F with current partial assignment M is *satisfiable* if and only if F with $M \cup l$ is satisfiable.
In this case, l is called the *unit literal* of the clause. The clause C is called the *antecedent* of l.

[1]The original proofs can be found in [DLL62].

Unit is applicable if there are unsatisfied clauses with exactly one unsatisfied literal. This is why unit is very cheap, as it only requires knowledge about one clause and not the whole formula.

6.1.3 Example (Satisfiable Formula, Deduction Rules)
We will now show how the rules can be used. We mark literals and clauses in the formula *red* that are false in *M* and *green* that are true in *M* for clarity. It is not specified how literals are chosen for rule application and branching. We select literals arbitrarily.

Rule	Formula	Assignment
-	$(x_1 \lor x_2) \land (x_3 \lor \neg x_2 \lor x_4 \lor \neg x_4) \land (\neg x_4 \lor \neg x_3) \land (x_2 \lor x_3)$	$\emptyset$
Pure	$(x_1 \lor x_2) \land (x_3 \lor \neg x_2 \lor x_4 \lor \neg x_4) \land (\neg x_4 \lor \neg x_3) \land (x_2 \lor x_3)$	$\{x_1\}$
Split	$(x_1 \lor x_2) \land (x_3 \lor \neg x_2 \lor x_4 \lor \neg x_4) \land (\neg x_4 \lor \neg x_3) \land (x_2 \lor x_3)$	$\{x_1, \neg x_3\}$
Unit	$(x_1 \lor x_2) \land (x_3 \lor \neg x_2 \lor x_4 \lor \neg x_4) \land (\neg x_4 \lor \neg x_3) \land (x_2 \lor x_3)$	$\{x_1, \neg x_3, x_2\}$
Split	$(x_1 \lor x_2) \land (x_3 \lor \neg x_2 \lor x_4 \lor \neg x_4) \land (\neg x_4 \lor \neg x_3) \land (x_2 \lor x_3)$	$\{x_1, \neg x_3, x_2, x_4\}$

There exists a model $M = \{x_1, \neg x_3, x_2, x_4\}$ for F, therefore F is satisfiable.

There are a few problems with this algorithm which stopped its use in industry. We will address each of these problems in the following chapters:

- Recursive function calls are expensive.
- Classical backtracking is not feasible for industrial instances, the search space is too large.
- Classical data representations slow down the solver for industrial instances.
- Classical deduction mechanism is too expensive .

6.2. Non-Chronological Backtracking

The main goal of this section is to understand non-chronological backtracking (NCB, also called backjumping). We show why this was such a breakthrough for SAT solving in general. We introduce different forms of application of NCB, namely through implication graphs and resolution and discuss why we will need to rewrite the recursive algorithm into an iterative one. Lastly, we show why backtracking more than one level in depth is generally desired and where using NCB will not speed up the solving process.

6.2.1. Iterative DPLL

We start by rewriting our recursive algorithm in an iterative manner. This addresses not only the problem of recursive function calls, but gives us the tools to backtrack more than one level, which was not easily doable in the recursive version. However, our main goal

in this section is defining the *trail* structure, which is needed for the transition system in Part III.
Each time the recursive algorithm calls itself, the state of the algorithm has to be saved. We avoid that by applying an *order* on the assignment and by adding checkpoints for every decision made. Therefore, each time the algorithm enters a conflicting state, it can backtrack all literals in the trail until a checkpoint is reached, effectively imitating the recursive algorithm. With this change, the algorithm becomes *in-place*. We will now define the trail structure formally.

6.2.1.1 Definition (Trail M[2])
A trail M is a finite sequence. An entry in M can either be a literal $l \in P$ or a checkpoint symbol $\diamondsuit$. Literals in M are *ordered*: $l < l'$ means l occurs before l' in M.
Every literal in M is associated with a decision level, starting with 0. Every checkpoint $\diamondsuit$ increases the levels of the following literals by one. A literal d is called *decision literal* if it follows directly after a checkpoint $\diamondsuit$. The decision literal after the last checkpoint is denoted by D.
The literals of level m without the checkpoint symbol are denoted by $M^{\langle m \rangle}$. The prefix of M including decision level m is written as $M^{[m]} = M^{\langle 0 \rangle}.\diamondsuit. \dots . \dots\diamondsuit.M^{\langle m \rangle}$, where "." denotes concatenation. We write *level* $l = i$ if l occurs in $M^{\langle i \rangle}$.

6.2.1.2 Remark (Trail)
From now on, when we talk of M, we mean the Trail structure we introduced in this section instead of the definition given in the preliminaries.

By using the trail, we can now replace recursive function calls. One way to describe the iterative algorithm is shown in Listing 2 provided by [ZM02].

Listing 2: Iterative DPLL

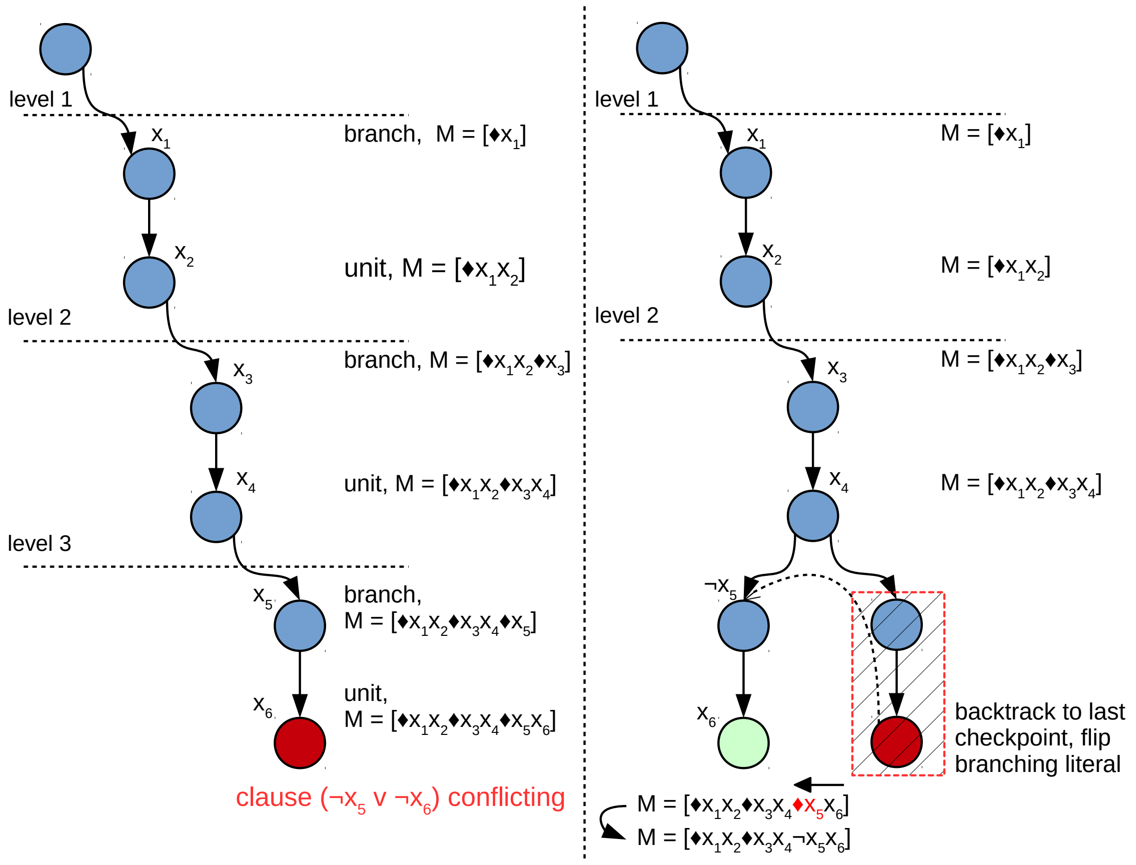
```
1  DPLLi(){
2    status = preprocess();
3    if(status != UNDEFINED) return status;
4    while(true) {
5      decide_next_branch();
6      while(true) {
7        status = deduce();
8        if(status == CONFLICT) {
9          blevel = analyze_conflict();
10         if(blevel == 0) return UNSATISFIABLE;
11         else backtrack(blevel);
12       }
13       else if(status == SATISFIABLE) return SATISFIABLE
14       else break;
15     }
16   }
17 }
```

The algorithm does not call itself recursively, instead it uses a while loop (Line four) until the formula is determined to be satisfied or conflicting (Line ten and thirteen). To imitate

[2]Definition similar to [KG07].

the recursive DPLL algorithm, we let the `preprocess()` function (Line two) imitate the `deduce()` function and return the current decision level decreased by one in the `analyze_conflict()` function (Line nine). We will demonstrate the solving process via a decision tree in the next example.

6.2.1.3 Example (Iterative DPLL Example, Decision Tree[3])
Let $F = (\neg x_1 \lor x_2) \land (\neg x_3 \lor x_4) \land (\neg x_5 \lor \neg x_6) \land (x_6 \lor \neg x_5 \lor \neg x_2)$.



level 1
x_1
branch, M = [♦x_1]
x_2
unit, M = [♦x_1x_2]
level 2
x_3
branch, M = [♦x_1x_2♦x_3]
x_4
unit, M = [♦x_1x_2♦x_3x_4]
level 3
x_5
branch,
M = [♦x_1x_2♦x_3x_4♦x_5]
x_6
unit,
M = [♦x_1x_2♦x_3x_4♦x_5x_6]
clause ($\neg x_5$ v $\neg x_6$) conflicting
level 1
x_1
M = [♦x_1]
x_2
M = [♦x_1x_2]
level 2
x_3
M = [♦x_1x_2♦x_3]
x_4
M = [♦x_1x_2♦x_3x_4]
$\neg x_5$
x_6
backtrack to last checkpoint, flip branching literal
M = [♦x_1x_2♦x_3x_4♦x_5x_6]
M = [♦x_1x_2♦$x_3x_4\neg x_5x_6$]

Since we deduced a conflict with the last application of the unit rule at the lowest level (Line seven), we backtrack to the last decision and flip the previous assignment of literal x_5 (Line eleven), thus decreasing the decision level by one. The loop continues and the formula is found to be satisfied with the current trail.

6.2.2. Implication Graphs

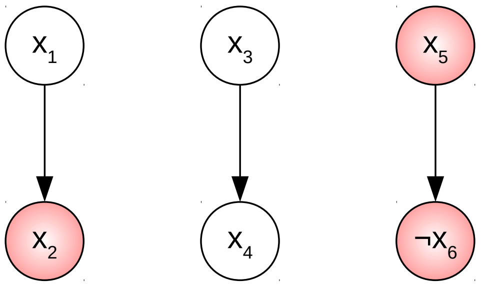
In this section, we introduce the concept of non-chronological backtracking via implication graphs first used in the solver GRASP [SS96]. The solving process itself can be seen as a decision tree (see previous Example 6.2.1.3). When we look at the classical backtracking mechanism, only the last decision is flipped to search for new solutions when a conflict occurs. Our goal is to determine whether we can undo more than one decision by analyzing the current conflict state of the formula[4].

[3]Instance taken from [KG07].
[4]There are many different deduction mechanisms used in SAT solvers. We show how to analyze the conflict via implication graphs, for which the unit deduction rule is needed.

Before giving the formal definition, we show an example.

6.2.2.1 Example

Let $F = (\neg x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_6 \vee \neg x_5 \vee \neg x_2)$ and $M = \{\diamondsuit 12 \diamondsuit 34 \diamondsuit 5\overline{6}\}$. We draw all literals contained in M as nodes of a graph. We mark the literals of the conflicting clause red and connect two nodes if one literal is implied by another one.



We can see that the decision variable x_3 in the tree has no effect on the conflict of the formula. Even if left out along with its unit implications, the conflict still occurs. We can derive from this graph that both clauses $(\neg x_1 \vee \neg x_5)$ and $(\neg x_2 \vee \neg x_5)$ would prevent this conflict if they were added to the original formula. In the case of the clause $(\neg x_1 \vee \neg x_5)$, assigning either x_1 or x_5 would lead to the unit implication of the negated other literal, thus preventing the conflict from happening. Those clauses are called *lemmas* or *backjump clauses*. One goal of NCB is to find these so called lemmas.

The notion of creating a graph out of variable assignments and unit implications is formalized in the definition of an *implication graph*.

6.2.2.2 Definition (Implication Graph[5])

Let M be a Trail. An implication graph is a labeled directed acyclic graph *G(V, E)*, where

- The nodes V represent assignments to variables. Each $v \in V$ is a labeled node corresponding to a literal $l \in M$ annotated with its decision level.
- Each edge represents an implication deriving from a clause that is unit under the current trail. Accordingly, edges are labeled with the respective antecedent clause of the unit literal the edge points to.
- The graph may contain a single conflict node, whose incoming edges are labeled with the corresponding conflict clause.

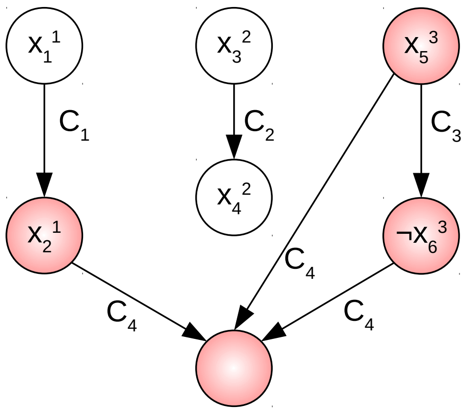
We will now show how to construct a full implication graph.

[5]Definition similar to [NOT06].

6.2.2.3 Remark (Construction of a Full Implication Graph)
Let F be a formula and M the current trail. Additionally, F should be in a conflicting state with at least one clause C_c conflicting in M.

1. Create a node for every literal $l \in M$. Annotate each literal with its current decision level.
2. Connect node v_i to node v iff C is the antecedent of the literal v and for every literal v_i which is contained in C. We do not connect the unit literal v to itself.
3. Create a conflict node. Connect edges from the conflict literals in C_c to the single conflict node.

6.2.2.4 Example (Full Implication Graph)
We modify our last Example 6.2.2.1 to suit the definition given earlier.
Let $F = (\neg x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_6 \vee \neg x_5 \vee \neg x_2)$ and $M = \{\diamondsuit 12 \diamondsuit 34 \diamondsuit 5\bar{6}\}$.

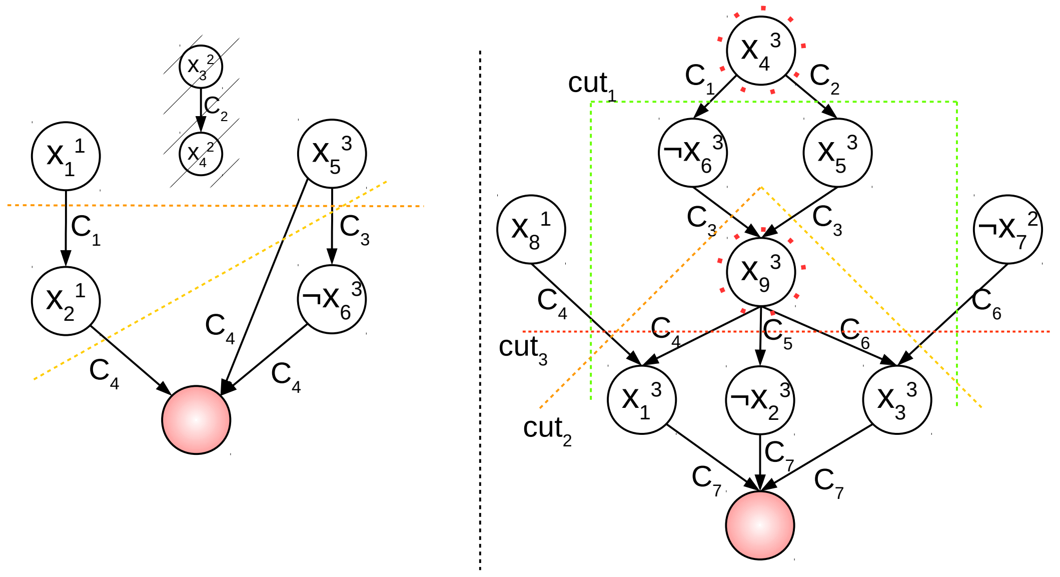


As mentioned earlier, *lemmas* prevent future conflicts if added to the formula. To methodically find such clauses, we need to separate the decision literals leading to the conflict node.
As described in [SM12], we can cut the graph in two parts[6]. One part contains at least all literals with no incoming arrows, the other contains at least the conflict node. One node can only be included in one part. The negated premises generated from the cut edges will generate a clause, which prevents the conflict from happening. The following example shows multiple graphs and multiple possible cut locations. The formula of the right graph is omitted, because it is not needed.

[6]It should be noted that the cuts are intended to be used on *partial* implication graphs, which we will introduce shortly. For our purposes, we can ignore all literals which do not have a path to the conflict node. While difficult to illustrate, the power of NCB comes from the fact that a partial implication graph is usually much smaller than a full one.

6.2.2.5 Example (Implication Graph Cuts)

We continue our last example on the left side and show a more complex implication graph on the right side. For the left graph, we can generate the clauses $(\neg x_1 \lor \neg x_5)$ and $(\neg x_2 \lor \neg x_5)$ which both prevent the conflict occurring in clause $(x_6 \lor \neg x_5 \lor \neg x_2)$.



