Modern SAT Solvers

Albert Schimpf

Technische Universität Kaiserslautern
a_schimpf12@cs.uni-kl.de

Abstract

We first outline the history of backtrack-based SAT solving, which starts with the Davis-Putnam Procedure, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, to todays modern versions of DPLL. We introduce a rule-based description of DPLL capturing the original DPLL algorithm and improve our system along the way, introducing new techniques used in most modern SAT solvers. Each improvement — non-chronological backtracking, efficient data structures for Boolean constraint propagation (BCP), and decision heuristics — is explained separately, so that the reader understands how a modern SAT solver works in theory and could implement such with minimal effort.

1 Introduction

The Propositional Satisfiability Problem (SAT) is the problem of determining whether there exists a variable assignment that evaluates a given formula to true. Even though the problem was the first to be classified as being NP-complete [3], only recently research has shown significant advances in solving hard and very large real-world SAT instances. There is need for SAT solvers to test satisfiability of a certain instance — circuit testing, planning and super-scalar processor verification are only a small portion of a big application domain. Thus, the motivation to develop even better solvers exists until today.

It should be noted, that given a formula, estimating the hardness is challenging without solving it first. Some early attempts at describing difficulty can be found in [17] and [7]. Estimating hardness of a formula is still under research.

There are different approaches to SAT solving, depending on whether an instance is randomly generated or derived from a real-world problem. A stochastic search algorithm[1] may be able to solve a large random satisfiable instance of SAT, but can not be applied to unsatisfiable instances, because it is not complete. On the other hand, backtrack search algorithms, such as modern DPLL based solvers, are able to solve real-world instances, and given enough time, are able to proof that an instance is unsatisfiable. For problems derived from real-world domains proving unsatisfiability is often the challenge.

This paper starts with describing the well-known original Davis-Putnam-Logeman-Loveland procedure. From then on, a rule-based description with modern strategies and techniques is introduced, namely an iterative implementation of non-chronological backtracking, efficient data structures, and selection heuristics. The goal of this paper is to answer the question of how to implement an efficient SAT solver by todays standards.

Related Work. Since this is a summary of the current state of SAT solving, this paper uses research from the last 40 years of improvements on the DPLL algorithm. Important related works include the introduction of a high-level description of DPLL by [11], the modular architecture by [14], and fast heuristics, conflict-driven learning, and efficient data structures by [15]. Benchmarks of all introduced techniques can be found in [13]. For a more general overview of SAT solving techniques, we advise to read the survey [9].

[1] for a stochastic search algorithm example, see [4]

2 Preliminaries

Formulas, assignments and satisfaction from [16]. 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 $\neg l$ or $\bar{l}$, 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. If $M \models F$, then M is a *model* of F. If F has no models then it is *unsatisfiable*.

3 Original Davis-Putnam Procedure and DPLL

The Davis-Putnam Procedure was originally presented as a two-phase proof-procedure for first-order logic [5]. The second phase, the propositional phase, operated with three rules: unit, pure and resolution. Since the formula explodes in size if resolution is used, improvements had to be done to restrict the growth of the formula. As such, most of todays modern SAT solvers use a case split, building upon the ideas of the Davis-Putnam-Logemann-Loveland algorithm [4]. Instead of resolution, Shannon's expansion is used to design a depth-first search algorithm with backtracking. The first recursive algorithm along with basic deduction functions is shown in Appendix A.

Shannon's expansion: $F \equiv (x \wedge F[x/1]) \vee (\neg x \wedge F[x/0])$

DPLL operates on a CNF formula and a set of variable assignments. Before the actual split, there is a deduction phase with rules like pure and unit which deduces necessary assignments, pruning search space in advance. The function stops when it concluded that the formula is either satisfied or conflicting.

For efficiency reasons, the formula presented to the solver is a CNF formula. 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*.

The recursive DPLL algorithm is simple and not very efficient: it solves only a fraction of given instances with high time consumption. In fact, all solvers based on chronological backtracking appear not to be suited for solving hard real-world problems [13]. Therefore, new techniques need to be introduced.

Rule-Based Abstraction of DPLL

Before introducing modern techniques, it is necessary to describe the SAT solving process in an unified way, namely as a transition system with states and transition rules. With this transistion system we can bridge the gap between theory and actual implementation and reason formally about completeness, correctness, and termination. We first start with basic definitions of the system, introduce basic rules to describe DPLL in its original form,

and add rules after introducing new techniques. The complete system can then be found in Appendix B.
There are currently two ways to describe a DPLL system, both being complete DPLL transition systems accompanied by correctness and termination proofs [16] [11]. Since [14] provides a modular object-oriented architecture for [11] with minimal overhead, we will concentrate on that system. In the following section we rely on definitions from [11].

DPLL State Transition System. We need to give a precise description of the solving process. Since DPLL uses Shannon's expansion recursively, each time the program calls itself, a new stack is created. We avoid that by adding checkpoint symbols $\diamondsuit$ to the current truth assignment. These checkpoints implicitly manage the decision levels of the literals; every checkpoint increases the decision levels of following literals by one. In addition to that, they can be visited to flip the decision of the check-pointed literal (removing the checkpoint in the process), imitating the recursive stack managed by the program. This allows for iterative implementation and avoids unnecessary large stacks. In summary, we want to track the formula and the current assignment with checkpoints, decision levels, and order of the assigned literals. We also need a way to store conflicts separately from the formula. With this information, we can define the DPLL *state*. An example run can be found in fig. 1.