For the right graph, at least three cuts are possible, named cut_1, cut_2, and cut_3 respectively. The following clauses are generated this way:

1. $(\neg x_8 \lor \neg x_4 \lor x_7)$,
2. $(\neg x_8 \lor x_6 \lor \neg x_5 \lor x_7)$,
3. $(\neg x_8 \lor \neg x_9 \lor x_7)$.

We can see that every clause effectively prevents the conflict from happening — every assignment, which is prevented by a clause, leads to the conflict through unit implications. The question remains which clause to pick.

We remind ourselves that our algorithm is depth-first. The higher deductions are made in the decision tree, the better. That is why we need to enforce unit propagation of the lemma as early as possible while skipping as many levels as possible. We achieve these goals by enforcing that the lemma contains only one literal assigned at the current decision level. This has the effect that the clause becomes unit if we backtrack at least one decision level[7]. From here on, we can backtrack as long as the backjump clause remains unit.

This means clause number two is not suited as a backjump clause. As for the other two clauses, we choose the one closer to the conflict, $(\neg x_8 \lor \neg x_9 \lor x_7)$. This is to anticipate the construction of partial implication graphs, which starts from the conflict node.

The only literal $\neg x_9$ in clause $(\neg x_8 \lor \neg x_9 \lor x_7)$ assigned at the current decision level is called a *unique implication point* in the implication graph and can be defined formally.

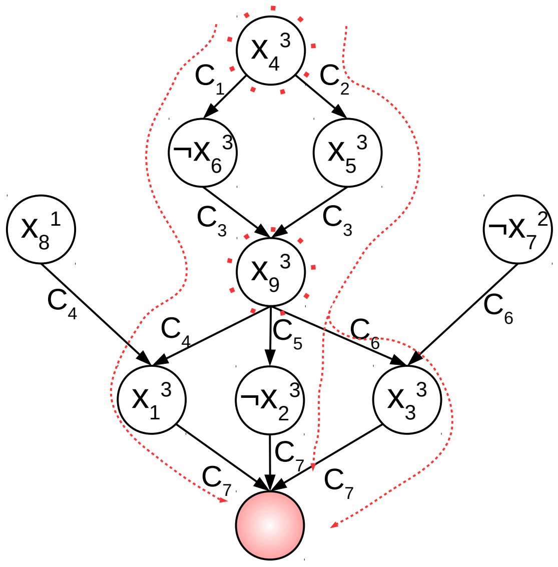
[7]We have to at least revert the last decision made. Also, if the backjump clause had more than one literal assigned at the current level, the clause would cease to be unit, even if we only backtracked one level.

The UIP is the literal which is flipped after the algorithm has finished backtracking. More precisely, the literal is assigned because it became unit after backtracking.

6.2.2.6 Definition (Unique Implication Point, UIP[8])
A unique implication point is any node (other than the conflict node) in the implication graph which is on all paths from the decision node to the conflict node of the current decision level.

It follows immediately that the decision node of the current decision level is already a UIP.

6.2.2.7 Example (UIP, Paths)
We continue the last Example 6.2.2.4 and mark the UIPs. We can see that there are two UIPs — literal x_4 and literal x_9. Only x_9 is on all paths from the decision literal to the conflict node apart from the decision literal itself.

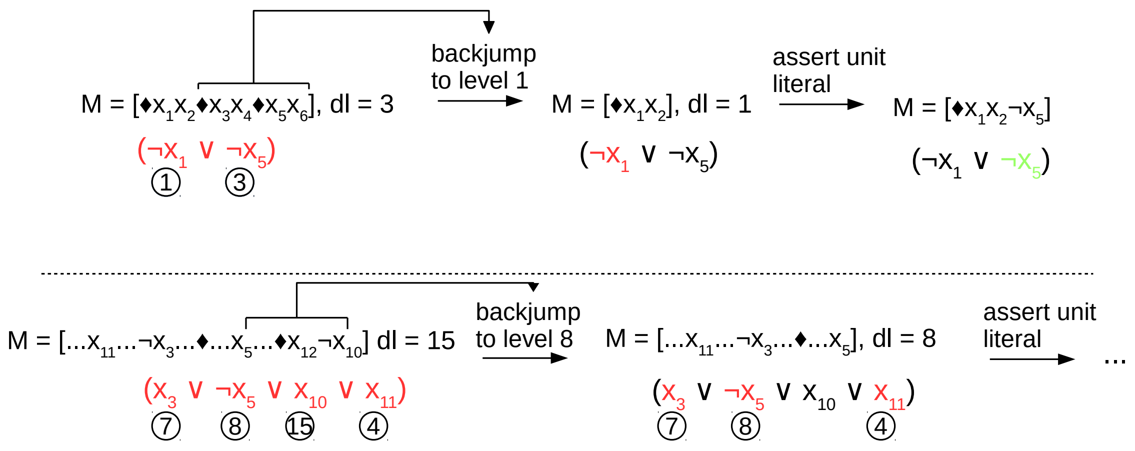


We now know how to generate backjump clauses using implication graphs and which literal to flip. The question that remains is how many levels we can backtrack when we found the lemma and know that it has one UIP.
Our goal is to skip to the farthest decision level which still makes the lemma unit after backtracking. Selecting the *second most recent* decision level of the lemma, that is the next biggest decision level after the current one, ensures maximum decision level skip.

[8]From [SM12]

6.2.2.8 Example (Second Most Recent Decision Level)

We will illustrate why selecting the second most recent decision level of the clause ensures maximum decision level skip. Since we know that only one literal of the backjump clause is assigned at the current decision level, we have to backtrack at least one level. We know that the clause ceases to be unit if the next biggest decision level of a literal *l* in the clause is also backtracked. Therefore, we know that we can backtrack to the next biggest decision level in the clause, called the *second most recent decision level* in the SAT literature.


$M = [\blacklozenge x_1 x_2 \blacklozenge x_3 x_4 \blacklozenge x_5 x_6]$, dl = 3
$(\neg x_1 \lor \neg x_5)$ ① ③
backjump to level 1
$M = [\blacklozenge x_1 x_2]$, dl = 1
$(\neg x_1 \lor \neg x_5)$
assert unit literal
$M = [\blacklozenge x_1 x_2 \neg x_5]$
$(\neg x_1 \lor \neg x_5)$
$M = [\dots x_{11} \dots \neg x_3 \dots \blacklozenge \dots x_5 \dots \blacklozenge x_{12} \neg x_{10}]$ dl = 15
$(x_3 \lor \neg x_5 \lor x_{10} \lor x_{11})$ ⑦ ⑧ ⑮ ④
backjump to level 8
$M = [\dots x_{11} \dots \neg x_3 \dots \blacklozenge \dots x_5]$, dl = 8
$(x_3 \lor \neg x_5 \lor x_{10} \lor x_{11})$ ⑦ ⑧ ④
assert unit literal
...

We know how to produce lemmas, we know which literal to flip, and we know how many levels we can backtrack at once. The remaining problem is that constructing full implication graphs is not efficient. One method which can improve the efficiency is constructing *partial* implication graphs instead.

The main idea of constructing partial implication graphs is to build the graph starting with the conflict literals and going from the end of the trail backwards. The construction is continued until the stopping criterion is satisfied. The construction usually stops at the *first UIP*, as implemented in CHAFF and in our own solver. As mentioned earlier, there is always a UIP and therefore the construction always stops.

Although there exist different schemes as when to stop the construction and which clauses to learn, experiments in [Zha+01] seem to indicate that stopping at the first UIP is the most efficient on all instances. Other solvers resolve further and learn intermediate clauses as well.

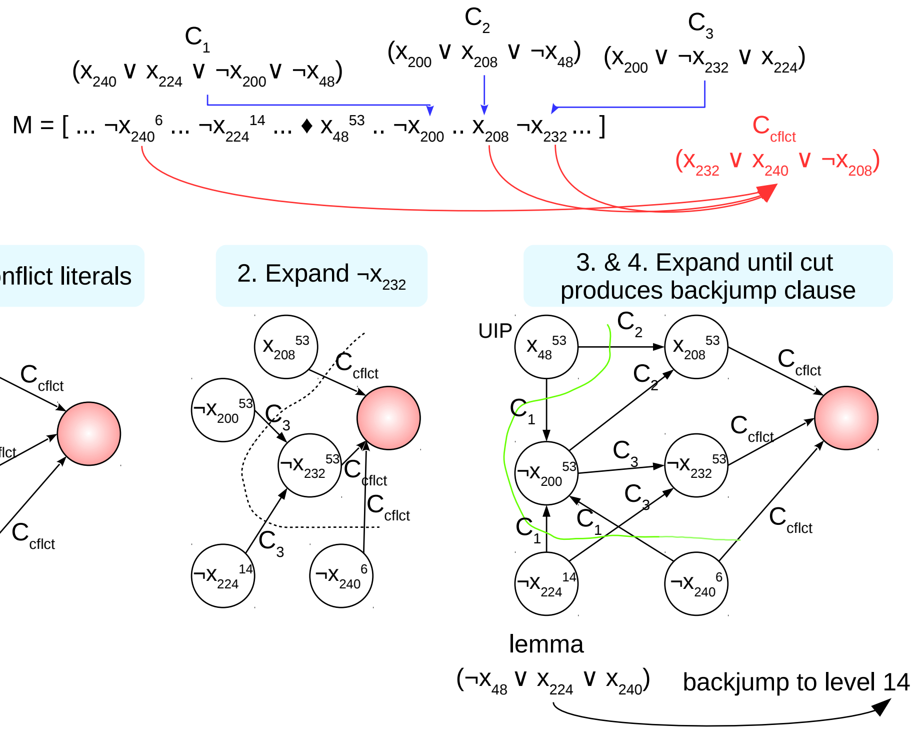
It should be noted, that we need the antecedent for every unit literal in the trail that we encounter. Most implementations save the reference to the antecedent clause when the unit rule is applied.

6.2.2.9 Remark (Partial Implication Graph Construction)
To construct a partial implication graph do the following steps.

- Create a conflict node.
- Create nodes of the literals in the conflict clause. Connect these nodes to the conflict node. Add these literals to the list of literals to be expanded.
- Expand this list of nodes with the saved antecedent clauses until a UIP is found. In the worst case, expand until the decision literal of the current decision level is found[9].
- Separate the graph with a cut to produce a backjump clause.

We will demonstrate constructing a partial implication graph in the following example.

6.2.2.10 Example (Partial Implication Graph Example)
We take a snapshot from the instance `aes_32_3_keyfind_1.cnf` from the SAT competition 2014 and construct a partial implication graph for a conflict. We can observe that we only need a glimpse into the actual state of the algorithm to find a suitable backjump clause. Explicitly, we only need the conflict clause and the antecedents C_1, C_2, and C_3 of the trail literals until the construction halts. We also observe that we jump from level 53 to level 14. Indeed, sometimes non-chronological backtracking can jump right to the beginning because a decision early on led to an unsatisfiable sub-tree.



C_1 $(x_{240} \lor x_{224} \lor \neg x_{200} \lor \neg x_{48})$
C_2 $(x_{200} \lor x_{208} \lor \neg x_{48})$
C_3 $(x_{200} \lor \neg x_{232} \lor x_{224})$
$M = [\dots \neg x_{240}^{6} \dots \neg x_{224}^{14} \dots \blacklozenge\, x_{48}^{53} \,..\, \neg x_{200} \,..\, x_{208} \; \neg x_{232} \dots]$
C_{cflct} $(x_{232} \lor x_{240} \lor \neg x_{208})$
1. Draw conflict literals
2. Expand $\neg x_{232}$
3. & 4. Expand until cut produces backjump clause
UIP
C_{cflct}
C_1
C_2
C_3
lemma
$(\neg x_{48} \lor x_{224} \lor x_{240})$
backjump to level 14

[9] This is true if the literals are expanded in the reverse order they appear in the trail, i.e. backwards. There are other solvers which apply other mechanisms to ensure that this process stops while ignoring the order in which the literals are expanded.

6.2.3. Resolution Calculus

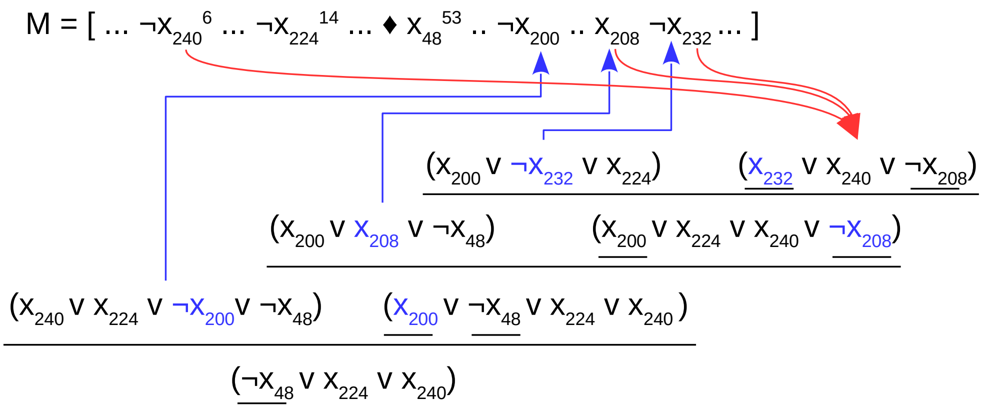
Instead of computing partial implication graphs, the same process can be seen as a derivation in the resolution calculus. This is more efficient than constructing graphs, as we do not have to model them in the framework.
The main goal remains the same. We still work from the trail end backwards and resolve antecedents with the current conflict set. The resolution process stops when there is only one literal left in the current decision level, precisely the first UIP. The following pseudo code from [ZM02] describes a more general process of the conflict analysis.

Listing 3: Resolution

```
1  analyze_conflict(){
2      cl = find_conflict_clause();
3      while(!stop_criterion_met(cl)){
4          lit = choose_literal(cl);
5          var = variable_of_literal(lit);
6          ante = antecedent(var);
7          cl = resolve(cl,ante,var);
8      }
9      add_clause_to_database(cl);
10     back_dl = second_most_recent_dl(cl);
11     return back_dl;
12 }
```

To ensure stopping at the first UIP, our stopping criterion (Line three) is that only one literal of current decision level remains in the clause. Additionally, we let `choose_literal` (Line four) pick the literal in the clause which is nearest to the trail end.

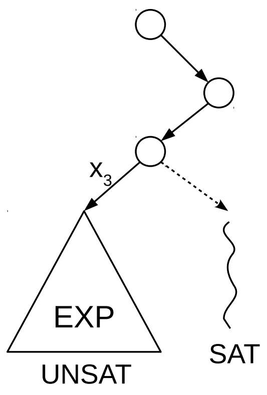
6.2.3.1 Example (Backwards Resolution)
We will revisit the partial implication graph construction in Example 6.2.2.10 and instead do it with resolution in the following example. A literal in the resolved clause is underlined if it is in the current decision level. Resolved literals are marked blue. We see that we do not need the graph structure in our program.


$M = [\dots \neg x_{240}^{6} \dots \neg x_{224}^{14} \dots \blacklozenge\ x_{48}^{53} .. \neg x_{200} .. x_{208}\ \neg x_{232} \dots]$
$(x_{200} \lor \neg x_{232} \lor x_{224})$
$(\underline{x_{232}} \lor x_{240} \lor \underline{\neg x_{208}})$
$(x_{200} \lor x_{208} \lor \neg x_{48})$
$(\underline{x_{200}} \lor x_{224} \lor x_{240} \lor \underline{\neg x_{208}})$
$(x_{240} \lor x_{224} \lor \neg x_{200} \lor \neg x_{48})$
$(\underline{x_{200}} \lor \underline{\neg x_{48}} \lor x_{224} \lor x_{240})$
$(\underline{\neg x_{48}} \lor x_{224} \lor x_{240})$

6.2.3.2 Remark (Termination and Correctness of Clause Learning, Conflict Analysis)
Formal arguments for general correctness and preservation of termination under non-chronological backtracking can be found in [MS95] [Sil99] [ZMG03]. The theoretical framework we will later use was provided with its own proofs which will be covered in Part III.

Non-chronological backtracking is often times more useful in unsatisfiable instances or instances with large unsatisfiable sub-trees. We can illustrate this in this theoretical example.

6.2.3.3 Example (Unsatisfiable Exponential Sub-Tree)
We assume decision x_3 is responsible for all following conflicts in this graph. Here, the algorithm descended into an exponentially large unsatisfiable sub-tree early on.