▶ **Definition 1** (DPLL State). Let P be a finite set of propositional literals. A DPLL *state* is a triple $\langle F, M, C\rangle$, where: F is a set of clauses over P, M is a *check−pointed sequence*, or check-pointed trail, of literals in P (an element is either a literal or the checkpoint symbol $\diamondsuit$) and C is either a subset of P or the symbol no_cflct. The relation $l \prec l'$ means l occurs before l' in M. The current decision level literals without checkpoint symbols at level m have the form $M^{\langle m\rangle}$. M can be uniquely written as $M = M^{\langle 0\rangle} + \diamondsuit + M^{\langle 1\rangle} + \diamondsuit + ... + \diamondsuit + M^{\langle d\rangle}$. Symbol "+" denotes concatenation and $\diamondsuit$ does not occur in any $M^{\langle i\rangle}$. When using "[]" as brackets in the superscript, the prefix of M up to decision level m is meant: $M^{[m]} = M^{\langle 0\rangle} + \diamondsuit + ... + ... \diamondsuit + M^{\langle m\rangle}$; this is an unique representation of M. We write level $l = i$ if l occurs in $M^{\langle i\rangle}$, with i being the decision level of the literal. The literal d^m after a checkpoint $\diamondsuit$ is called the *decision literal* of decision level m. We speak of the decision literal d of the current decision level without the superscript. To change a state, a rule needs to be applied. If the guard of a rule is satisfied, it can be applied to change the DPLL state.

We can now define rules to describe a basic DPLL state transition system. The algorithm in Appendix A has explicit unit propagation (deduction function), explicit conflict detection (functions isSatisfied() and isConflicting()), an explicit decision step (chosseFreeVariable function), and implicit backtracking (recursive DPLL calls). We need to describe the decision step, unit propagation, conflict detection, and backtracking for the basic algorithm explicitly. Pure and subsumption is not described by this system, as these deduction steps are not part of most modern solvers.

$$(Decide)\ \frac{l\in P \text{ and } l,\overline{l}\notin M}{M:=M+\diamondsuit+l} \qquad (UnitPropag)\ \frac{l\vee l_1\vee ...\vee l_k\in F \text{ and } \overline{l_1},...,\overline{l_k}\in M \text{ and } l,\overline{l}\notin M}{M:=M+l}$$

$$(Conflict)\ \frac{C=no\_cflct \text{ and } \overline{l_1}\vee ...\vee \overline{l_k}\in F \text{ and } l_1,...,l_k\in M}{C:=\{l_1,...,l_k\}}$$

$$(Backtrack)\ \frac{C=\{l_1,...,l_k\} \text{ and } \diamondsuit\in M}{C:=no\_cflct \text{ and } M:=M^{[level\ d-1]}+\overline{d}}$$

Decide: Assign a currently unassigned literal in the trail M with a checkpoint. If the trail leads to an unsatisfiable formula, one can backtrack to the checkpoint, remove it, and flip l to explore the other assignment of l.

UnitPropag: If there is a clause with only one literal l unassigned and all other literals are falsified in M, add the literal to the trail.

Conflict: If there is currently no conflict and a clause cl is conflicting in M, then set C as the set of conflicting literals of the clause cl.

Backtrack: If there is currently a conflict and at least one checkpoint, backtracking is possible. Backtrack all literals until a checkpoint is reached, remove that chekpoint, and add the negation of the previous decision level literal to the trail, which means both assignments of literal d have been searched.

The defined system with rules is in itself non-deterministic. Every rule can be applied when the guard is satisfied. Still, restricting the usage of rules to a certain order is needed to speed up the process and to describe a specific algorithm correctly. In the case of the classic DPLL algorithm, the *strategy* of the system is as follows:

$$(((\text{Conflict; Backtrack}) \;||\; \text{UnitPropag})^* \;;\; [\text{Decide}])^*$$

Symbol '||' denotes non-deterministic choice, 'a*' means "apply a as long as possible", and '[a]' means "apply a once if possible".

An example run on $F_{init} = \{\overline{1} \vee 2, \overline{3} \vee 4, \overline{5} \vee 6, \overline{2} \vee \overline{5} \vee \overline{6}\}$ would look like this:

$$\begin{array}{c} < F, [], no\_cflct > \\ < F, [\Diamond 12 \Diamond 3], no\_cflct > \\ < F, [\Diamond 12 \Diamond 34 \Diamond 56], no\_cflct > \\ < F, [\Diamond 12 \Diamond 34\overline{5}6], no\_cflct > \end{array} \xrightarrow[\xrightarrow{Conflict}]{\xrightarrow{Decide}\xrightarrow{UnitPropag}} \begin{array}{c} < F, [\Diamond 1], no\_cflct > \\ < F, [\Diamond 12 \Diamond 34], no\_cflct > \\ < F, [\Diamond 12 \Diamond 34 \Diamond 56], \{2, 5, 6\} > \end{array} \xrightarrow[\xrightarrow{Backtrack}]{\xrightarrow{Decide}\xrightarrow{UnitPropag}} \begin{array}{c} < F, [\Diamond 12], no\_cflct > \\ < F, [\Diamond 12 \Diamond 34 \Diamond 5], no\_cflct > \\ < F, [\Diamond 12 \Diamond 34\overline{5}], no\_cflct > \end{array} \xrightarrow[\xrightarrow{Decide}]{\xrightarrow{UnitPropag}\xrightarrow{Decide}}$$

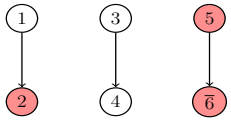
Fig. 1: A run of classic DPLL on a satisfiable instance F.