We can see that with chronological backtrack, it would take exponential time to recover from the bad decision x_3. With NCB instead, we would analyze the conflict at an additional computing cost, but the benefit of escaping the exponentially big unsatisfiable subtree would instantly pay off.
To further clarify the importance, we will look at a more practical example, as shown in [Har09]. Consider the clauses $C_1 \equiv (\neg x_1 \vee \neg x_n \vee x_{n+1})$ and $C_2 \equiv (\neg x_1 \vee \neg x_n \vee \neg x_{n+1})$ as part of an *unsatisfiable* formula F. Exploring the trail x_1 to x_n leads to a conflict forcing us to backtrack. Since F is unsatisfiable, the solvers backtracks further and each time the assignments x_1 to x_{n-1} are changed, the case where x_n is 1 again is unnecessarily explored. By analyzing the conflict we can determine that x_1 and x_n caused a conflict for either C_1 or C_2, so adding the conflict clause $(\neg x_1 \vee \neg x_n)$ eliminates this combination, and backtracking to the correct decision level prunes a large fraction of the search space.
Since most interesting instances are unsatisfiable [CA93] [CA96] [Sar99], we can see why NCB became such an important technique. Moreover, for some group of instances, NCB will slow down the solving process. Since analyzing the conflict introduces overhead, instances where the gained information is insignificant will be solved faster by chronological backtracking. This can be the case for random or hard-combinatorial instances. We will test this when we analyze our framework in later chapters.

7. Efficient Techniques

In the following chapters, we will cover efficient data structures, popular heuristics for underspecified parts of the solver, and miscellaneous improvements that proved themselves useful for SAT solvers. We limit our selection to techniques which already have been implemented in our own framework and describe them in more detail. Other techniques, which are interesting or are a improvement to already existing ones, will be mentioned either briefly or at the end of this work.

7.1. Efficient Data Structures

In this section we will cover two popular implementations for formula data structures in SAT solvers. There are many aspects to consider when choosing a data structure, and the trend is to pick an important feature of the solver and build the structure around it. Most solvers pick the unit rule (sometimes called Boolean constraint propagation, or BCP engine) as their main deduction mechanism.

As a reminder, we focus on industrial SAT solvers. Looking at representative instances of industrial propositional formulas (for example provided by the SAT competition[10]), they can be characterized by two main points: (1) They are large compared to formulas in other categories ($> 10^5$ variables, $> 10^6$ clauses) and (2) they are said to be *structured*[11]. The first goal is to introduce a simple data structure, one without information loss and for its simplicity certainly efficient for small instances. After that, we introduce the 2-watched-literals data structure introduced by CHAFF, which itself is based on the idea of head-tail-lists proposed by SATO. This structure is heavily tailored towards the unit deduction mechanism, because other than retrieving unit and conflict status, all operations are very expensive to compute. We end with a special section mentioning some unique approaches and ideas, some of them implemented in our solver.

7.1.1. Counter-Based Formula Representation

Until the introduction of specialized (often *lazy*, more on that later) data structures, counters were used to determine the state of each clause. Here, clauses keep counters to retrieve the state of the clause. Variables have two lists of the clauses that contain the variable positively and negatively. Every time a variable is set to true, false, or has its assignment undone, counters from clauses that contain this variable have to be

[10]http://www.satcompetition.org/

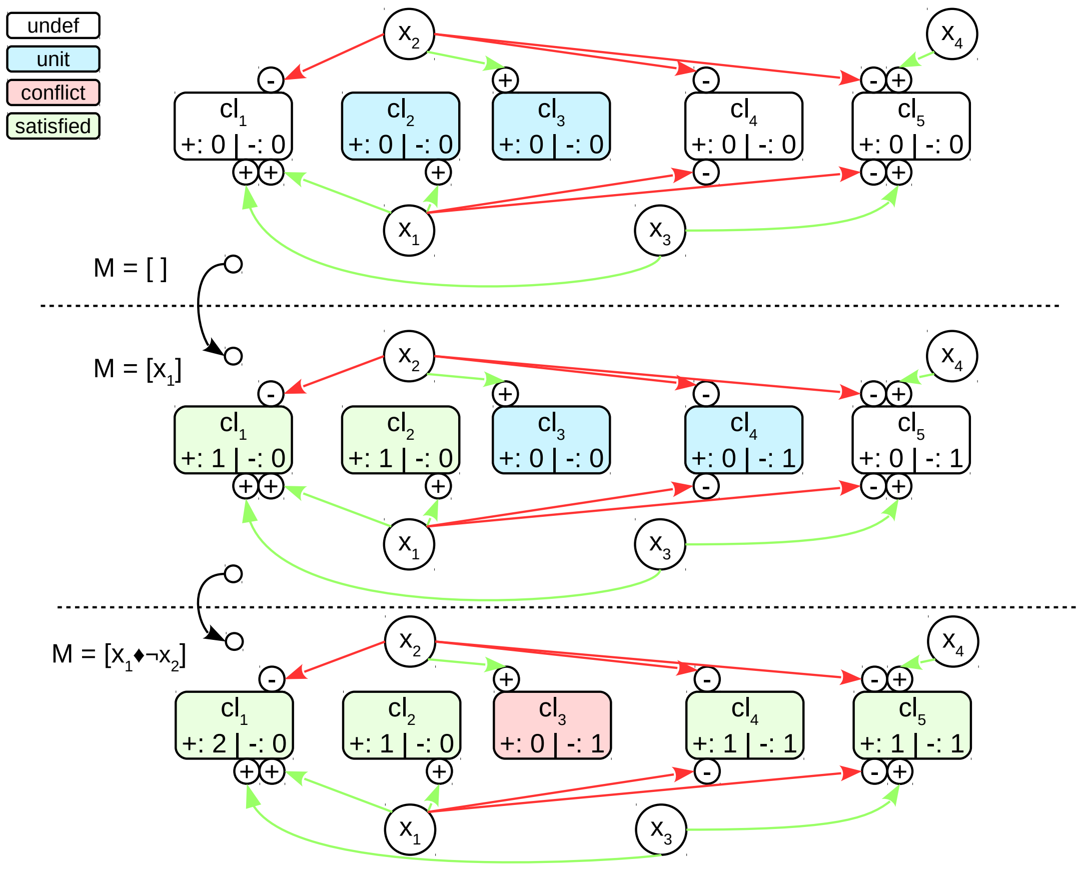
[11]Most of the time, 'structuredness' could not be defined in a mathematically precise manner. That is because given a formula, estimating the hardness is still challenging without solving it first. While estimating hardness of a formula is still under research, more recent research shows a promising approach via the graph property *community structure* [Lia+15].

updated. The four main operations[12] — isUnit(), isConflicting(), isSatisfied() and isUndefined() — can be defined as follows:

1. `isUnit()` iff ((unsatisfiedLit == sizeOfClause - 1) && satisfiedLit == 0)
2. `isSatisfied()` iff satisfiedLit > 0
3. `isConflicting()` iff (unsatisfiedLit == sizeOfClause)
4. `isUndefined()` iff !isSatisfied() && !isConflicting()

We will demonstrate how a typical counter-based clause responds to variable assignments in the next example.

7.1.1.1 Example (Counter-Based Representation)
We depict the clauses of a formula as counter-based clauses. Clause literals are hidden on purpose, as they are not needed to identify the clause state.
Let F = $(\neg x_2 \vee x_1 \vee x_3) \wedge (x_1) \wedge (x_3) \wedge (\neg x_2 \vee \neg x_1) \wedge (\neg x_2 \vee x_4 \vee \neg x_1 \vee x_3)$.



We will now briefly look at the advantages and the disadvantages of this data structure, which we discussed in [Sch14]. We can see that this structure is very light-weight. For every clause only three integers have to be saved, two counters and one to save the size of the clause. If we separate the actual literals from the clause, this data structure

[12]In most implementations only *isUnit*() and *isConflicting*() is used.

becomes very cache-friendly[13]. Since we check the state of a clause via counters, at the time the clause becomes unit the actual unit literal is not known. It has to be searched manually. Concerning theoretical performance, the counter-based approach seems bad runtime wise. If the formula has *m* clauses and *n* variables, with each clause having an average of *lits* literals, then assigning a value to a variable means $\frac{lits*m}{n}$ counters have to be updated. The same is true for undoing an assignment. Additionally, the unit literal has to be searched manually after a clause has been found to be unit.

7.1.2. Two-Watched-Literals

Data structures which do not have the full information about the clause state are called *lazy data structures*. In some industry fields, clauses can get large enough that non-lazy data structures become the bottleneck of the algorithm. Therefore, effort was invested on how to handle big clauses.
Based on the idea used in the solver SATO, the 2-watched-literals scheme is a specialized data structure for fast unit detection. To know whether the clause is unit or conflicting, we only need to know the state of two variables. The following example will clarify this observation.

7.1.2.1 Example (Lazy State)
Let $C_1 = (x_{12} \vee x_{14} \vee x_3 \vee x_1 \vee x_6 \vee x_5)$ and $C_2 = (x_1 \vee x_2 \vee x_{10} \vee x_9 \vee x_6)$. We hide all literals except two in each clause. We now look at the clause state, where $M = [x_2 \diamondsuit x_4 x_7 x_8 \diamondsuit x_{10}]$.

$$(? \vee ? \vee x_3 \vee ? \vee ? \vee x_5) \quad (x_1 \vee ? \vee ? \vee ? \vee x_6)$$

We can see that both clauses can not be unit or conflicting with the current trail. We also see that we lose the information about the second clause being satisfied by literal x_{10}.

As seen in the earlier example, 2-watched-literals relies on the fact that only the state of two literals in a single clause has to be known to determine unit and conflict status[14]. These two special literals are called *watches* and can move in either direction. The selection of the watches is arbitrary and happens at the time the clause is created. When these two initial literals are selected, references from the variable the literal corresponds to are also created.
We now look at how the state of a 2-watched-literals clause is stored, how to assign variables, and how to select new watches for a clause. The state of a 2-watched-literals clause is stored as two flags — one flag indicates unit state, the other is used to check if the clause is conflicting. These flags may change when a new watch is searched and are reset when a watched literal is backtracked. When the value 1 is assigned to a variable *p*, the watched negative literals list is iterated over. For each clause the literal is contained

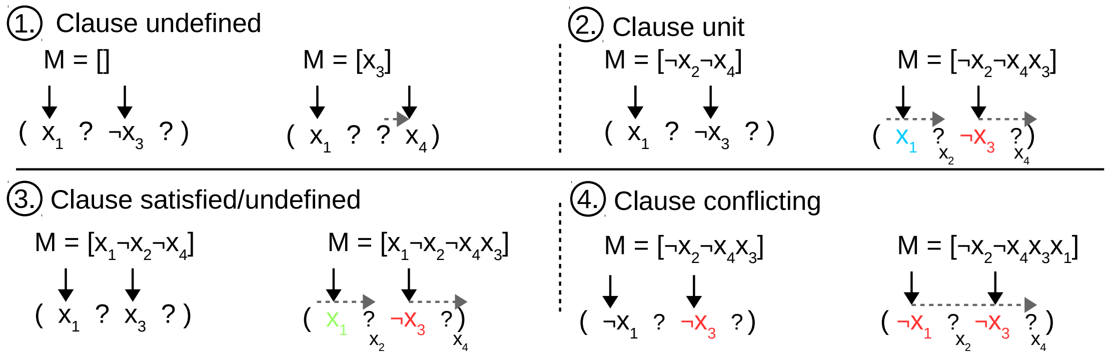
[13]The actual literals can then be fetched on demand.
[14]Only one literal is needed for conflict detection. The second watch is used to detect the last assignment before the clause becomes unit which is not possible with only one watch.

in we try to find a new literal *l* which is not false in the current trail *M*. There are four different cases that can occur, illustrated in Example 7.1.2.2. Assigning value 0 follows similarly.

1. Either an undefined literal or a literal which is true in *M* is found and is not the other watch: Set the previous watch to the new literal and connect the new variable to this clause. The reference from the old variable has to be removed.
2. All literals except the other watch are false in *M* and the other watch is undefined: The clause is a now unit clause, *l* being the unit literal.
3. All literals except the other watch are false in *M* and the other watch is true in *M*: The clause is now satisfied, nothing has to be done.
4. All literals except the other watch are false in *M* and the other watch is false in *M*: The clause is now conflicting.

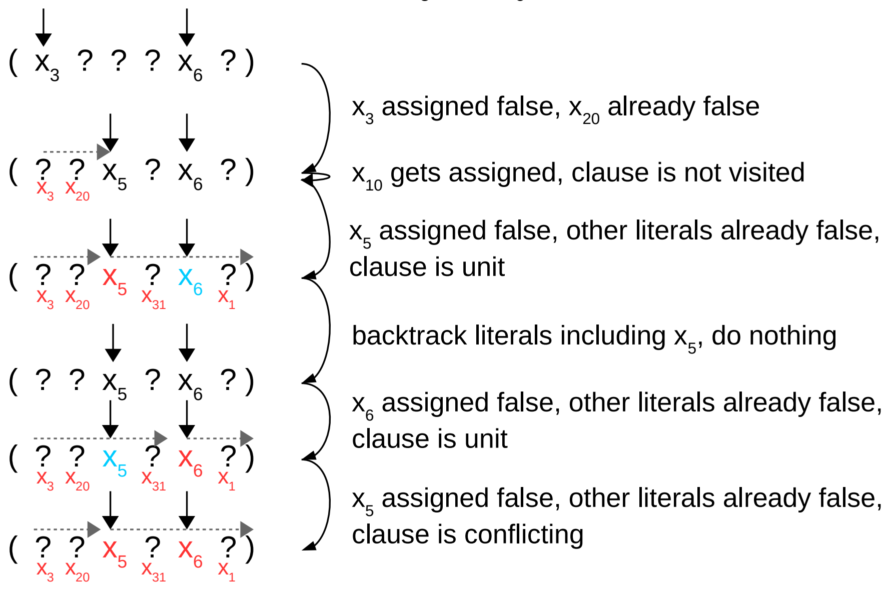
We see that the watches are not at all times unassigned. This is the case if the clause is currently unit or conflicting.

7.1.2.2 Example (2-Watched-Literals Variable Assignments)
We illustrate how new watches are searched for every case.



The following example demonstrates a typical life cycle of a 2-watched-literal clause.

7.1.2.3 Example (2-Watched-Literals Clause Life Cycle)
We will show how the clause $(x_3 \vee x_{20} \vee x_5 \vee x_{31} \vee x_6 \vee x_1)$ responds to variable assignments as a 2-watched-literals clause. We select x_3 and x_6 as initial watches.


(x3 ? ? ? x6 ?)
x3 assigned false, x20 already false
(? ? x5 ? x6 ?)
x10 gets assigned, clause is not visited
x5 assigned false, other literals already false, clause is unit
(? ? x5 ? x6 ?)
backtrack literals including x5, do nothing
(? ? x5 ? x6 ?)
x6 assigned false, other literals already false, clause is unit
(? ? x5 ? x6 ?)
x5 assigned false, other literals already false, clause is conflicting
(? ? x5 ? x6 ?)
x3 x20 x31 x1

We will again revisit the advantages and disadvantages discussed in [Sch14]. When assigning a value 1 to a variable, clauses that contain the literal positively will not be visited at all. This also holds true for assigning value 0 and clauses containing the literal negatively. Let m be the number of clauses and n the number of variables in the formula. Since each clause has effectively only two pointers, this means the status of only $\frac{m}{n}$ clauses needs to be updated. A huge improvement over SATOs head/tails list lies in the effort of undoing a variable assignment. In the 2-watched-literals scheme *no* work has to be done when unassigning a variable. It should be noted that the bigger the average clause size gets, the more effective this data structure becomes. We rarely need to visit a clause reference when assigning a variable that is not watched in many clauses. The disadvantage is the overhead when a new watch is searched. This is why this structure is only suited to represent big clauses, because a counter-based representation is more efficient the smaller a clause becomes.

7.1.3. Other Approaches

In this chapter we briefly talk about other approaches which modify already introduced structures or are of interest to our framework.

Satisfied Clause Hiding When assigning a variable in the counter-based data structure, a large number of clauses have to be traversed. One option is to exclude already satisfied clauses from that list to reduce the time used for assigning variables.

Efficient Small Clauses Special clauses that occur often in industrial instances are binary and ternary clauses. It is known that the status of such clauses can be computed in constant time. One efficient implementation can be found in Ryan's thesis [Rya02].

Mixed Data Structures The idea behind lazy data structures and 2-watched-literals scheme in particular is to reduce clause reference visits when assigning or backtracking a variable. In general, more computational time has be invested when such a clause gets actually visited. The effect gets more prominent the smaller the clause gets. One approach to handle this problem is to instantiate clause types depending on size. For unary, binary, and ternary clauses one can use constant data structures. For small clauses, the counter-based approach is a better choice than the lazy 2-watched-literals structure. And lastly, use lazy data structures for large clauses.

7.2. Variable Selection Heuristic

We remind ourselves that our goal is to design a complete solver which correctly determines satisfiability on all instances. Therefore, after the deduction phase it has to pick a free variable to explore the search space. How the variable is picked, however, is not specified.
At first glance this might seem not important, but there are two reasons — backed up by experiments — which indicate this step being among the most important ones. Firstly, a good decision heuristic (which could also be tailored to the problem) can decide if the instance is solvable in a reasonable time frame or not. Secondly, inefficient or too complex decision heuristics may easily become the primary bottleneck for a solver.
We will divide the heuristics we cover into accurate and efficient ones. Since branching selection is not the main focus, we will introduce only some of the many techniques that exist. All heuristics introduced can be or have already been implemented in the framework and will be analyzed in the appropriate chapter.

7.2.1. Accurate Variable Selection

The main idea behind accurate heuristics is using information about the whole formula, sometimes including the current assignment of variables (i.e. they are *state-dependent*), to derive an *accurate* but expensive to compute heuristic. It quickly became apparent that this type of branch selection is not usable for industry application, because computation would require large time consumption at every decision level. Naturally and over time, usage of accurate branching heuristics faded into the background. In the following we present some of the most used heuristics at the time.

BOHM's Heuristic Pick the variable that generates the most satisfying clauses while preferring small clauses over big ones. The goal is to satisfy many clauses with the next branch. This was the best state-dependent branching heuristic around 1992.

MOM's Heuristic Short for **M**aximum **O**ccurrences in clauses of **M**inimum **S**ize. Searches for *literals* that occur the most often in small clauses. The goal is to find literals that constraint the formula the most early on. There are many variants for this type of heuristic. They mostly differ in how to count literal occurrences (either count positive and negative literals separately or together) and additional priority functions. An overview for variants of this heuristic can be found in [Sar99].

7.2.2. Efficient Variable Selection

There are two reasons why efficient branching heuristics became popular in recent years. On the one hand, accurate heuristics did not capture relevant information for industry instances. On the other hand, solvers became more and more optimized and calculating the next variable to branch would become the bottleneck if it was too complex. We will look at two efficient heuristics.

VSIDS Variable State Independent Decaying Sum. A literal counting technique which keeps scores for each literal. Increases the score of literals whenever a clause is added. Periodically, all the scores are divided by a constant number. The idea is that literals which are active in conflicts and in recently learned clauses get bumped. Additionally, by decaying periodically we retrieve literals which are currently important in order to satisfy newly learned clauses.

VSIDS has proven its effectiveness over time. One of the most popular heuristic (including its variants) used until today.

MiniSAT A variant of VSIDS used in the MiniSAT solver. Instead of increasing the score when a new clause is added, the score is increased whenever a conflict occurs. Introduced low-level improvements, such as using an arraylist-based heap structure to retrieve the highest undefined variable. MiniSAT additionally randomizes variable selection two percent of the time.

7.3. Other Techniques

We will discuss other methods which have proved themselves useful for SAT solvers. We will cover restarts, clause forget, and preprocessing. These techniques will be implemented in our solver.

7.3.1. Restarts

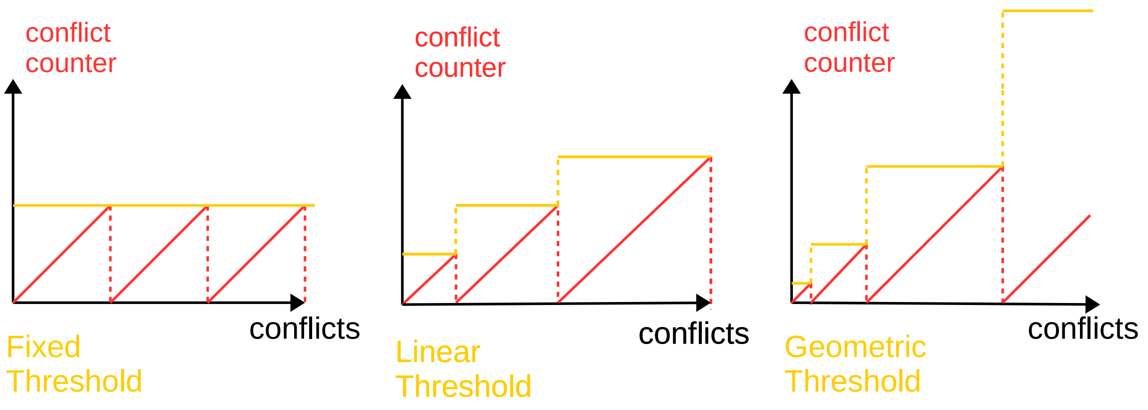
The main goal of search restart strategies is to increase robustness of a solver by ensuring that the solver does not get stuck in a sub-tree for too long. This also means that unsatisfiable sub-trees are not searched because learned clauses prohibit searching them again. One popular mechanic to implement restarts is counting how many conflicts occur and reseting after a certain threshold has been met. We will visualize the following three restart methods in Example 7.3.1.1.

Fixed Conflict Counting The solver restarts after a fixed amount of conflict has occurred. Popular constants include 550 (BerkMin), 700 (zChaff), and 16000 (Siege).

Linear Conflict Counting The solver increases the restart threshold by a fixed amount every time it restarts.

Geometric Conflict Counting The restart strategy is given two parameters. The first parameter is used for the first restart. The second parameter is used to multiply the current conflict threshold, thus increasing it geometrically after a restart is used. Used for example in MiniSAT with parameters (150, 1.5) .

7.3.1.1 Example (Graphs Restart Curves)



7.3.1.2 Remark (Aggressive Restarts)
We can see that the solver zChaff uses 700 as a fixed restart parameter. Low fixed parameters should only be used with clause learning to ensure that the solver does not get stuck restarting.

7.3.2. Clause Forget

Many learned clauses are accumulated during the runtime of a solver. These clauses are used for two things. Firstly, they may become an antecedent clause for another unit literal, ensuring completeness for non-chronological backtracking. Secondly, after a restart they prevent the search of an unsatisfiable sub-tree. Nevertheless, some clauses are redundant and less *important* than other ones.
It would be advantageous if there was some global measure which could identify most important clauses. Measures could include clause size or resolution steps during analysis, which may intuitively affect the importance of a clause. Contrary to belief

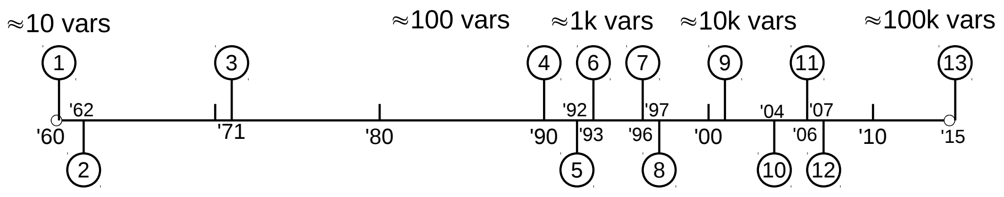
(especially the belief that big clauses are more important than small ones), [AS08] seems to suggest that there is no quality global measure of how to identify important clauses. This is why fine tuning forget heuristics is difficult.

7.3.3. Preprocessing

Some deduction mechanisms are very expensive, even with non-lazy data structures. The goal of preprocessing is to apply expensive operations only one time at the start until all deduction rules are exhausted, sometimes solving the whole instance in the process. Intuitively, it may appear that preprocessing is always preferable, as most of the time the instance shrinks compared to the original size. The results are mixed. Indeed, preprocessing can also slow down the solver, as shown in [LMS01]. As preprocessing is not actually part the main solving process, we refer to [EB05] for actual used preprocessing techniques.

8. Overview

In the last chapters, we discussed many old and new techniques of SAT solvers. Here we display what we introduced in a time line again for a quick overview. The time line is annotated with years, started from 1960 and ending in 2015. Marked above the line are the number of variables for industrial instances, which could be solved at the time.



1 Davis-Putnam algorithm
2 Davis-Putnam-Logemann-Loveland algorithm
3 SAT NP-completeness by Cook
4 MOM heuristic
5 BOHM heuristic
6 Non-chronological backtracking in CSP domain
7 NCB ported to SAT solver GRASP, implication graphs
8 NCB SAT solver rel_sat
9 2-watched-literals, efficient resolution, VSIDS. Solver Chaff
10 Efficient small clauses. Solver Siege
11 Abstract DPLL
12 SAT transition system
13 VSIDS theoretical research

For thirty years since the initial DPLL algorithm, solvers were not used for any practical purposes. This is visualized by the big gap between 1962 and 1966 in the time line. Since the introduction of non-chronological backtracking first used in the solver GRASP, propositional testing gained attraction in the research field. More solvers were developed and new techniques were discovered, which led to solvers which could be used in a practical environment. As the engineering side gets more and more advanced, interest in the theoretical side is also increasing. Recent research includes why some heuristics are efficient in certain instance domains and how to define 'hardness' of a formula without solving it first.

Part III. Modular SAT Framework and Analysis

In this part, we introduce the state transition system proposed by Sava Krstić and Amit Goel [KG07] which describes the SAT solving process at an abstract level. Based on this system, we develop our own solver and benchmark it on industry instances provided by SATLIB. We also use this framework for a seminar and discuss results concerning its use as a teaching tool.

9. State Transition Systems for SAT

The past of SAT solving is mostly dominated by experimental results and clever engineering. One of the many problems is that one can not easily estimate the 'hardness' of a formula without solving it first. Another problem was that the solving process itself was not described at an abstract level.
Abstract DPLL by [NOT06] and the non-deterministic transition system proposed by [KG07] try to address the second issue by providing a theoretical foundation for SAT solvers. This additional abstraction layer between theory and code allows reasoning about correctness and termination in a formal way. Moreover, it helps developers by providing a clear guideline to follow and separates high-level techniques from low-level ones by under-specifying the transition system. We will use the transition system from [KG07] as a basis. We will also rely on definitions provided by [KG07].
In the following sections, we will introduce the *DPLL State* and rules to operate on the state of the non-deterministic system. After that, we introduce *DPLL Strategies* which restricts the behavior of the system to essentially mimic actual solver algorithms.

9.1. DPLL, DPLL State, DPLL Rules

A *DPLL State* and rules, which operate on the state, from the *DPLL* system. The state captures all relevant information of the solving process such as formula (syntax), assignment trail (semantics), and conflict management.

9.1.1 Definition (DPLL State <F,M,C>)
Let P be a finite set of propositional literals. A DPLL *state* is a triple $\langle F, M, C \rangle$. F is a set of clauses over P, M is a *Trail*, and C is either a subset of P or the symbol *no_cflct*. The initial state is <F_{init}, $M = \emptyset$, C = *no_cflct*>, where F_{init} is an arbitrary set of clauses.

The following two sections introduce rules to operate on the DPLL state. One basic rule set which is able to imitate the recursive DPLL and an extended rule set which describes modern solvers at a high level.

9.1.1. Basic DPLL

We will need to model three basic functions for the transition system to function properly. Selecting variables to branch upon (decide), detecting a conflict, and backtracking after a conflict has been detected. Additionally, we need to model each deduction rule independently. For now, we will only model the unit rule. We will postpone correctness and termination proofs until all rules have been introduced. We start with the decide rule.

Decide. This rule simulates the split in the recursive algorithm (Line 10, Listing 1) and the decision (Line 5, Listing 2) in the iterative one. It selects a free literal which can be added to the trail with a checkpoint. This checkpoint can be used to explore the other assignment of the literal. The Example 9.1.1.2 shows how the rule is applied.

9.1.1.1 Definition (Decide Rule)
If there exists a free literal $l \in P$, the literal can be added to trail M after insertion of a checkpoint $\diamondsuit$.

$$\boxed{\frac{l \in P \text{ and } l, \neg l \notin M}{M := M + \diamondsuit + l}}$$

9.1.1.2 Example (Decide Example)
Consider $< F_{init} = \{\{\neg x_1, x_2\}, \{x_1, x_2\}\}, M = \emptyset, C = no\_cflct >$, $P = \{x_1, x_2, \neg x_1, \neg x_2\}$

$$\begin{tikzcd}
\{\{\neg x_1, x_2\}, \{x_1, x_2\}\} \arrow[d, bend right, "decide\ x_1 = 0"'] & M = \emptyset \arrow[d, bend left, "\frac{\neg x_1 \in P \text{ and } \neg x_1, x_1 \notin M}{M := M + \diamondsuit + \neg x_1}"] \\
\{{\color{green}\{\neg x_1, x_2\}}, \{{\color{red}x_1}, x_2\}\} \arrow[d, bend right, "decide\ x_2 = 1"'] & M = [\diamondsuit \neg x_1] \arrow[d, bend left, "\frac{x_2 \in P \text{ and } x_2, \neg x_2 \notin M}{M := M + \diamondsuit + x_2}"] \\
\{{\color{green}\{\neg x_1, x_2\}}, {\color{green}\{x_1, x_2\}}\} & M = [\diamondsuit \neg x_1 \diamondsuit x_2]
\end{tikzcd}$$

Unit. This models the unit deduction rule 6.1.2. The last free literal of an undefined clause can be added to the trail. The Example 9.1.1.4 shows how the unit rule is applied.

9.1.1.3 Definition (Unit Rule)
If there is a clause with only one literal l unassigned and all other literals are negated in M, l can be added to the trail.

$$\boxed{\frac{l \vee l_1 \vee ... \vee l_k \in F \text{ and } \neg l_1, ..., \neg l_k \in M \text{ and } l, \neg l \notin M}{M := M + l}}$$

9.1.1.4 Example (Unit Example)
Consider $< F = \{\{\neg x_1, x_2\}, \{x_1, x_2\}\}, M = \emptyset, C = no\_cflct >, P = \{x_1, x_2, \neg x_1, \neg x_2\}$

$$\begin{tikzcd}
\{\{\neg x_1, x_2\}, \{x_1, x_2\}\} \arrow[d, bend right, "decide\ x_1 = 0"'] \\
\{\{\neg x_1, x_2\}, \{x_1, x_2\}\} \arrow[d, bend right, "unit\ x_2 = 1"'] \\
\{\{\neg x_1, x_2\}, \{x_1, x_2\}\}
\end{tikzcd}
\qquad
\begin{tikzcd}
M = \emptyset \arrow[d, bend left, "\text{Unit guard not satisfied}"] \\
M = [\Diamond \neg x_1] \arrow[d, bend left, "\frac{x_2 \vee x_1 \in F \text{ and } \neg x_1 \in M \text{ and } x_2, \neg x_2 \notin M}{M := M + x_2}"] \\
M = [\Diamond \neg x_1 x_2]
\end{tikzcd}$$

Conflict. We detect conflicts with the conflict rule. This enables other rules which need a conflict state for their guard, like backtrack. Conflict rule application is shown in Example 9.1.1.6.

9.1.1.5 Definition (Conflict Rule)
If there is currently no conflict and a clause C_c is conflicting in M, then set C as the set of conflicting literals of the clause C_c.

$$\boxed{\frac{C = no\_cflct \text{ and } \neg l_1 \vee ... \vee \neg l_k \in F \text{ and } l_1, ..., l_k \in M}{C := \{l_1, ..., l_k\}}}$$

9.1.1.6 Example (Conflict Example)
Consider $< F = \{\{\neg x_1, x_2\}, \{x_1\}\}, M = \emptyset, C = no\_cflct >, P = \{x_1, x_2, \neg x_1, \neg x_2\}$

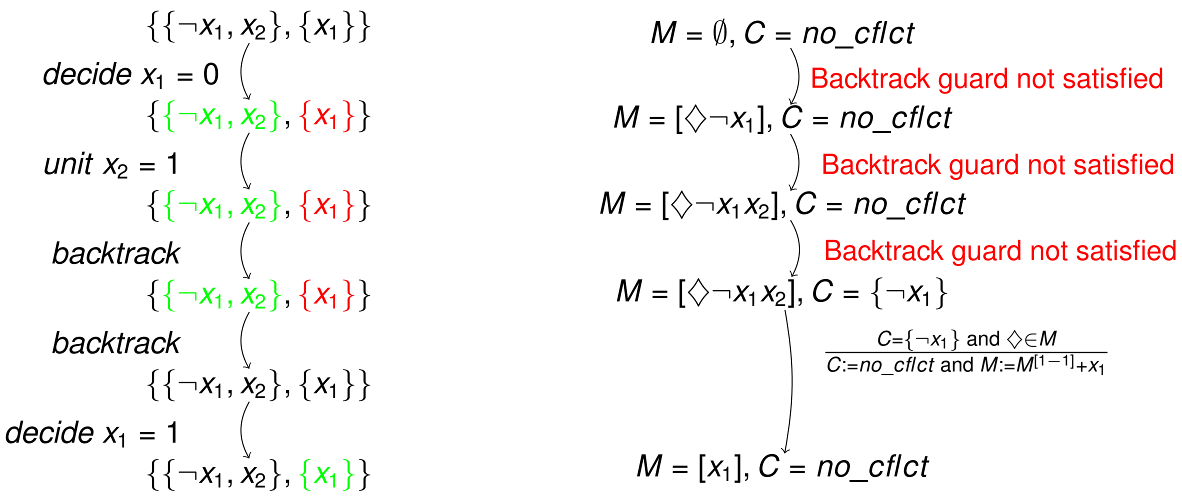
$$\begin{tikzcd}
\{\{\neg x_1, x_2\}, \{x_1\}\} \arrow[d, bend right, "decide\ x_1 = 0"'] \\
\{\{\neg x_1, x_2\}, \{x_1\}\} \\
\{\{\neg x_1, x_2\}, \{x_1\}\}
\end{tikzcd}
\qquad
\begin{tikzcd}
M = \emptyset, C = no\_cflct \arrow[d, bend left, "\text{Conflict guard not satisfied}"] \\
M = [\Diamond \neg x_1], C = no\_cflct \arrow[d, bend left, "\frac{C = no\_cflct \text{ and } x_1 \in F \text{ and } \neg x_1 \in M}{C := \{\neg x_1\}}"] \\
M = [\Diamond \neg x_1], C = [\neg x_1]
\end{tikzcd}$$

Backtrack. Whenever a conflict occurs, we need to backtrack to the last checkpoint. This rule ensures that the state transition system terminates on all instances together with the decide and conflict rule, as the whole search space can be explored with these three rules. An example of the backtrack rule can be seen in Example 9.1.1.8.