The main goal is introducing new techniques for SAT solvers and defining them as rules for this state transition system. While most improvements to a SAT solver will be low-level, high-level improvements can still be made. If a new technique requires a new rule, it will be introduced at the end of that chapter. All rules can be found in Appendix B.

4 Conflict-Driven Backjumping and Clause Learning

First introduced in [18] and [1], *conflict-directed backjumping* in combination with *conflict-directed learning* tries to find the reason for a conflict, i.e. which decisions are actually responsible for the conflict. The motivation for non-chronological backtracking can be illustrated in this example:

Example 1. Consider the formula F = $(\overline{1} \vee 2) \wedge (\overline{3} \vee 4) \wedge (\overline{5} \vee \overline{6}) \wedge (6 \vee \overline{5} \vee \overline{2})$ with the trail M = $\{\Diamond 12 \Diamond 34 \Diamond 5\overline{6}\}$. F is conflicting in this state. Chronological backtracking assumes that every decision is responsible for the conflict and flips the last decision literal 5. Some decisions, however, do not contribute to the current conflict, which can not be seen when only presented as a decision trail. To filter out the relevant decision levels, an *implication graph* has to be constructed. The nodes of this graph represent the literals in the trail and the edges represent unit propagations from clauses, which causes the unit literal to become unit (the *antecedent* clause).



Example 1: Implication graph

In this trail, decisions 1 and 5 are incompatible, as both of them lead the last clause to a conflicting state. From this graph, we can easily see that decision 3 with it's unit implication 4 does not affect the conflict. We could ignore this assignments and the conflict would still occur. The actual reason of the conflict, conditions $\overline{1} \vee \overline{5}$ and $\overline{2} \vee \overline{5}$, are called *lemmas*, because both can be used to prevent the assignments that cause this conflict. We have to at least revert the last decision level (here the decision 5) and prevent an assignment that would not satisfy the lemma. We select a literal from the lemma, that is currently false in M and backjump as much as possible, skipping irrelevant decision levels until a lemma becomes unit again. Ignoring the decision level of literal 3 and unit implication 4 has the following effect: the system with trail M = $\{\Diamond 12\overline{5}\}$ is more *advanced* than the system with trail M = $\{\Diamond 12\Diamond 34\overline{5}\}$, i.e. more search space is pruned by jumping over irrelevant decision levels. That means if a unit implication is not dependent on some previous decision levels, one can always place these implications higher up the depth-first search tree. This prevents searching for the same assignments again in every sub-tree of the not dependent decision levels.

The main goals of backjumping can now be stated as follows: (1) Find the reason for the conflict. (2) Force unit propagation of the lemma as early as possible. Additionally, the lemma is learned to prevent similar conflicts in the future. This is especially useful if restarts are used.

In practice, the whole implication graph is not needed. We can work backwards starting at the end at the trail until a certain condition is met. The condition is finding a *Unique Implication Point* (UIP). This guarantees that the clause becomes unit, if the jump is not too large.

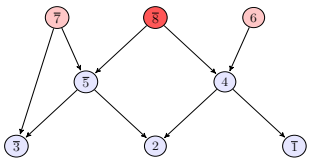
▶ **Definition 2.** *Unique Implication Point from [16].* Let D be the set of all the literals of a conflict clause C that have become false at the current decision level. A UIP in the (partial) implication graph of C is any literal that belongs to all paths in the graph from the current decision literal to the negation of a literal in D.

By definition, the decision literal itself is a UIP. In the previous example, there is only one UIP, the decision literal 5. Thus, the graph would only consist of the nodes 2,5 and $\overline{6}$, because we would stop the process after reaching literal 5 in the trail. Constructing the implication graph backwards and obtaining a lemma is illustrated in the next example.

▶ **Example 3.** Consider a state $< F, M, \{\overline{1}, 2, \overline{3}\} >$ where $F = \{\overline{9} \vee \overline{6} \vee 7 \vee \overline{8}, 8 \vee 7 \vee \overline{5}, \overline{6} \vee 8 \vee 4, \overline{4} \vee \overline{1}, \overline{4} \vee 5 \vee 2, 5 \vee 7 \vee \overline{3}, 1 \vee \overline{2} \vee 3\} \cup F_{other}$ [2] and $M = \{...6...\overline{7}...\Diamond 9\overline{8}5\overline{4}\overline{1}2\overline{3}\}$.
With this information the following implication graph can be built. A practical strategy is stopping at the *first* UIP, as [18] pointed out.

Steps for constructing this implication graph:

1. Start by drawing the conflict literals, $\overline{3}$,2, and $\overline{1}$.
2. Get clauses that unit propagated the literals $\overline{3}$, 2, and $\overline{1}$: $\overline{3}$ is implied by $\{5 \vee 7 \vee \overline{3}\}$, 2 by $\{\overline{4} \vee 5 \vee 2\}$, and $\overline{1}$ by $\{\overline{4} \vee \overline{1}\}$.
3. Apply step 2 for literals 4, $\overline{5}$.