9.1.1.7 Definition (Backtrack Rule)
If there is currently a conflict and at least one checkpoint, backtracking is possible. Backtrack all literals until a checkpoint is reached, remove that checkpoint, and add the negation of the previous decision level literal to the trail. As a reminder, we denote the literal after the last checkpoint by D.

$$\boxed{\frac{C = \{l_1, ..., l_k\} \text{ and } \Diamond \in M}{C := no\_cflct \text{ and } M := M^{[level\ D-1]} + \neg D}}$$

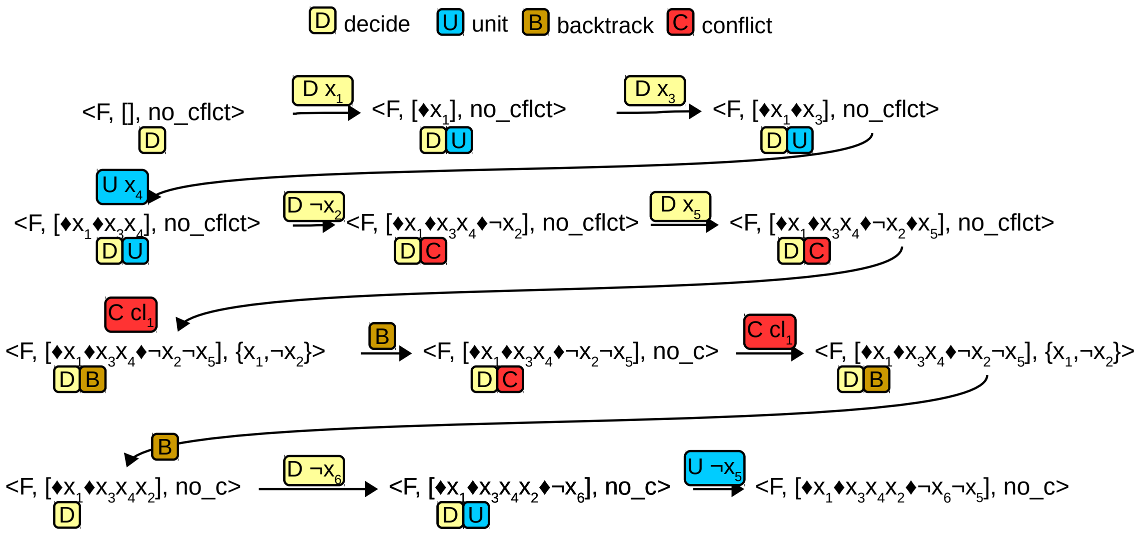
9.1.1.8 Example (Backtrack Example)
Consider $< F = \{\{\neg x_1, x_2\}, \{x_1, x_2\}\}, M = \emptyset, C = no\_cflct >$, $P = \{x_1, x_2, \neg x_1, \neg x_2\}$



$\{\{\neg x_1, x_2\}, \{x_1\}\}$
decide $x_1 = 0$
$\{\{\neg x_1, x_2\}, \{x_1\}\}$
unit $x_2 = 1$
$\{\{\neg x_1, x_2\}, \{x_1\}\}$
backtrack
$\{\{\neg x_1, x_2\}, \{x_1\}\}$
backtrack
$\{\{\neg x_1, x_2\}, \{x_1\}\}$
decide $x_1 = 1$
$\{\{\neg x_1, x_2\}, \{x_1\}\}$
$M = \emptyset, C = no\_cflct$
Backtrack guard not satisfied
$M = [\diamond\neg x_1], C = no\_cflct$
Backtrack guard not satisfied
$M = [\diamond\neg x_1 x_2], C = no\_cflct$
Backtrack guard not satisfied
$M = [\diamond\neg x_1 x_2], C = \{\neg x_1\}$
$\frac{C=\{\neg x_1\} \text{ and } \diamond\in M}{C{:=}no\_cflct \text{ and } M{:=}M^{[1-1]}{+}x_1}$
$M = [x_1], C = no\_cflct$

With these four basic rules we are now able to emulate the recursive DPLL behavior. The following example illustrates the non-deterministic transition system behavior on a satisfiable instance. Applicable rules are depicted directly under the state. Literal selection and rule application order is arbitrary.

9.1.1.9 Example (Satisfiable Instance)
Let $F = \{\{\neg x_1, x_2\}, \{\neg x_3, x_4\}, \{\neg x_5, x_6\}, \{\neg x_2, \neg x_5, \neg x_6\}\}$. We apply rules randomly until no rule guard is satisfied anymore. At the end, the formula is either satisfied or conflicting.



D decide
U unit
B backtrack
C conflict
<F, [], no_cflct>
D
D x₁
<F, [♦x₁], no_cflct>
D U
D x₃
<F, [♦x₁♦x₃], no_cflct>
D U
U x₄
<F, [♦x₁♦x₃x₄], no_cflct>
D U
D ¬x₂
<F, [♦x₁♦x₃x₄♦¬x₂], no_cflct>
D C
D x₅
<F, [♦x₁♦x₃x₄♦¬x₂♦x₅], no_cflct>
D C
C cl₁
<F, [♦x₁♦x₃x₄♦¬x₂¬x₅], {x₁,¬x₂}>
D B
B
<F, [♦x₁♦x₃x₄♦¬x₂¬x₅], no_c>
D C
C cl₁
<F, [♦x₁♦x₃x₄♦¬x₂¬x₅], {x₁,¬x₂}>
D B
B
<F, [♦x₁♦x₃x₄x₂], no_c>
D
D ¬x₆
<F, [♦x₁♦x₃x₄x₂♦¬x₆], no_c>
D U
U ¬x₅
<F, [♦x₁♦x₃x₄x₂♦¬x₆¬x₅], no_c>

We see that without restriction of rule usage the system can do unecessary work, mainly ignoring the conflict at hand and using other rules instead.

9.1.2. Complete DPLL

To describe modern SAT solvers we need to extend our rule set. Currently, the backtrack rule allows for only one level to revert. We also need to model the resolution step as a rule and enable clause learning which changes the formula in the state. These rules should not affect the theoretical termination and correction proofs. Additionally, we need to model clause forget and restarts, which will affect termination. Therefore, the proofs given at the end are only valid without the forget and restart extension.

ExplainUIP. This rule is part of the `analyze_conflict` function of the iterative algorithm, Listing 3. ExplainUIP is applicable as long as there are at least two literals in the conflict literal set which are in the current decision level. It stops when only one literal in the conflict set has the current decision level. In the worst case, that literal is the decision literal.

9.1.2.1 Definition (ExplainUIP Rule)
If there is a conflict with $l \in C$ (1) and there is a clause that caused l to become unit (2) with all other literals in the unit clause assigned before l (3), general resolution is applicable.
To ensure stopping at the first UIP, all other literals in the conflict set must have the same or lower decision level (4) and there have to be at least two literals in C at the current decision level (5).

$$\boxed{\dfrac{(1)\ l \in C \quad \text{and} \quad (2)\ l \vee \neg l_1 \vee ... \vee \neg l_k \in F \quad \text{and} \quad (3)\ l_1, ..., l_k < l \\ \text{and} \quad (4)\ \forall l' \in C : l' \leq l \quad \text{and} \quad (5)\ \exists l' \in C : \mathit{level}\ l' = \mathit{level}\ l, l' \neq l}{C := C \cup \{l_1, ..., l_k\} \backslash \{l\}}}$$

Learn. The negated disjunction of the conflict literal set can always be added to the clause database without modifying the satisfiability of the formula. Initially, the conflict set is a conflict clause which is already contained in the formula. Through resolution this conflict set changes and can be added to the formula with this rule.

9.1.2.2 Definition (Learn Rule)
If there is a conflict and the negated disjunction of the conflict literals are not already in the formula, the negated conflict literal clause can be added to the formula.

$$\boxed{\dfrac{C = \{l_1, ..., l_k\} \text{ and } \neg l_1 \vee ... \vee \neg l_k \notin F}{F := F \cup \{\neg l_1 \vee ... \vee \neg l_k\}}}$$

Backjump. The resolution process provides a lemma, a UIP, and a backjump level. Whenever these conditions are provided and the conflict set is not empty, backjump is applicable. It should be noted that backjumping forces the the lemma to be learned to be able to apply this rule.

9.1.2.3 Definition (Backjump Rule)
If there is a conflict and there is a literal $\neg l$ among a conflict clause in F with a highest decision level, backjump is possible.

$$\boxed{\frac{C = \{l, l_1, ..., l_k\} \text{ and } \neg l \vee \neg l_1 \vee ... \vee \neg l_k \in F \text{ and level } l > m \geq \text{level } l_i \text{ for (i = 1,..,k)}}{C := no\_cflct \text{ and } M := M^{[m]} + \neg l}}$$

With these rules added to the rule set, excluding the backtrack rule, the following result holds true. A full proof for this theorem can be found in [KG07].

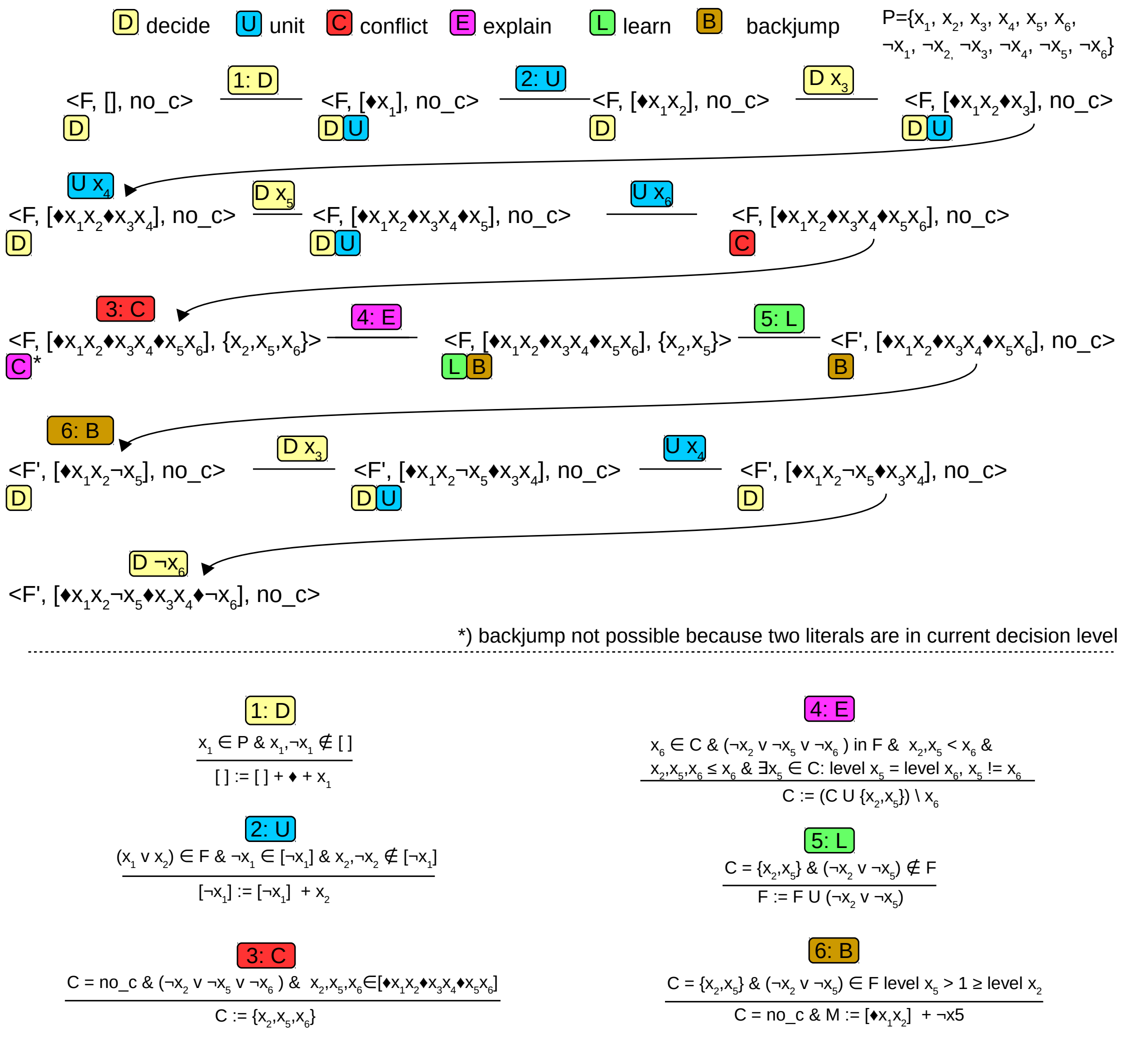
9.1.2.4 Theorem
All runs of DPLL are finite. If, initialized with the set of clauses F_{init}, DPLL terminates in the state $\langle F, M, C\rangle$, then: (1) C = no_cflct or C = $\emptyset$; (2) If C = $\emptyset$ then F_{init} is unsatisfiable; (3) If C = no_cflct, then M is a model for F_{init}.

The Example 9.1.2.5 demonstrates how this rule system operates on a specified input formula. Applicable rules are depicted directly under the state. Literal selection and rule application oder is arbitrary.

9.1.2.5 Example (Complete DPLL Run)

We use the complete rule set exhaustively until no rules can be applied anymore. Every time a rule is applied the first time, it is explained in detail below the DPLL run.

$$\text{Let } F = \{\{\neg x_1, x_2\}, \{\neg x_3, x_4\}, \{\neg x_5, x_6\}, \{\neg x_2, \neg x_5, \neg x_6\}\}$$


D decide
U unit
C conflict
E explain
L learn
B backjump
P={x1, x2, x3, x4, x5, x6, ¬x1, ¬x2, ¬x3, ¬x4, ¬x5, ¬x6}
<F, [], no_c>
1: D
<F, [♦x1], no_c>
2: U
<F, [♦x1x2], no_c>
D x3
<F, [♦x1x2♦x3], no_c>
U x4
<F, [♦x1x2♦x3x4], no_c>
D x5
<F, [♦x1x2♦x3x4♦x5], no_c>
U x6
<F, [♦x1x2♦x3x4♦x5x6], no_c>
3: C
<F, [♦x1x2♦x3x4♦x5x6], {x2,x5,x6}>
4: E
<F, [♦x1x2♦x3x4♦x5x6], {x2,x5}>
5: L
<F', [♦x1x2♦x3x4♦x5x6], no_c>
6: B
<F', [♦x1x2¬x5], no_c>
D x3
<F', [♦x1x2¬x5♦x3x4], no_c>
U x4
<F', [♦x1x2¬x5♦x3x4], no_c>
D ¬x6
<F', [♦x1x2¬x5♦x3x4♦¬x6], no_c>
*) backjump not possible because two literals are in current decision level
1: D
x1 ∈ P & x1,¬x1 ∉ [ ]
[ ] := [ ] + ♦ + x1
2: U
(x1 v x2) ∈ F & ¬x1 ∈ [¬x1] & x2,¬x2 ∉ [¬x1]
[¬x1] := [¬x1] + x2
3: C
C = no_c & (¬x2 v ¬x5 v ¬x6) & x2,x5,x6∈[♦x1x2♦x3x4♦x5x6]
C := {x2,x5,x6}
4: E
x6 ∈ C & (¬x2 v ¬x5 v ¬x6) in F & x2,x5 < x6 & x2,x5,x6 ≤ x6 & ∃x5 ∈ C: level x5 = level x6, x5 != x6
C := (C U {x2,x5}) \ x6
5: L
C = {x2,x5} & (¬x2 v ¬x5) ∉ F
F := F U (¬x2 v ¬x5)
6: B
C = {x2,x5} & (¬x2 v ¬x5) ∈ F level x5 > 1 ≥ level x2
C = no_c & M := [♦x1x2] + ¬x5

Lastly, we need to define restarts and clause forget. The usage of these rules has to be restricted in the implementation to guarantee termination.

9.1.2.6 Definition (Forget Rule)

If there is currently no conflict and there exists a clause which is implied by the formula without it, the clause can be removed.

$$\boxed{\dfrac{C = no\_cflct \text{ and } l_1 \vee \ldots \vee l_k \in F \text{ and } F \backslash \{l_1 \vee \ldots \vee l_k\} \models \{l_1 \vee \ldots \vee l_k\}}{F := F \backslash \{l_1 \vee \ldots \vee l_k\}}}$$

9.1.2.7 Definition (Restart Rule)

Restart is possible at all times provided there is currently no conflict.

$$\boxed{\dfrac{C = no\_cflct}{M := M^{[0]}}}$$

9.2. DPLL Overview

In summary, the non-deterministic transition system consists of these rules.