Example 3: Partial implication graph

The backjump clause $\{7 \vee 8 \vee \overline{6}\}$ can be found by selecting all nodes with no incoming arrows. We know $\overline{8}$ is the first UIP, because of its decision level (the same as the last decision literal)

[2] F_{other} is irrelevant in this example.

and all paths from $\overline{8}$ lead to all conflict literals. A common strategy for picking the decision level is jumping to the *second most recent* decision level of the backjump clause (in this example, backjump to the decision level of 7). This skips all irrelevant decision level, while still guaranteeing that the backjump clause becomes unit after the jump. Thus, this clause prevents the same conflict again, ensuring that the combination $\{\overline{7}, \overline{8}, 6\}$ that led to the conflict is not encountered again.

Resolution. Before we extend our DPLL system, we need a more practical way of obtaining a backjump clause. The backwards process can be described as stepwise resolution until only one literal of the current decision level is in the resolution clause (the first UIP scheme). We start with the conflict clause and work backwards in the assertion trail. Since every literal after the decision literal is implied by an unit clause, we can resolve the current conflict clause with the unit clause of the literal we are currently looking at in the trail. In the worst case, we stop at the decision literal, in that case the only UIP. Revisiting the previous example, instead of constructing the conflict graph, we apply the backwards resolution process until the first UIP is found. [3]

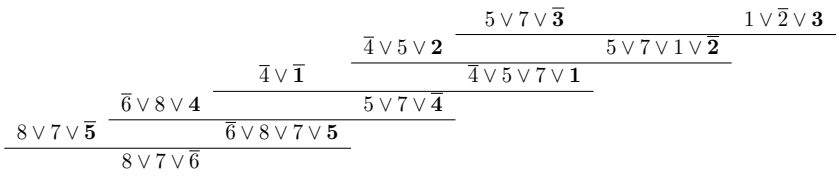


Fig. 2: Stepwise resolution from the conflict clause until only one literal is in the current decision level.

So why is learning a backjump clause so important? An example from [10] shows that backjumping to the last decision level is often not the best strategy.

Therefore, consider the clauses $C_1 \equiv (\overline{x_1} x_n x_{n+1})$ and $C_2 \equiv (\overline{x_1 x_n x_{n+1}})$ 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 constructing an implication graph (or by backwards resolution), we can determine that x_1 and x_n caused a conflict for either C_1 or C_2, so adding the conflict clause $(\overline{x_1} \vee \overline{x_n})$ eliminates this combination, and backjumping to the correct decision level prunes a large fraction of the search space.

Analysis of chronological and non-chronological backtracking shows that non-chronological backtracking allows for solving hard instances in the first place, as tested in [13].

4.1 Backjumping in the State Transition System

We need to describe the resolution step and the learning step as rules in the system. Furthermore, the backtrack rule needs to be modified to allow backjumping to lower decision levels.

[3] It should be noted, that it is possible that a literal can neither be a decision literal nor an unit literal with a reason, if chronological backtracking is used. The backtracked literal in that case has no reason, and is not an unit implication. These literals are skipped in the resolution process, but can cause the resolution process to terminate, leaving more than one literal of the current decision level in the conflict set.

$$(Explain)\ \frac{l\in C \text{ and } l\vee\overline{l_1}\vee\ldots\vee\overline{l_k}\in F \text{ and } l_1,\ldots,l_k\prec l \text{ and } \forall l'\in C: l'\preceq l \text{ and } \exists l'\in C: level\ l'=level\ l, l'\neq l}{C:=C\cup\{l_1,\ldots,l_k\}\setminus\{l\}}$$

$$(Learn)\ \frac{C=\{l_1,\ldots,l_k\} \text{ and } \overline{l_1}\vee\ldots\vee\overline{l_k}\notin F}{F:=F\cup\{\overline{l_1}\vee\ldots\vee\overline{l_k}\}}$$

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

$$\begin{array}{c|c|c|c|c|c}
< F, [], no_cflct > & \xrightarrow{Unit} & < F, [1], no_cflct > & \xrightarrow{Decide} & < F, [1\Diamond 2], no_cflct > & \xrightarrow{Unit} \\
< F, [1\Diamond 2\overline{3}], no_cflct > & \xrightarrow{Conflict} & < F, [1\Diamond 2\overline{3}], \{1,2,\overline{3}\} > & \xrightarrow{Explain} & < F, [1\Diamond 2\overline{3}], \{1,2\} > & \xrightarrow{Learn} \\
< F', [1\Diamond 2\overline{3}], \{1,2\} > & \xrightarrow{Backjump} & < F', [1\overline{2}], no_conflict] > & \xrightarrow{Unit} & < F', [1\overline{23}], no_conflict] > & \xrightarrow{Conflict} \\
< F', [1\overline{23}], \{1\overline{23}\} > & \xrightarrow{Explain} & < F', [1\overline{23}], \{1\overline{2}\} > & \xrightarrow{Explain} & < F', [1\overline{23}], \{1\} > & \xrightarrow{Explain} \\
< F', [1\overline{23}], \emptyset > & & & & & \\
\end{array}$$

Fig. 3: A run of DPLL on an unsatisfiable instance F.
$F_{init} = \{1\vee\overline{2}, \overline{1}\vee\overline{2}\vee\overline{3}, \overline{1}\vee\overline{2}\vee 3, 1, \overline{3}\vee 2, \overline{1}\vee 2\vee 3\}$
$F' = F\cup\{\overline{1}\vee\overline{2}\}$

Explain[4]**:** Condition one and three ensure the reverse order of traversing the assertion trail. The last two conditions guarantee that this rule can not be applied when only one literal in the current decision level is left (the *first* UIP). Clause $l\vee\overline{l_1}\vee\ldots\vee\overline{l_k}\in F$ is the unit clause that caused l to become unit, this reference has to be saved during the solving process for efficient access. The literals in this clause can be resolved with the current saved conflict literals C. This describes one resolution step.

Learn: Simple rule to add a conflict clause to the formula. Since literals in C describe a conflicting set of literals, one can always add the negated disjunction of these literals to the formula without changing the properties of the formula.

Backjump: As opposed to the backtrack guard, the backjump guard enforces that there is one unique highest decision level literal (the UIP l) among the conflict literals. That is not always the case, this is why backjump can not function without the explain rule. Another requirement is that the backjump clause $\bar{l}\vee\overline{l_1}\vee\ldots\vee\overline{l_k}$ has to be in the formula. This clause *has* to become unit after backjumping, otherwise it is not guaranteed that exhaustive resolution can be applied until only one literal is left in the current conflict literal set. We can pick the level freely between the current level and the highest level of the other conflict literals. Since we want to enforce unit propagation as early as possible, m is picked as the minimum of that range, i.e. the largest *level* l_i is chosen as m.

$$\begin{array}{c|c|c|c|c|c}
< F, [], no_cflct > & \xrightarrow{Decide} & < F, [\Diamond 1], no_cflct > & \xrightarrow{Unit} & < F, [\Diamond 12], no_cflct > & \xrightarrow{Decide} \\
< F, [\Diamond 12\Diamond 3], no_cflct > & \xrightarrow{Unit} & < F, [\Diamond 12\Diamond 34], no_cflct > & \xrightarrow{Decide} & < F, [\Diamond 12\Diamond 34\Diamond 5], no_cflct > & \xrightarrow{Unit} \\
< F, [\Diamond 12\Diamond 34\Diamond 56], no_cflct > & \xrightarrow{Conflict} & < F, [\Diamond 12\Diamond 34\Diamond 56], \{2,5,6\} > & \xrightarrow{Explain} & < F, [\Diamond 12\Diamond 34\Diamond 56], \{2,5\} > & \xrightarrow{learn} \\
< F', [\Diamond 12\Diamond 34\Diamond 56], \{2,5\} > & \xrightarrow{Backjump} & < F', [\Diamond 12\overline{5}], no_cflct > & \xrightarrow{Decide} & < F', [\Diamond 12\overline{5}\Diamond 3], no_cflct > & \xrightarrow{Unit} \\
< F', [\Diamond 12\overline{5}\Diamond 34], no_cflct > & \xrightarrow{Decide} & < F', [\Diamond 12\overline{5}\Diamond 34\Diamond\overline{6}], no_cflct > & & & \\
\end{array}$$

Fig. 4: A run of DPLL on an satisfiable instance F taken from [11]. Notice the decision level jump from 3 to 1.
$F_{init} = \{\overline{1}\vee 2, \overline{3}\vee 4, \overline{5}\vee 6, \overline{2}\vee\overline{5}\vee\overline{6}\}$
$F' = F\cup\{\overline{2}\vee\overline{5}\}$

The main goal of this system is to provide correctness and termination proofs, while guiding the implementation. Fig. 3 and fig. 4 are two examples of a DPLL run with these new rules. The correctness of the system is stated by the following theorem, proved in [11].

▶ **Theorem 4.** *(Correctness). 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: (a) $C = no\_cflct$ or $C = \emptyset$; (b) If $C = \emptyset$ then F_{init} is unsatisfiable; (c) If $C = no\_cflct$, then M is a model for F_{init}.*

5 Data Structures for Efficient BCP

As instances grow larger in size, the importance of efficiently storing and accessing clauses of the formula rises, too. If formula storage is not tailored to the solver's needs, big instances with millions of clauses become impossible to solve because of delays of inefficient clause structure.

There are two popular approaches to describe a set of clauses: the *counter-based* and the *lazy* approach. We will introduce one possible implementation of each approach: classical counter-based implementation and the *two-watched-literals* scheme.

5.1 Counter-Based Approach

Until the introduction of lazy implementations, the dominant approach to represent any set of clauses was counter-based. Here, clauses are represented as lists[5] of literals. Variables have a list of the clauses that contain the variable positively and negatively. These lists are not modified, instead counters (for active positive and negative literals) in clauses are used to keep track of the clause status. Every time, a variable is set to true, false or has it's assignment undone, all counters from clauses, that contain this variable have to be updated. If a variable satisfies the clause, the reference to that variable is saved. The 4 main operations[6] — isUnit(), isConflicting(), isSatisfied() and isUndefined() — can then be defined as follows:

1. $isUnit()$ iff active positive literals + active negative literals == 1
2. $isSatisfied()$ iff active != null
3. $isConflicting()$ iff active positive literals + active negative literals == 0
4. $isUndefined()$ iff !isSatisfied() && !isConflicting()

▶ **Example 5.** *Counter-Based Representation.* Consider the formula of the previous examples, F = $(\overline{1} \vee 2) \wedge (\overline{3} \vee 4) \wedge (\overline{5} \vee \overline{6}) \wedge (6 \vee \overline{5} \vee \overline{2})$ with the trail M = $\{\diamondsuit 12 \diamondsuit 34 \diamondsuit 5\}$. Both variables and clauses need to be stored. A counter-based variable has pointers to all clauses which include a literal of that variable. Clauses only maintain counters and a pointer to a variable if the clause is satisfied. Additionally, to find the unit literal, the clause stores all references to contained literals. The current sate is described in fig. 5, and the state after assignment in fig. 6.