Decide	$\dfrac{l \in P \text{ and } l, \neg l \notin M}{M := M + \diamondsuit + l}$
Unit	$\dfrac{l \vee l_1 \vee \ldots \vee l_k \in F \text{ and } \neg l_1, \ldots, \neg l_k \in M \text{ and } l, \neg l \notin M}{M := M + l}$
Conflict	$\dfrac{C = no\_cflct \text{ and } \neg l_1 \vee \ldots \vee \neg l_k \in F \text{ and } l_1, \ldots, l_k \in M}{C := \{l_1, \ldots, l_k\}}$
ExplainUIP	$\dfrac{l \in C \text{ and } l \vee \neg l_1 \vee \ldots \vee \neg l_k \in F \text{ and } l_1, \ldots, l_k < l \text{ and } \forall l' \in C : l' \leq l \text{ and } \exists l' \in C : level\ l' = level\ l, l' \neq l}{C := C \cup \{l_1, \ldots, l_k\} \setminus \{l\}}$
Learn	$\dfrac{C = \{l_1, \ldots, l_k\} \text{ and } \neg l_1 \vee \ldots \vee \neg l_k \notin F}{F := F \cup \{\neg l_1 \vee \ldots \vee \neg l_k\}}$
Backjump	$\dfrac{C = \{l, l_1, \ldots, l_k\} \text{ and } \neg l \vee \neg l_1 \vee \ldots \vee \neg l_k \in F \text{ and level } l > m \geq \text{level } l_i \text{ for (i = 1,..,k)}}{C := no\_cflct \text{ and } M := M^{[m]} + \neg l}$
Forget	$\dfrac{C = no\_cflct \text{ and } l_1 \vee \ldots \vee l_k \in F \text{ and } F \setminus \{l_1 \vee \ldots \vee l_k\} \models \{l_1 \vee \ldots \vee l_k\}}{F := F \setminus \{l_1 \vee \ldots \vee l_k\}}$
Restart	$\dfrac{C = no\_cflct}{M := M^{[0]}}$

To more precisely model the behaviour of modern solvers, we can restrict the rule usage by a regular expression, so called *strategy*. The following regular expression describes the solver Chaff at a high-level.

$$\left(\left((Conflict; ExplainUIP^*; [Learn; Backjump]) \,||\, Unit^*\right); [Decide]^*\right)$$

The additional rules forget and restart can be placed right before decide.

10. Framework Analysis

In this section, we use our knowledge from previous sections about modern SAT solvers and the transition system to develop a modular SAT solver framework. Before we develop the actual solver, we need to think about how input and output is handled.

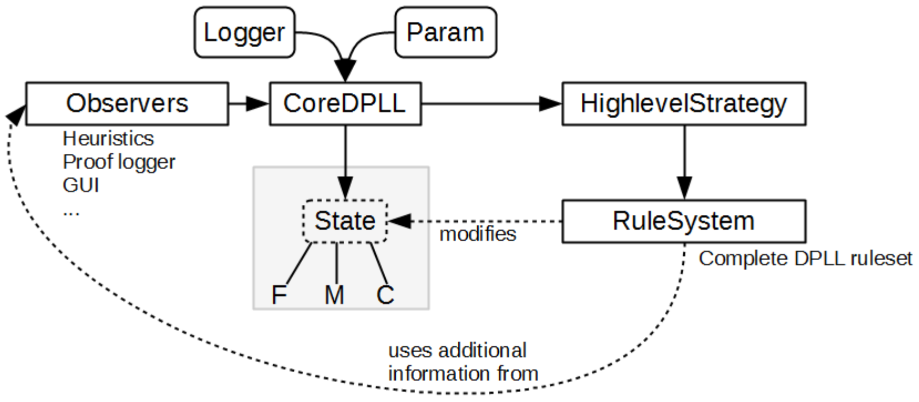
10.1 Remark (DIMACS Input, Output)
Most solvers accept input according to the DIMACS format[15]. In short, a file is divided in three sections. One comment section (line one to line five), one section containing information about clause count and variable count (line six), and the last section contains the clauses itself. A literal is written as an integer starting from one. ’0’ marks the end of a clause. Negation is denoted as ’-’. This format is handed out to solvers which have to be able to convert it into the internal data structure beforehand.

Listing 4: DIMCAS Format

```
1 c
2 c comments
3 c instance info
4 c sat/unsat
5 c etc
6 p cnf 5 3
7 1 -5 4 0
8 -1 4 3 4 0
9 -3 -4 0
```

The output format should be one line stating whether the instance is satisfiable or not (s SATISFIABLE, S UNSATISFIABLE). If the instance is satisfiable, append a second line with the variable assignments starting with the letter ’v’. In pure UNSAT tracks, it is required that the solver is able to produce an UNSAT proof in a special format. More information about that can be found on the official SAT competition page.

We start with an overview of the solver architecture. It is closely related to [Mar] with additional separation of modules for code clarity.



[15]SAT competition provides a tool to check if a file conforms to the DIMACS format at http://www.satcompetition.org/2013/files/CNFChecker.zip

Parameter Entry point of the solver framework. Handles parameter inputs and initializes solver.

CoreDPLL Main module which handles initialization of every component needed. Forms the core together with the HighlevelStrategy and the RuleSystem Module.

HighlevelStrategy Uses rules defined in the RuleSystem to implement an algorithm.

RuleSystem Every rule should be implemented here separately.

Logger Logs the solving process.

Observer Observers can include heuristics (decision, restart) which can used by the RuleSystem to guide the rule application. Other uses include GUI or proof logging for UNSAT proofs.

CoreDPLL manages all input given in form of arguments and initializes all important modules. CoreDPLL together with the HighlevelStrategy and RuleSystem module represent the transition system of [KG07].
We first need to make a clear distinction between modules which modify the state of the system and ones which only have to read it. All packages inside the Core — i.e. CoreDPLL, HighlevelStratey, and RuleSystem — must be able to modify the internal state via rules. All other modules only need to be able to read the state and have to get notified when a rule is applied. We achieve this by making DPLL observable, that is every module that wants to observe the transition system must implement a SolverListener interface. This interface contains all rules which are defined in [KG07]. By separating the modules this way, we achieve a separation of heavy computing functions which makes the code easier to understand.
On the one hand, we have the CoreDPLL which concerns itself with efficient data structures (formula, trail, efficient rule implementation). On the other hand, heuristics are able to observe the solving process independently to gather needed information. This ensures easy identification of bottlenecks.
This framework is build upon clearly defined interfaces for all classes. These implementations of interfaces are then instanced in their appropriate factories. Thus it is easy to develop and insert new modules into the framework. We will show some sample implementations of techniques introduced in previous sections.

10.1. Example Module Implementations

In this section, we will show how to use the framework to implement different modules specified by interfaces. We will implement: a simple counter-based data structure, a decision heuristic, a learn heuristic, a forget heuristic, and a restart heuristic[16].

10.1.1. Counter-Based Formula

To create a new formula data structure, the interfaces for *Clause, SetOfClauses*, and *Variable* have to be implemented. There are already abstract formula implementations for often used functions, which can be used as a starting point. These abstract implementations are designed to not be a bottleneck, as all important functions provided have a constant time consumption.
As described in 7.1.1, a counter-based clause keeps counters to indicate its status. They are initially zero. Additionally, we have to know how big the clause is and need to keep a reference to the parent formula. The size and the reference are already provided by the abstract data structure.

Listing 5: Clause Counter-Based Implementation

```
public class ClauseCbs extends ClauseAbs implements CounterBasedClause {
    private int satisfiedLit = 0;
    private int unsatisfiedLit = 0;

    ...
}
```

Concerning the state check, we need to explicitly implement the `isUnit` and `isConflict` methods. The functions are simple counter checks.

```
    public boolean isConflicting() { return (unsatisfiedLit == literals); }

    public boolean isUnit() { return (unsatisfiedLit == literals - 1)
                                && satisfiedLit == 0; }
```

To actually find the unit literal, we need to retrieve the complete literal list and search for the first and only unsatisfied literal. To make every clause cache-friendly (i.e. reduce the memory footprint as much as possible), the literals are saved separately from the clause and are identified by a unique identifier (Line 2). This mechanism is also provided by the abstract implementation.

```
1     public Integer getUnitLiteral() {
2         for (Integer l : F.getLiterals(id)) {
3             if (F.getVariable(l).getAssignment() == -1) {
4                 return l;
5             }
6         }
7     }
```

[16]The only module left which is needed for the framework to function correctly is the implementation of the trail interface and the rule set. Since the trail is quite large and the rule implementations are simple, we will leave these out.

To modify the state, we need to implement the counter-modifying functions. Excluding the case where the satisfied counter is increased, we also need to notify the formula if the clause becomes unit or conflict as a result. This small notify overhead is compensated heavily by the fact that, if not notified, the whole formula has to be searched to find the unit or conflict clauses again.

```java
    public void incrementSatisfied(){ satisfiedLit += 1; }

    public void incrementUnsatisfied() {
        unsatisfiedLit += 1;
        if (isUnit()) F.foundUnitClause(this);
        if (isConflicting()) F.foundConflictClause(this);
    }

    public void decrementSatisfied() {
        satisfiedLit -= 1;
        if (isUnit()) F.foundUnitClause(this);
    }

    public void decrementUnsatisfied() {
        unsatisfiedLit -= 1;
        if (isUnit()) F.foundUnitClause(this);
    }
```

Lastly, we need to think about how connections are handled when a clause is learned and forgotten. The abstract implementation handles variable connection removal. The only thing we need to ensure is that a clause is conflicting after it has been learned. A counter-based clause is conflicting if the unsatisfied literal counter equals the size of the clause.

```java
    public void assertConflictingState() {
        unsatisfiedLit = literals;
        satisfiedLit = 0;
    }
```

With these functions, the clause implementation is complete. We will now implement the counter-based Variable class. A counter-based variable keeps references to clauses which it is contained positively and negatively in. Therefore, we need two lists.

Listing 6: Variable Counter-Based Implementation

```java
public class VariableCbs extends VariableAbs implements CounterBasedVariable {
    private final List<CounterBasedClause> containedPositively = new ArrayList<>();
    private final List<CounterBasedClause> containedNegatively = new ArrayList<>();

    ...
}
```

Next are the state modifying functions. The functions for set true, set false, and undo assignments have to be implemented. The abstract implementation handles the assignment side. Additionally, we need to update the states for all clauses in which this variable is contained. If a variable is set to true, visit every negatively contained clause and increase the unsatisfied counter (Line two), and every positively contained clause and increase the satisfied counter (Line three). If a variable is set to false, switch the visit order. If an assignment is undone, revert the changes accordingly.

```
1     public void setTrue() {
2         containedNegatively.forEach(CounterBasedClause::incrementUnsatisfied);
3         containedPositively.forEach(CounterBasedClause::incrementSatisfied);
4 
5         setAssignment(1);
6     }
7 
8     public void setFalse() {
9         containedPositively.forEach(CounterBasedClause::incrementUnsatisfied);
10        containedNegatively.forEach(CounterBasedClause::incrementSatisfied);
11
12        setAssignment(0);
13    }
14
15    public void undoAssignment() {
16        if (getAssignment() == 1) { // Case 1
17            containedPositively.forEach(CounterBasedClause::decrementSatisfied);
18            containedNegatively.forEach(CounterBasedClause::decrementUnsatisfied);
19        }
20        else if (getAssignment() == 0) { // Case 2
21            containedNegatively.forEach(CounterBasedClause::decrementSatisfied);
22            containedPositively.forEach(CounterBasedClause::decrementUnsatisfied);
23        }
24
25        setAssignment(-1);
26    }
```

Lastly, we need to implement connect and remove functions to respond to clause connections. We remove/insert the clauses into the two lists correctly.

```
public void connectToClausePositive(CounterBasedClause cl) {
    containedPositively.add(cl);
}

public void connectToClauseNegative(CounterBasedClause cl) {
    containedNegatively.add(cl);
}

public void removeConnections(Integer l, Clause cl) {
    CounterBasedClause cb = (CounterBasedClause) cl;
    if(l < 0){
        containedNegatively.remove(cb);
    }
    else{
        containedPositively.remove(cb);
    }
}
```

To anticipate mixed data structures, we need to declare an interface for the counter-based data structure. This interface should have every function not already in the clause/variable interface. This way, we can create a mixed variable/clause class which uses these interface functions accordingly.

The last thing we need to implement is the SetOfClauses interface. It mostly concerns itself with correct input conversion and correct clause learning/forgetting. As this is not as interesting to implement as the variable and clause interface, the implementation with comments can be found in the Appendix A. With these three classes, we have a complete and compact counter-based implementation.

10.1.2. MiniSAT Decision Heuristic

As a close relative to the VSIDS heuristic, we will implement the heuristic used in MiniSAT. The behavior of MiniSAT is described in 7.2.2.

As a dynamic decision heuristic, we need to implement the VariableSelection interface as well as the SolverListener interface. To respond to state changes, we need to add this heuristic to the observer list. As this heuristic is based on *variable scores*, we need to create an additional structure which keeps track of said scores. This is called the activity class. In short, the activity class keeps counters and provides methods to bump a variable activity and decay all scores. How to implement this structure can be found in [Mar], although it is described for C++ and not Java. And lastly, we need to keep these variables *sorted by activity* at all times, to enable fast retrieval of unassigned literals with the highest score. As an initial data structure we take the PriorityQueue implementation in Java[17].

Listing 7: MiniSAT Decision Heuristic Implementation

```
public class MiniSATVar implements SolverListener, VariableSelectionStrategy {
    private final SetOfClauses F;
    private ActivitiesVariable activities;
    private PriorityQueue<Integer> activityQueue;

    public MiniSATVar(Solver solver) {
        F = solver.getState().F();
        solver.addObserver(this);
    }

    ...
}
```

Concerning initialization, the observer interface provides a function to notify when an initial clause is added and when the formula has loaded completely. We use this to increase the score of the initial clauses' variables accordingly (Line three, Line fourteen).

```
1      private List<Integer> toBump = new ArrayList<>();
2
3      public void onLearnInitial(List<Integer> cl) { toBump.addAll(cl);}
4
5      public void onSetOfClausesLoaded() {
6          int size = F.getFreeVariablesCount();
7          activities = new ActivitiesVariable(size);
8          activityQueue = new PriorityQueue<>(size, new ActComparator(activities));
9          // Initially, all variables are free; add every variable to the activity queue
10         for (Integer i = 0; i < size; i++) {
11             activityQueue.offer(abs(i));
12         }
13
14         bumpVarsInClause(toBump);
15     }
```

[17]We will later see why this is a bad decision when analyzing this framework for bottlenecks.

To retrieve an undefined variable with the highest score, we need to poll the queue until a free variable is found. This variable is ensured to have the highest score because of the properties of the priority queue.

```
public Variable getVariable() {
    Integer var = activityQueue.poll();

    boolean undef = F.isVariableUndefined(var+1);
    while(!undef){
        var =  activityQueue.poll();
        undef = F.isVariableUndefined(var+1);
    }

    return F.getVariable(var+1);
}
```

It should be noted that the variables in the framework start at index 1. This is to model literals as integers, which enables easy negation checks. That is why we need to shift the index by +1/-1 in this heuristic.
The last thing we need to decide on is how to respond to state changes. MiniSAT prefers variables which are in recent conflict and which are involved in resolution steps. It also periodically decays scores after a conflict occurred. Therefore, we need use respective observer functions for conflict and resolution. Additionally, we need to re-insert variables after the corresponding literal gets backtracked.

```
public void onExplain(List<Integer> ante, Integer resLit,
                      List<Integer> resolved) {
    bumpVarsInClause(resolved);
}

public void onConflict(List<Integer> cl) {
    activities.decayAll();
    bumpVarsInClause(cl);
}

public void onBacktrack(Integer l) {
    if (!activityQueue.contains(abs(l)-1)) {
        activityQueue.add(abs(l)-1);
    }
}
```

The function to bump a variable is left. After the variable has its score increased, the heap has to be updated. Since PriorityQueue does not provide such a functionality, we need to remove and re-insert the element again (Line nine and ten).

```
1     private void bumpVarsInClause(List<Integer> cl) {
2         for (Integer lit : cl) bumpVariableActivity(abs(lit));
3     }
4
5     private void bumpVariableActivity(int var) {
6         activities.bump(var-1);
7
8         if(activityQueue.contains(var-1)){
9             activityQueue.remove(var);
10            activityQueue.add(var);
11        }
12    }
```

This concludes the implementation of a decision heuristic in this framework. What we implemented can be seen as an effective heuristic skeleton, because heuristics based on the VSIDS heuristic can be derived from this with small changes. We can produce the VSIDS heuristic by increasing scores whenever a clause is added instead of when a conflict occurs. Since VSIDS selects *literals* instead of *variables*, we need to change the activity class to accommodate literals also.

10.1.3. Other Modules

In this section we will briefly talk about other modules contained in the framework.

Trail The Trail interface has many functions which need to be implemented. Since the implementation of the trail is not interesting, we decide to leave out the implementation of it. As this is one of the core modules, this is needed for the framework to function.

Forget, Restart Sample implementations can be found in Part B and C of the Appendix respectively.

Preprocess There is currently no preprocessing implemented in the framework.

11. Results

The framework we developed has two major goals. Firstly, we want an implementation of a solid foundation of a SAT solver framework based on a transition system. It can then be used to compare different techniques and develop new components. Secondly, we want to use the framework as a teaching tool to get new students into the propositional testing field. As the gap of understanding efficient solvers and university teaching widens, it becomes increasingly more difficult to teach it in a compact manner. We will discuss the framework fares as a teaching tool, analyze the initial benchmarks, and make conclusion based on initial hypothesis.