[5] Experiments show that linked lists have a big disadvantage compared with the array data structure in terms of cache misses that translates to substantial slow-down in the solving process [19].

[6] In most implementations only $isUnit()$ and $isConflicting()$ is used.

Green edges signify that a positive literal of the variable is in the clause, red edges signify negative literals. Green clauses are already satisfied because of the backarrow (the variable reference which satisfies the clause in the current assignment). Dark blue clauses denote unit status, i.e. only one active literal is currently undefined. Light blue clauses denote that the clause is undefined and currently not unit. Counters are omitted. Once the status of a clause reaches unit, the actual unit literal has to be found first. By keeping all literals in a list, a clause can traverse this list and search for the only literal that is currently unassigned. This literal is the unit literal, in this case $\overline{6}$.

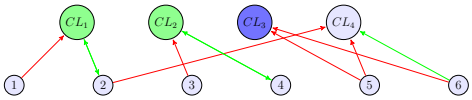


Fig. 5: Set before assignment.

After assigning $\overline{6}$, all references from 6 have to be updated: the variable pointer of CL_3 denotes that the clause is satisfied by 6 and CL_4 is conflicting because no active literals are available. Undoing an assignment of a variable works in a similar way.

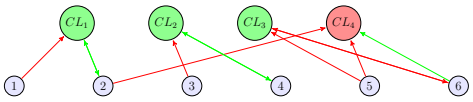


Fig. 6: Set after assignment.

This approach looks promising and easy to implement, but the actual effort to keep track of these counters is too much for modern SAT solvers. If the formula has m clauses and n variables, with each clause having an average of l literals, then assigning a value to a variable means $\frac{l*m}{n}$ counters have to be updated. The same is true for undoing an assignment. Because of clause learning and the fact that some learned clauses have many literals [19], this approach makes the engine relatively slow.

5.2 Lazy Approach - Two-Watched-Literals Scheme

Most DPLL engines operate with the unit propagation as the main focus, because most of the solving time is spent in this deduction phase [19]. The main idea of a lazy approach is to speed up the process of finding those unit clauses. As mentioned in the counter-based approach, each variable has a pointer to potentially many clauses. The idea is to keep the count of these pointers low and to speed up the assign and unassign operations. To determine if a clause is unit, i.e. has currently only one literal not assigned, we need to know the status of only *two* literals in the clause. As long as those two watched literals are not assigned, the clause can not be an unit clause (potentially losing information if the clause is already satisfied).

As in the counter-based approach, each clause stores it's literals in a list[7]. On top of that, each clause has two special literals, called *watched literals*. This idea is used so that a variable keeps only a list of watched positive literals and watched negative literals, for each of which the variable can effectively be used for declaring the clause as unit or unsatisfied. Initially, the watched literals are unassigned.

Assigning the value 1 to a variable p, the watched negative literals list is iterated and for each clause the literal is contained, a new literal l is searched which is not set to 0. There are four different cases that can happen:

- l exists: Set the previous watch to l and connect new variable to this clause. The reference from the old variable has to be removed.

[7] This may lead to many cache misses, because the memory footprint of a clause is too big. One possible optimization is to store this information separately to restrict the size of a clause as much as possible, effectively reducing cache misses.

- l exists, is the only literal and l is the watch and currently unassigned: clause is a unit clause, l being the unit literal.
- l exists, is the only literal and l is the watch and currently assigned: do nothing, because clause is satisfied.
- All literals in the clause are false in M and no such l exists, then the clause is conflicting.

Assigning the value 0 is processed accordingly.
Fig. 7 illustrates this process with a previous DPLL run, fig. [4]. We look at the last clause.

Action	Clause	status		Description
—	6	$\overline{5}$	$\overline{2}$	The watched literals (highlighted) are unassigned at the beginning and are freely chosen.
(*Decide* 1)	6	$\overline{5}$	$\overline{2}$	Clause will not be visited. Neither 6 nor $\overline{2}$ are being modified.
(*Unit* 2)	6	$\overline{5}$	$\overline{2}$	$\overline{2}$ is watched. Moving the watch to another unassigned literal literal.
(*Decide* 3)	6	$\overline{5}$	$\overline{2}$	Clause will not be visited.Neither 6 nor $\overline{5}$ are being modified.
(*Unit* 4)	6	$\overline{5}$	$\overline{2}$	Clause will not be visited. Neither 6 nor $\overline{5}$ are being modified.
(*Decide* 5)	6	$\overline{5}$	$\overline{2}$	One watched literal is modified. Clause can not find a free literal, and determines that the clause is now unit.
(*Unit* $\overline{6}$)	6	$\overline{5}$	$\overline{2}$	Clause is now conflicting because another unit clause implied $\overline{6}$. Backtrack.
Backjump to lvl 1	6	$\overline{5}$	$\overline{2}$	After backtracking no work has to be done. 6 and $\overline{5}$ are still being watched.
Backjump assigns $\overline{5}$	6	$\overline{5}$	$\overline{2}$	Assigning watched literal. Clause will not be visited.

Fig. 7: Two-Watched-Literal Scheme Example

There are many advantages to this approach: When assigning a value 1 to a variable, clauses that contain the literal positively will not be visited at all. This also holds with assigning value 0 and clauses containing the literal negative. Also, 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 variables assignment: In the two-watched-literals scheme *no* work has to be done when unassigning a variable. Again, the prediction of efficiency of lazy data structures is shown in [13].

6 Heuristics for the Decide Transition

When we apply the decide rule, the question remains of which literal to assign. While random choice or chronological choice has no impact on the termination of the system, heuristics can improve searching time of a solver significantly. There is generally a trade-off between *accurate* and *fast* heuristics, and practical implementations tend to apply fast heuristics.

Since the *Variable State Independent Sum* (VSIDS) heuristic is used in *Chaff*, we will describe this heuristics and skim over other lesser used ones.

VSIDS - a State Independent Heuristic. As solvers become more efficient and an increasing number implements a lazy data structure, the time of the function for calculating the next branch becomes more dominant in the run time. VSIDS tries to address this issue by being *state independent*: the heuristic is not dependent on the variable assignment.

The main goal of VSIDS is selecting literals that appear most frequently in all clauses, favoring literals in recently added learned clauses. Literals are initialized with a score of 0 and every time a clause is added, all literals in the clause are *boosted*, i.e. the score is increased. The scores are periodically *decaying*, i.e. they are all divided by a constant after a certain amount of time.

As an example, [15] implements VSIDS as follows: (1) Maintain a list of unassigned literals sorted by score and (2) update list when adding conflict clauses. Since the list is sorted, decision can be derived in $O(1)$.

Other Selection Heuristics. Other selection heuristics include Dynamic Largest Individual Sum, Jeroslow-Wang and Static Largest Individual Sum, but they are not used mostly because of high overhead or due to introduction of lazy data structures (and therefore lazy knowledge of clause status). For an overview of selection heuristics refer to [18]. A comparison between VSIDS and SLIS can be found in Appendix D.

7 Additional Strategies

There are many possibilities to speed up the solving process. We will describe three popular concepts used in most DPLL engines.

Preprocessing aims to simplify the instance to shorten the solving process. Because this is only applied one time, expensive checks can be implemented in the preprocess stage. Popular mechanics are pure rule, subsumption rule and unit rule until neither of these three are possible anymore. Interestingly enough, preprocessing does not always speed up the process and can even slow it down, as [12] shows. For a general picture of preprocessing techniques, refer to [6].

Restarts are used to increase robustness of a solver. By restarting at certain times, randomization can ensure that different sub-trees are searched each time the solver restarts. Since learned clauses are kept after a restart, similiar conflicts that ocurred prior to a restart are avoided. It also tries to reduce variance in similar instances of problems, which is a problem of almost all solvers. A small permutation in the initial formula could lead to big changes concerning solving time [2]. It is empirically validated that restarts can improve the results, and consequently most modern SAT solvers make use of this concept [13].

A restart strategy can be as simple as this, based on the number of occurred conflicts: Maintain a conflict counter. Each time a conflict occurs, increase the counter. If the counter exceeds a chosen threshold, tell the solver to restart. When restarting, increase the threshold. Restarts can be modeled in our state transition system, see Appendix C. Preservation of termination is not necessary in practice, but aggressive restarts should only be used with a appropriate clause learning/forgetting heuristic, or else the solver restarts without making much progress.

Clause Forgetting. Every time a backjump clause is learned, the formula increases in size. This is a problem, because learning is used very often, but some clauses are not very *useful* or have too many literals. One measure of usefulness is to measure the number of conflicts that involve this clause [8]. This measure works well in keeping the learning clause count low. Clause forgetting can also be modeled in our state transition system. See Appendix C.

8 Summary: A Modern SAT Solver

We can now describe a complete modern solver used in industry. The goal of the state transition system introduced in Chapter 3 is to describe all modern SAT solvers at a high-level view, while guiding the implementation. By using this state transition system, choosing the "two-watched-literals" scheme (Chapter 5), using the decision heuristic VSIDS (Chapter 6) and fine tuning the forget and restart rules, the system describes a top-level picture of the solver *Chaff* [15], a solver which emerged as the best complete solver of the SAT competition 2002 and 2004. As a reminder, we also need to restrict the usage of the rules in a certain order. The *Chaff* solver uses this strategy[8]:

$$(((\text{Conflict ; Explain* ; [Learn; Backjump]}) \;||\; \text{UnitPropag})^* \text{ ; [Decide]})^*$$

9 Conclusion

We surveyed techniques which are used in popular SAT solvers. We introduced a framework to describe the DPLL process at a high-level, and described the modern non-chronological backtracking technique with this system. We focused on the conflict-driven side of SAT solvers, and mentioned modern low-level techniques. With the help of the state transition system, it is possible to describe a large variety of solvers while helping to bridge the gap between theory and implementation. Furthermore, such a system enables implementation of smart low-level techniques. In summary, the setting given by our system enabled us to describe a modern SAT solver used in industrial software and hardware verification at an abstraction layer close to an actual implementation, while still giving freedom to implement efficient low-level techniques.