11.1. Seminar Results

One goal of this framework is to teach students modern SAT techniques. The idea is that students can use a bare bone framework to implement the necessary solver parts and then pick an area (efficient trail, data structure, heuristics, etc.) to concentrate on. Here, the framework must ensure that the communication of the modules works correctly and without introducing hidden bottlenecks for the students.
To make use of this framework, we held a seminar about modern SAT solvers. The structure was as follows:

Recursive DPLL. The basis for the seminar was knowledge from a lecture about propositional logic and first-order predicate calculus. Students already knew how to apply the Davis-Putnam algorithm on a theoretical level on small instances. At this point, the notion that one needs to implement a data structure for the algorithm to function was much less pronounced than it should be. Therefore, the first step was to let the students implement their own recursive solver based on the knowledge they already had. This ensured that all participants refreshed their programming language skills and had their own working solver after the first phase.

Iterative DPLL. The second phase was used to change the recursive algorithm to an iterative one. The trail structure was introduced together with a basic part of the framework. The idea was that after the second phase, all important components would already be understood and extended in the third phase, in which efficient concepts were introduced. In short, this phase was meant as an introduction to the framework and as preparation for the theoretical transition system.

Transition System. Lastly, the transition system was defined and explained. The framework was formally introduced with all its components and interactions. The task for the students was to implement the most efficient solver in the remaining time, by selecting which basic function to implement and picking areas to specialize on.

Report. After the seminar, students were told to write a short summary on what they have learned and to give feedback on what to improve.

The result was that students implemented two fully functional solvers after six weeks. One recursive solver, and one modern iterative one. The iterative solvers were then run on selected instances. We will analyze the result in the next chapter.

11.2. Seminar Benchmarks

For benchmarking, we will use instances provided by SATLIB[18]. Although these instances are somewhat dated, they are used as a first step to find out if the solver behaves as expected. Since this framework is made from scratch, this will also help to find remaining bottlenecks in the design.
The test specification is as follows. A first generation Intel Core i7 2,97 GHz together with 1GB RAM was used to run the tests. Each instance was solved one time with a timeout of one hour. Instances were randomly selected with the addition of the hardest ones from each domain. This quantitative test is used to show how many instances are solved in general.

[18]http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

Robustness Tests. Without going into further detail about the standard parameters, the following table presents the results for benchmarking 124 instances[19]. Our expectancy was that it solves most of the problems instances, with industrial benchmarks being solved efficiently.

Domain Group	Instances	Solved	Total Time (m)	Avg. Time (m)
Flat Graph Colouring	8	8	3.5	0.43
DIMACS Benchmarks	44	35	547.9	12.45
Uniform Random 3SAT	18	16	167.4	9.3
SAT Competition Beijing	10	8	121.9	12.9
Morphed Graph Colouring	6	6	0.04	<0.01
Random 3SAT (other)	7	7	0.05	<0.01
Planning	8	7	63.5	7.9
All Interval Series	4	4	0.69	0.17
Quasigroup Instances	9	6	184.5	20.5
Bounded Model Checking	10	6	354.0	35.4

The overall solving time exceeds a day. Given how dated these instances are, the result is quite modest. Especially the bounded model checking group shows unexpected results, because our solver should be specialized in solving industrial instances and did not solve half of them. The same result showed in other groups, too. With minor differences between instances, the overall high time consumption is a similarity across all implemented solvers in the seminar.
These results indicate heavy bottlenecks. Before looking into some hypothesis we need to revisit the implementation and remove most of the hidden bottlenecks. This is done by using VisualVM which is able to show how much computing time is spent per function. What follows are fixed bottlenecks in no particular order.

Trail Implemented a new trail module. The old trail had naive implementations of functions which had a linear runtime in the size of the trail. Most of them now have constant runtime.

HighlevelStrategy BCP Unit propagation did not stop when a conflict occurred. This led to unnecessary assignments which had to be backtracked again, too. Unit application now stops whenever a conflict occurs.

FindUnitClause Previously, the clause database had to be searched to find unit and conflict clauses. Now the formula is notified whenever a conflict and unit clause is found preventing the search of the clause database.

Cache-Friendly Clauses Literals are now separated from the clause memory footprint, which reduces cache misses.

[19]Selected instances with more info can be found in Appendix D.

Learn Guard The expensive linear runtime search for the learn guard is now completely removed. The learn guard is unnecessary if the CHAFF high-level strategy is used, as mentioned in [KG07].

Java PriorityQueue Java's PriorityQueue implementation has a linear runtime behavior for contain checks. We need a constant runtime for the contain function. The heap structure implemented in [Mar] provides needed runtime constraints.

With these improvements, the benchmarks are done again with the same hardware specification. Our expectation is that most of the instances are solved before timeout. The exceptions are large random problems which are not focus of the solver. The results are as follow. We excluded timeouts in the average time if they occurred in both runs, which is marked by '*'.

Domain Group	#	Solved	Total (m)	Avg. (s)	Old Avg. (s)
Flat Graph Colouring	8	8	0.3	0.25	25.8
DIMACS Benchmarks	44	35	540	1.1*	13.6*
Uniform Random 3SAT	18	16	125	16.8*	177.8*
SAT Competition Beijing	10	9	82	147*	413*
Morphed Graph Colouring	6	6	<0.01	0.014	0.428
Random 3SAT (other)	7	7	0.01	0.12	0.44
Planning	8	8	2.9	21.6	476.3
All Interval Series	4	4	0.03	0.43	10.4
Quasigroup Instances	9	9	62	413	1230
Bounded Model Checking	10	9	133	486*	2104*

We can see that most instances are solved. With the exception of random instances and hard graph problems (DIMACS Benchmarks, Uniform Random 3SAT) we reached our goal of solving most instances. We will now look into following hypotheses, which are believed to be true and backed up by many experiments. Our framework should show the same results.

1. Non-chronological backtracking solves industry instances significantly faster than chronological backtracking. Chronological backtracking solves some instances faster by a considerable amount.
2. A dynamic decision heuristic (MiniSAT) performs better than a static one (SLIS).
3. Restarts reduce deviations from the average solving time.
4. Clause forget improves the solver's performance on large instances.

We select some instances from Appendix D. An Intel Core i7-2630QM @ 2.0 GHZ is used where 512MB RAM is allocated for the benchmark process. The tests are run

50 times. The average time and number of timeouts are then used for the results. The best and worst try is excluded in the average time. We will use abbreviations for the parameters used in benchmarks. The abbreviations are described in the following table. The forget heuristics are described in more detail shortly. Since we currently only have the counter-based data structure implemented, we will not mention the formula data structure parameter.

Strategy	Decision Heuristic
Chronological backtracking (CB)	MiniSAT(MS)
Non-chronological backtracking (NCB)	Static Largest Integer Sum (SLIS)

Forget Heuristic	Restart Heuristic
Do not forget clauses (fn)	Do not restart (RN)
Forget random large clauses (fl)	Restart Fixed 700 (RF700)
Forget random small clauses (fsh)	Restart Fixed 2000 (RF2000)
Forget random clauses fixed (fs)	Restart Geometric (RG)
Forget random clauses size (fsize)	

The forget heuristics fl, fsh, and fs forget clauses with a chance of 33% after a fixed number of clauses have been learned. This constant is set to 1000. The heuristic fsize instead responds dynamically to the formula by forgetting clauses randomly 33% of the time whenever the number of the learned clauses gets as big as the number of the initial clauses.

1. CB vs NCB. The parameters are selected as follows: (CB/NCB), MS, RG, fn. Time is noted in seconds. Our expectation is that on large industrial instances[20] NCB severely outperforms CB.

Instance	#Variables	#Clauses	SAT	NCB	T/O	CB	T/O
`bmc-galileo-8`	58074	294821	true	**525**	18	>1000	50
`bmc-ibm-1`	9685	55870	true	**18**	0	>1000	50
`ssa7552-160`	1391	3126	true	0.012	0	0.011	0
`ii16c2`	924	13803	true	**0.29**	0	23.12	0
`4blocks`	758	47820	true	**4.5**	0	449.5	5
`flat200-100`	600	2237	true	1.32	0	**0.30**	0
`uuf250-090`	250	1065	false	153	0	**33**	0
`uf100-0896`	100	430	true	0.037	0	**0.018**	0

Industrial instances are clearly solved faster by non-chronological backtracking, to the point where chronological backtracking is not able to solve them (`bmc-ibm-1` and

[20]Large is seen in context to the SATLIB 2002 benchmarks. Here, large means usually around 1000 variables and more than 50000 clauses.

`bmc-galileo-8`). The gap lessens for graph instances and other instances, where analyzing the conflict does not provide important information (`ssa7552-169` and `flat200-100`). Here, CB is approximately as fast as or faster than NCB. Random satisfiable and unsatisfiable instances are solved considerably faster with chronological backtracking (uuf250-090 and uf100-0896).

2. MiniSAT vs SLIS. The parameters are selected as follows: NCB, (MS/SLIS), RG, fn. Static Largest Integer Sum is a decision heuristic which uses the initial literal count as a score to select decision literals. It does not respond dynamically to the solving process. We expect that the dynamic decision heuristic outperforms the static one on most instances.

Instance	MS	T/O	SLIS	T/O
`bmc-galileo-8`	**525**	18	985	45
`bmc-ibm-1`	**18**	0	858	36
`ssa7552-160`	0.012	0	0.014	0
`ii16c2`	**0.29**	0	1.72	0
`4blocks`	**4.5**	0	8.9	0
`flat200-100`	**1.32**	0	2.62	0
`uuf250-090`	**153**	0	367	0
`uf100-0896`	0.037	0	0.045	0

We observe that dynamically responding to state changes overall improves the solving speed. This is especially visible on industry instances (`bmc-ibm-1` and `bmc-galileo-8`), where a big speedup is achieved.

3. Restarts The parameters are selected as follows: NCB, MS, (RN/RG/RF700/RF2000), fn. We expect that restarts improve the overall solving time over fifty runs.

Instance	RN	T/O	RF700	T/O	RF2000	T/O	RG	T/O
`bmc-galileo-8`	600	20	657	20	526	13	525	18
`bmc-ibm-1`	36	0	19	0	21	0	**18**	0
`ssa7552-160`	0.012	0	0.011	0	0.012	0	0.012	0
`ii16c2`	0.42	0	0.33	0	0.47	0	**0.29**	0
`4blocks`	8.9	0	**3.9**	0	5.0	0	4.5	0
`flat200-100`	1.56	0	2.34	0	2.47	0	**1.32**	0
`uuf250-090`	186	0	193	0	225	0	**153**	0
`uf100-0896`	0.045	0	0.042	0	0.047	0	**0.037**	0

Again, the hypothesis is in accordance with our result. Geometric restarts reduce the solving time — on average and compared to not using restarts at all — by a considerable amount. In the worst case, they do not affect the solving speed (`ssa7552-160`).

4. Forget Heuristics The parameters are selected as follows: NCB, MS, RG, (fn/fs/fsh/fsize). Our expectation is that at least one forget heuristic improves solving time on large industry instances.

Instance	fn	fs	fl	fsh	fsize
`bmc-galileo-8`	525	545	**489**	498	564
`bmc-ibm-1`	17.9	18.5	24.9	**14.5**	21.4
`ssa7552-160`	0.012	0.011	0.012	0.012	0.012
`ii16c2`	0.29	0.33	0.32	0.29	0.42
`4blocks`	4.5	**3.9**	5.6	6.2	4.3
`flat200-100`	**1.32**	2.34	2.24	2.62	2.07
`uuf250-090`	**153**	193	189	189	199
`uf100-0896`	0.037	0.042	0.039	0.044	0.041

We can observe that, on average, forget heuristics seem to slow the solver down. The only instances which yield a time improvement are large industrial ones (`bmc-ibm-1` and `bmc-galileo-8`). These results reinforce the fact that forget heuristics are hard to tune.

To conclude the analyzing chapter, we will revisit claims we made throughout this work.

Problems with the recursive algorithm: At the end of Section 6.1 we made four different claims on why the recursive algorithm was not suited for industry use.

Although this was not in the scope of this work, the recursive solvers implemented in the seminar showed that they were only able to solve the easiest instances of the benchmark set. Additionally, groups which implemented the pure rule were significantly slower than other groups which only implemented the efficient unit rule. These observations validate the claims that the recursive version is inefficient and that classical deduction mechanisms — mainly the pure rule — are not suited for industrial application.

Concerning the efficiency of classical backtracking, we tested it in this chapter ("*1. CB vs NCB*"). We came to the conclusion that non-chronological backtracking is indeed needed to solve large industry instances.

Lastly, we made the claim that classical data structures slow down the solver for large instances derived from application domains. This claim is left for future work, because we have to implement the two-watched-literals data structure again after fixing most of the framework's bottlenecks.

Non-chronological backtracking and unsatisfiable instances: At the end of Chapter 6 (see Example 6.2.3.3) we ended with reasons why NCB is more useful on unsatisfiable instances. As SATLIB provided mostly satisfiable industrial instances, we could not test this claim. This is also left to future work.

12. Conclusion and Future Work

We implemented a modular SAT solver framework based on an abstract transition system proposed by [KG07]. Most of the techniques introduced in Part II are implemented and benchmarked.

While generally the framework was well received by the participating students, there are aspects which can be improved. The documentation has to be updated to accommodate the changes we made in the architecture of the framework. Additionally, unit tests should be provided for the students to help testing and integrating new modules. All in all, the feedback has led to a generally cleaner and more efficient framework structure. This should be helpful for other seminars to come.

The initial benchmarking showed that there was room for improvement, especially finding unnecessary bottlenecks. We analyzed the framework and removed most bottlenecks. We now have a solid foundation for a modular SAT solver framework. We can now concentrate on implementing more modules, as more and more new efficient techniques get proposed by new solvers. We can also think about how to use the modularity for its advantage, by determining which modules can be used either before or during runtime. More research has to be done on this matter.

Part IV. Appendix

A. Counter-Based SetOfClauses Implementation

The abstract SetOfClauses implementation handles conversion from a DIMACS file. As a result, the initial literals are provided as a simple list during conversion. We need to instance a new counter-based clause and connect it correctly to the variables in it.
The only difference between adding an initial clause and a learned clause is as follows: An initial clause can be unit at the start (Line five). A learned clause should always be conflicting (Line ten). We will implement a more general `addCl` function (Line thirteen) and use it for adding initial and learned clauses (Line three and eight). Instantiating arrays is left out because it is trivial.

Listing 8: SetOfClauses Counter-Based Implementation

```
 1 public class SetOfClausesCbs extends SetOfClausesAbs {
 2 
 3     public void addClause(List<Integer> cl) {
 4         Clause n = addCl(cl);
 5         if (n.isUnit()) foundUnitClause(n);
 6     }
 7 
 8     public void addLearnedClause(List<Integer> cl) {
 9         Clause n = addCl(cl);
10         n.assertConflictingState();
11     }
12 
13     private Clause addCl(List<Integer> cl) {
14         // only for instancing the initial formula
15         ClauseCbs n = new ClauseCbs(solver);
16         for (int l : cl) {
17             VariableCbs var = (VariableCbs) getVariables()[abs(l)];
18 
19             if (l < 0) {
20                 var.connectToClauseNegative(n);
21             } else {
22                 var.connectToClausePositive(n);
23             }
24 
25         }
26 
27         n.setId(clauses.size());
28         n.literals = cl.size();
29         initialClauses.add(n);
30         clauses.add(cl);
31 
32         return n;
33     }
34     ...
35 }
```

B. Forget Heuristic Implementation

Since we do not know which global property of clauses can be used to identify important clauses effectively (see 7.3.2), we decide to forget clauses after a certain amount has been learned randomly.
As with the decide heuristic, we need to observe the solving process. We forget clauses (in this case small clauses) if the learned clause count exceeds 100 with a certain percentage.

Listing 9: Forget Heuristic Implementation

```
public class ForgetRandomShort implements ForgetStrategy,SolverListener {
    private int learnClauseCount = 0;
    private final Random r;
    private final SetOfClauses F;

    public ForgetRandomShort(Solver solver) {
        F = solver.getState().getF();
        solver.addObserver(this);
        r = new Random();
    }

    public boolean shouldForget() {
        return learnClauseCount > 100;
    }

    public ArrayList<Clause> forgetClauses() {
        learnClauseCount = 0;

        List<Clause> cl = F.getLearnedClauses();
        ArrayList<Clause> forgot = new ArrayList<>();
        for (Clause c : cl) {
            float chance = r.nextFloat();
            if (c.getLiterals().size() <= 5) {
                if (chance <= 0.50f) {
                    forgot.add(c);
                }
            }
        }
        forgot.forEach(F::forgetLearnedClause);

        return forgot;
    }


    public void onLearn(List<Integer> cl) {
        learnClauseCount++;
    }

    ...

}
```

C. Restart Heuristic Implementation

We will implement a linear increasing restart heuristic (see 7.3.1).

The main idea behind conflict restart heuristics are keeping a conflict threshold (’`conflictsForNextRestart`’, Line four and thirty one) and modifying it by a constant if needed (’`conflictsForNextRestartConst`’, Line thirty five). The initial threshold can be chosen freely (Line thirty one).