References

1 Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. AAAI'97/IAAI'97, pages 203–208. AAAI Press, 1997.

2 F. Brglez, Xiao Yu Li, and M. F. Stallman. The role of a skeptic agent in testing and benchmarking of SAT algorithms, 2002.

3 S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158. ACM, 1971.

4 M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

5 M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

6 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. SAT'05, pages 61–75. Springer, 2005.

7 Ian P. Gent and Toby Walsh. The SAT phase transition. pages 105–109. John Wiley and Sons, 1994.

8 Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Appl. Math.*, 155(12):1549–1561, June 2007.

9 Jun Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1996.

[8] Clause forgetting and restarts are not in this strategy, though they can be placed directly before the decide step.

10 J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 1st edition, 2009.

11 Sava Krstić and Amit Goel. Architecting solvers for SAT modulo theories: Nelson-oppen with DPLL. volume 4720 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2007.

12 I. Lynce and J. Marques-Silva. The puzzling role of simplification in propositional satisfiability, 2001.

13 I. Lynce and J. Marques-Silva. Building state-of-the-art SAT solvers. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence (ECAI 2002), IOS*, pages 166–170. Press, 2002.

14 F. Marić. Flexible implementation of sat solvers. Technical report, Faculty of Mathematics, University of Belgrade.

15 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. DAC '01, pages 530–535. ACM, 2001.

16 Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.

17 Mukul R. Prasad, Philip Chong, and Kurt Keutzer. Why is ATPG easy?, 1999.

18 J. P. Marques Silva and Karem A. Sakallah. GRASP a new search algorithm for satisfiability. ICCAD '96, pages 220–227. IEEE Computer Society, 1996.

19 Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. CAV '02, pages 17–36. Springer, 2002.

A Recursive Davis-Putnam Procedure

The main idea behind DPLL is a recursive depth-first search based on Shannon's expansion. There are three commonly used deduction functions, which do not alter satisfiability of the formula if applied.

- **Pure rule.** If a variable v is exclusively contained in the formula as a literal l (or as a literal $\neg l$), evaluate v to *true* (or *false* respectively).
- **Unit rule.** If there exists a clause cl which contains only one literal l (or $\neg l$), evaluate the corresponding variable v to *true* (or *false* respectively) to satisfy cl.
- **Subsumption rule.** If there exists a clause cl_1, which is *subsumed* by another clause cl_2, remove cl_1 from the formula. A clause cl_1 is *subsumed* by another clause cl_2, if every literal in cl_2 is contained in cl_1.

Listing 1 Recursive DPLL Pseudo Code from [19]

```
DPLL(formula, assignment){
  necessary = deduction(formula, assignment);
  newAsgnmnt = union(necessary, assignment);
  if (isSatisfied(formula, newAsgnmnt)){
    return SATISFIABLE;
  }
  if (isConflicting(formula, newAsgnmnt)){
    return CONFLICT;
  }
  var = chooseFreeVariable(formula, newAsgnmnt);
  asgn1 = union(newAsgnmnt, assign(var, 1));
  if (DPLL(formula, asgn1)==SATISFIABLE){
    return SATISFIABLE;
  }else{
    asgn2 = union(newAsgnmnt, assign(var, 0));
    return DPLL(formula, asgn2);
  }
}
```

B Rules of the State Transition System

Decide	$\frac{l \in P \text{ and } l,\bar{l} \notin M}{M := M + \diamond + l}$
UnitPropag	$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \text{ and } \bar{l_1},\ldots,\bar{l_k} \in M \text{ and } l,\bar{l} \notin M}{M := M + l}$
Conflict	$\frac{C = no\_cflct \text{ and } \bar{l_1} \vee \ldots \vee \bar{l_k} \in F \text{ and } l_1,\ldots,l_k \in M}{C := \{l_1,\ldots,l_k\}}$
Explain	$\frac{l \in C \text{ and } l \vee \bar{l_1} \vee \ldots \vee \bar{l_k} \in F \text{ and } l_1,\ldots,l_k \prec l \text{ and } \forall l' \in C : l' \preceq l \text{ and } \exists l' \in C : level\ l' = level\ l, l' \neq l}{C := C \cup \{l_1,\ldots,l_k\} \setminus \{\bar{l}\}}$
Learn	$\frac{C = \{l_1,\ldots,l_k\} \text{ and } \bar{l_1} \vee \ldots \vee \bar{l_k} \notin F}{F := F \cup \{\bar{l_1} \vee \ldots \vee \bar{l_k}\}}$
Backjump	$\frac{C = \{l, l_1,\ldots,l_k\} \text{ and } \bar{l} \vee \bar{l_1} \vee \ldots \vee \bar{l_k} \in F \text{ and level } l > m \geq \text{level } l_i \text{ for (i = 1,..,k)}}{C := no\_cflct \text{ and } M := M^{[m]} + \bar{l}}$
Forget	$\frac{C = no\_cflct \text{ and } l_1 \vee \ldots \vee l_k \in F \text{ and } F \setminus \{l_1 \vee \ldots \vee l_k\} \models \{l_1 \vee \ldots \vee l_k\}}{F := F \setminus \{l_1 \vee \ldots \vee l_k\}}$
Restart	$\frac{C = no\_cflct}{M := M^{[0]}}$

Table 1: Rules of DPLL with additional Forget and Restart rules from [11]