Listing 10: Restart Heuristic Implementation

```
1  public class RestartConflictCountingFixed implements RestartStrategy,
2          SolverListener {
3      //can be set by param class when parsing program arguments
4      public static Integer NEXTRESTART = 10;
5  
6      private Integer conflictCount;
7      private Integer conflictsForNextRestart;
8      private Integer conflictsForNextRestartConst;
9  
10     public RestartConflictCountingFixed(@NotNull Solver solver) {
11         solver.addObserver(this);
12         conflictCount = 0;
13         conflictsForNextRestart = 0;
14         conflictsForNextRestartConst = 0;
15         calculateConflictForFirstRestart();
16     }
17 
18     public boolean shouldRestart() {
19         return conflictCount >= conflictsForNextRestart;
20     }
21 
22     public void onRestart() {
23         conflictCount = 0;
24         calculateConflictsForNextRestart();
25     }
26 
27     private void calculateConflictForFirstRestart() {
28         conflictsForNextRestartConst = 0;
29         // Berkmin: 550 // Zchaff: 700
30         // Eureka: 2000 // Siege: 20000
31         conflictsForNextRestart = NEXTRESTART;
32     }
33 
34     private void calculateConflictsForNextRestart() {
35         conflictsForNextRestart += conflictsForNextRestartConst;
36     }
37 
38     public void onConflict(List<Integer> cl) {
39         conflictCount++;
40     }
41 
42     ...
43 }
```

D. Selected Benchmark Instances

A selection of benchmark instances from SATLIB. These instances were run with the hardware specification provided in 11.2. Out of 124 instances, 111 could be solved with a timeout of one hour.

Name	SAT?	Time (s)
All intervall series 4/ais10.cnf	TRUE	<1
All intervall series 4/ais12.cnf	TRUE	<1
All intervall series 4/ais6.cnf	TRUE	<1
All intervall series 4/ais8.cnf	TRUE	<1
AIM 10/sat/aim-100-2_0-yes1-4.cnf	TRUE	<1
AIM 10/sat/aim-100-6_0-yes1-2.cnf	TRUE	<1
AIM 10/sat/aim-100-6_0-yes1-4.cnf	TRUE	<1
AIM 10/sat/aim-200-2_0-yes1-3.cnf	TRUE	<1
AIM 10/sat/aim-50-1_6-yes1-3.cnf	TRUE	<1
AIM 10/unsat/aim-100-1_6-no-2.cnf	FALSE	<1
AIM 10/unsat/aim-100-2_0-no-2.cnf	FALSE	<1
AIM 10/unsat/aim-200-1_6-no-1.cnf	FALSE	<1
AIM 10/unsat/aim-200-2_0-no-1.cnf	FALSE	<1
AIM 10/unsat/aim-200-2_0-no-4.cnf	FALSE	<1
BF 3/bf0432-007.cnf	FALSE	<1
BF 3/bf1355-075.cnf	FALSE	<1
BF 3/bf2670-001.cnf	FALSE	<1
DUBOIS 4/dubois100.cnf	FALSE	<1
DUBOIS 4/dubois20.cnf	FALSE	<1
DUBOIS 4/dubois50.cnf	FALSE	<1
GCP 3/g125.17.cnf	TO	TO
GCP 3/g250.15.cnf	TO	TO
GCP 3/g250.29.cnf	TO	TO
HANOI 2/hanoi4.cnf	TRUE	11
HANOI 2/hanoi5.cnf	TO	TO
IL 3/ii16c2.cnf	TRUE	<1
IL 3/ii32b3.cnf	TRUE	<1
IL 3/ii8a2.cnf	TRUE	<1
JNH 4/jnh1.cnf	TRUE	<1
JNH 4/jnh16.cnf	FALSE	<1
JNH 4/jnh217.cnf	TRUE	<1
JNH 4/jnh309.cnf	FALSE	<1
LRAN 3/f1000.cnf	TO	TO
LRAN 3/f2000.cnf	TO	TO
LRAN 3/f600.cnf	TO	TO
PARITY 4/par16-3-c.cnf	TRUE	3
PARITY 4/par32-5-c.cnf	TO	TO
PARITY 4/par8-1.cnf	TRUE	<1
PARITY 4/par8-4-c.cnf	TRUE	<1
PHOLE 3/hole10.cnf	TO	TO
PHOLE 3/hole6.cnf	FALSE	<1
PHOLE 3/hole8.cnf	FALSE	21
PRET 3/pret150_40.cnf	FALSE	<1
PRET 3/pret150_75.cnf	FALSE	<1
PRET 3/pret60_60.cnf	FALSE	<1
SSA 3/ssa0432-003.cnf	FALSE	<1
SSA 3/ssa7552-158.cnf	TRUE	<1
SSA 3/ssa7552-160.cnf	TRUE	<1
Flat graph colouring 8/flat100-2.cnf	TRUE	<1
Flat graph colouring 8/flat175-15.cnf	TRUE	<1
Flat graph colouring 8/flat175-2.cnf	TRUE	<1
Flat graph colouring 8/flat200-1.cnf	TRUE	<1
Flat graph colouring 8/flat200-100.cnf	TRUE	1
Flat graph colouring 8/flat30-52.cnf	TRUE	<1
Flat graph colouring 8/flat30-68.cnf	TRUE	<1
Flat graph colouring 8/flat75-53.cnf	TRUE	<1
morphed graph colouring 6/sw100-2.cnf	TRUE	<1
morphed graph colouring 6/sw100-26.cnf	TRUE	<1
morphed graph colouring 6/sw100-27.cnf	TRUE	<1
morphed graph colouring 6/sw100-47.cnf	TRUE	<1
morphed graph colouring 6/sw100-7.cnf	TRUE	<1
morphed graph colouring 6/sw100-82.cnf	TRUE	<1
Planning 8/bw_large.a.cnf	TRUE	<1
Planning 8/bw_large.b.cnf	TRUE	<1
Planning 8/bw_large.c.cnf	TRUE	6
Planning 8/bw_large.d.cnf	TRUE	166
Planning 8/logistics.a.cnf	TRUE	<1
Planning 8/logistics.b.cnf	TRUE	<1
Planning 8/logistics.c.cnf	TRUE	<1
Planning 8/logistics.d.cnf	TRUE	<1
Backbone-Minimality 7/BMS_k3_n100_m429_10.cnf	TRUE	<1
Backbone-Minimality 7/BMS_k3_n100_m429_118.cnf	TRUE	<1
Backbone-Minimality 7/BMS_k3_n100_m429_229.cnf	TRUE	<1
Backbone-Minimality 7/BMS_k3_n100_m429_321.cnf	TRUE	<1
Backbone-Minimality 7/BMS_k3_n100_m429_76.cnf	TRUE	<1
Backbone-Minimality 7/RTI_k3_n100_m429_106.cnf	TRUE	<1
Backbone-Minimality 7/RTI_k3_n100_m429_38.cnf	TRUE	<1
SAT Competition Bejing 10/2bitadd_10.cnf	TO	TO
SAT Competition Bejing 10/2bitmax_6.cnf	TRUE	<1
SAT Competition Bejing 10/3bitadd_31.cnf	TRUE	1274
SAT Competition Bejing 10/3blocks.cnf	TRUE	<1
SAT Competition Bejing 10/4blocks.cnf	TRUE	35
SAT Competition Bejing 10/4blocksb.cnf	TRUE	<1
SAT Competition Bejing 10/e0ddr2-10-by-5-1.cnf	TRUE	3
SAT Competition Bejing 10/enddr2-10-by-5-8.cnf	TRUE	2
SAT Competition Bejing 10/ewddr2-10-by-5-1.cnf	TRUE	2
SAT Competition Bejing 10/ewddr2-10-by-5-8.cnf	TRUE	3
Bounded Model Checking 10/bmc-galileo-8.cnf	TRUE	1157
Bounded Model Checking 10/bmc-ibm-1.cnf	TRUE	7
Bounded Model Checking 10/bmc-ibm-10.cnf	TO	TO
Bounded Model Checking 10/bmc-ibm-11.cnf	TRUE	1122
Bounded Model Checking 10/bmc-ibm-13.cnf	TRUE	1511
Bounded Model Checking 10/bmc-ibm-2.cnf	TRUE	<1
Bounded Model Checking 10/bmc-ibm-3.cnf	TRUE	7
Bounded Model Checking 10/bmc-ibm-4.cnf	TRUE	566
Bounded Model Checking 10/bmc-ibm-5.cnf	TRUE	<1
Bounded Model Checking 10/bmc-ibm-7.cnf	TRUE	<1
Sat-encoded Quasigroup instances 9/qg1-07.cnf	TRUE	<1
Sat-encoded Quasigroup instances 9/qg2-07.cnf	TRUE	<1
Sat-encoded Quasigroup instances 9/qg3-08.cnf	TRUE	<1
Sat-encoded Quasigroup instances 9/qg3-09.cnf	FALSE	1379
Sat-encoded Quasigroup instances 9/qg4-08.cnf	FALSE	2
Sat-encoded Quasigroup instances 9/qg5-12.cnf	FALSE	5
Sat-encoded Quasigroup instances 9/qg5-13.cnf	FALSE	2329
Sat-encoded Quasigroup instances 9/qg6-09.cnf	TRUE	<1
Sat-encoded Quasigroup instances 9/qg7-11.cnf	FALSE	3
Uniform Random-3-SAT 18/sat/uf100-01.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-055.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-056.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-066.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-0671.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-0672.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-0679.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-0892.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-0894.cnf	TRUE	<1
Uniform Random-3-SAT 18/sat/uf100-0896.cnf	TRUE	<1
Uniform Random-3-SAT 18/unsat/uuf100-06.cnf	FALSE	<1
Uniform Random-3-SAT 18/unsat/uuf125-087.cnf	FALSE	<1
Uniform Random-3-SAT 18/unsat/uuf200-06.cnf	FALSE	48
Uniform Random-3-SAT 18/unsat/uuf250-0100.cnf	TO	TO
Uniform Random-3-SAT 18/unsat/uuf250-08.cnf	TO	TO
Uniform Random-3-SAT 18/unsat/uuf250-090.cnf	FALSE	221
Uniform Random-3-SAT 18/unsat/uuf75-02.cnf	FALSE	<1
Uniform Random-3-SAT 18/unsat/uuf75-046.cnf	FALSE	<1

Listings

1. Recursive DPLL Algorithm 10
2. Iterative DPLL 12
3. Resolution 20
4. DIMCAS Format 40
5. Clause Counter-Based Implementation 42
6. Variable Counter-Based Implementation 43
7. MiniSAT Decision Heuristic Implementation 45
8. SetOfClauses Counter-Based Implementation 55
9. Forget Heuristic Implementation 56
10. Restart Heuristic Implementation 57

Bibliography

[AS08] Gilles Audemard and Laurent Simon. “Experimenting with Small Changes in Conflict-Driven Clause Learning Algorithms.” In: *CP*. Ed. by Peter J. Stuckey. Vol. 5202. Lecture Notes in Computer Science. Springer, Sept. 24, 2008, pp. 630–634. ISBN: 978-3-540-85957-4. URL: `http://dblp.uni-trier.de/db/conf/cp/cp2008.html#AudemardS08`.

[BS97] Roberto J. Bayardo Jr. and Robert C. Schrag. “Using CSP Look-back Techniques to Solve Real-world SAT Instances”. In: AAAI’97/IAAI’97. Providence, Rhode Island: AAAI Press, 1997, pp. 203–208. ISBN: 0-262-51095-2. URL: `http://dl.acm.org/citation.cfm?id=1867406.1867438`.

[CA93] James M. Crawford and Larry D. Auton. “Experimental Results on the Crossover Point in Satisfiability Problems”. In: *Proceedings of the Eleventh National Conference on Artificial Intelligence*. AAAI’93. Washington, D.C.: AAAI Press, 1993, pp. 21–27. ISBN: 0-262-51071-5. URL: `http://dl.acm.org/citation.cfm?id=1867270.1867274`.

[CA96] James M. Crawford and Larry D. Auton. “Experimental Results on the Crossover Point in Random 3sat”. In: *Artificial Intelligence* 81 (1996), pp. 31–57.

[Coo71] S. A. Cook. “The Complexity of Theorem-proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: `10.1145/800157.805047`. URL: `http://doi.acm.org/10.1145/800157.805047`.

[DLL62] M. Davis, G. Logemann, and D. Loveland. “A Machine Program for Theorem-proving”. In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: http://doi.acm.org/10.1145/368273.368557.

[DP60] M. Davis and H. Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: http://doi.acm.org/10.1145/321033.321034.

[EB05] Niklas Eén and Armin Biere. “Effective Preprocessing in SAT Through Variable and Clause Elimination”. In: SAT’05. Springer, 2005, pp. 61–75. ISBN: 3-540-26276-8, 978-3-540-26276-3. DOI: 10.1007/11499107_5. URL: http://dx.doi.org/10.1007/11499107_5.

[Har09] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. 1st. Cambridge University Press, 2009. ISBN: 0521899575, 9780521899574.

[KG07] Sava Krstić and Amit Goel. “Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL”. English. In: ed. by Boris Konev and Frank Wolter. Vol. 4720. Lecture Notes in Computer Science. Springer, 2007, pp. 1–27. ISBN: 978-3-540-74620-1. DOI: 10.1007/978-3-540-74621-8_1. URL: http://dx.doi.org/10.1007/978-3-540-74621-8_1.

[Lia+15] Jia Hui Liang et al. “Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers.” In: *Haifa Verification Conference*. Ed. by Nir Piterman. Vol. 9434. Lecture Notes in Computer Science. Springer, 2015, pp. 225–241. ISBN: 978-3-319-26286-4. URL: http://dblp.uni-trier.de/db/conf/hvc/hvc2015.html#LiangGZZC15.

[LMS01] I. Lynce and J. Marques-Silva. *The Puzzling Role of Simplification in Propositional Satisfiability*. 2001.

[Mar] F. Marić. *Flexible Implementation of SAT solvers*. Tech. rep. Faculty of Mathematics, University of Belgrade.

[Mos+01] Matthew W. Moskewicz et al. “Chaff: Engineering an Efficient SAT Solver”. In: DAC ’01. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379017. URL: http://doi.acm.org/10.1145/378239.379017.

[MS95] Joao Marques-Silva. “Search Algorithms for Satisfiability Problems in Combinational Switching Circuits”. PhD thesis. University of Michigan, 1995. URL: http://eprints.soton.ac.uk/265010/.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)”. In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977. ISSN: 0004-5411. DOI: 10.1145/1217856.1217859. URL: http://doi.acm.org/10.1145/1217856.1217859.

[Pro93] Patrick Prosser. “Hybrid Algorithms for the Constraint Satisfaction Problem”. In: *Computational Intelligence* 9.3 (1993), pp. 268–299. ISSN: 1467-8640. DOI: 10.1111/j.1467-8640.1993.tb00310.x. URL: http://dx.doi.org/10.1111/j.1467-8640.1993.tb00310.x.

[Rya02] Lawrence Ryan. “Efficient Algorithms for Clause-Learning SAT Solvers”. Simon Fraser University, 2002.

[Sar99] Stefan Saroiu. *An Overview of the MOMs Heuristics*. Report. University of Washington, 1999.

[Sch14] Albert Schimpf. *Modern SAT Solvers*. Technical Report. TU Kaiserslautern, 2014.

[Sil99] João P. Marques Silva. “The Impact of Branching Heuristics in Propositional Satisfiability Algorithms”. In: *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*. EPIA ’99. London, UK, UK: Springer-Verlag, 1999, pp. 62–74. ISBN: 3-540-66548-X. URL: http://dl.acm.org/citation.cfm?id=645377.651196.

[SM12] G. Weissenbacher S. Malik. *Boolean Satisfiability Solvers: Techniques and Extensions*. 2012.

[SS96] J. P. Marques Silva and Karem A. Sakallah. “GRASP a New Search Algorithm for Satisfiability”. In: ICCAD ’96. San Jose, California, USA: IEEE Computer Society, 1996, pp. 220–227. ISBN: 0-8186-7597-7. URL: http://dl.acm.org/citation.cfm?id=244522.244560.

[Tse83] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*. Ed. by J. Siekmann and G. Wrightson. Berlin, Heidelberg: Springer, 1983, pp. 466–483.

[Zha+01] Lintao Zhang et al. “Efficient Conflict Driven Learning in a Boolean Satisfiability Solver”. In: *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*. ICCAD ’01. San Jose, California: IEEE Press, 2001, pp. 279–285. ISBN: 0-7803-7249-2. URL: http://dl.acm.org/citation.cfm?id=603095.603153.

[ZM02] Lintao Zhang and Sharad Malik. "The Quest for Efficient Boolean Satisfiability Solvers". In: CAV '02. Springer, 2002, pp. 17–36. ISBN: 3-540-43997-8. URL: `http://dl.acm.org/citation.cfm?id=647771.734434`.

[ZMG03] Lintao Zhang, Sharad Malik, and Aarti Gupta. "Searching for Truth: Techniques for Satisfiability of Boolean Formulas". PhD thesis. 2003.

[ZS96] H. Zhang and M. E. Stickel. *An Efficient Algorithm for Unit Propagation*. 1996.