Boolean Satisfiability
Lecture 6: Conflict-Driven Clause Learning
(to be continued)

Edward A. Hirsch*

May 12, 2026

Lecture 6

In this lecture we discuss the CDCL approach that forms the basis of most known state-of-the-art SAT solvers.

For this particular lecture, I strongly suggest to look either at the (animated) slides or at some of the other available animations: these lecture notes do not cover animated examples!

Contents

*Ariel University, http://edwardahirsch.github.io/edwardahirsch

1 Introduction

In the previous lecture we have already seen the DPLL approach to solving **SAT**. The **CDCL** (**Conflict-Driven Clause Learning**) approach can be viewed as a generalization of DPLL (though we speak about it in somewhat different terms).

Currently, CDCL is the most popular the most successful method in **SAT** solving. In was suggested in 1996–99. Since then, dozens of CDCL solvers appeared (to name a few: GRASP, zChaff, CaDiCaL, Glucose) and are now widely used by the industry. They are also used by CS theorists and mathematicians for checking combinatorial conjectures experimentally.

We have seen previously that the basic DPLL algorithm (splitting over a variable + unit clause elimination + pure literal elimination) is equivalent to treelike resolution proofs. The CDCL algorithm (with restarts) is known to be equivalent to daglike resolution proofs (which are sometimes provably exponentially shorter than treelike proofs), so it has a greater potential. We will not prove this fact though.

The MiniSAT project (`http://minisat.se/` , by Niklas Eén and Niklas Sïensson) provides an opportunity to write a CDCL **SAT** solver without doing everything from scratch: it is a well-documented open-source platform that everyone can take as the start of their project (for example, a master thesis!).

2 Another look at DPLL

Before we proceed to CDCL, let us look at the run of a DPLL-type algorithm a little bit differently.

At least for the DPLL-type algorithm that we have seen before, at each node we choose a variable and consider the two possible values for it (even if we split over a clause, we can consider it as a sequence of substitutions to variables, for example, for $(x \vee y \vee z)$ we consider $x = 1$ vs $x = 0$, in the latter case we consider $y = 1$ vs $y = 0$, and in the last case the value of z is determined from the unit clause (z), so we have the same three substitutions $[x \leftarrow 1]$, $[x \leftarrow 0, y \leftarrow 1]$, $[x \leftarrow 0, y \leftarrow 0, z \leftarrow 1]$ that we have seen before).

Think now of choosing a value for a variable x as a **decision** (recall decision trees!). How does it happen that we come to a leaf marked by `False`, and have to backtrack and examine other assignments? It means that our decisions were wrong!

How do we understand it in practice? At the very end of the branch we substitute a value for a variable (either because of our decision or because of the unit clause elimination), and – alas – some clause becomes falsified. Surely, this clause was a unit clause before that. Since we would not make such a stupid decision ourselves, it means that we actually obtained two contradictory unit clauses: z, $\overline{z}$ and substituted that value got from one of them into the other one. So the values of some variables (on which we did not decide!) were forced (recall the PPZ algorithm!), perhaps through a cascade of unit clause eliminations, and eventually these forced values become contradictory, this is what we call a **conflict**.

The DPLL's way to resolve a conflict is to overturn the last decision and consider the other value for the corresponding variable, and so on – returning from the recursion. The CDCL's way to deal with conflicts is different.

3 The basic CDCL engine

3.1 The intuition

The unit clauses involved in the conflict may appear later in other branches, when after making different decisions on the same variables we arrive to a similar formula (part) and thus behave similarly to what we did before. We want to avoid the unnecessary search.

So let us attempt to learn this information as a new clause that is logically implied by the input formula (irrespective of the decisions made) and that will prevent us from doing the same search again (it will turn into a unit clause earlier and will guide us away from the useless subspace).

Also if the wrong decisions that we made happened at higher levels of what would be the DPLL search tree, we backtrack higher than DPLL would do.

3.2 Overall organization

Contrary to DPLL, a CDCL-type algorithm does not use the recursion explicitly, namely, it never considers the second value of the decision variable: it happens automatically when needed. When the CDCL algorithm cancels its decisions, it decides how many decisions it cancels (this is called **non-chronological backtracking** in constrast to DPLL's chronological backtracking), and the direction of the further search is determined from the learned clauses.

Contrary to DPLL, a typical CDCL solver does not modify the formula clauses. It can add (learn) new clauses, and it keeps the track of the current assignment. However, it does not substitute the values physically to the clauses (that is, it does not remove or shorten the clauses, it only marks some literals in them temporarily true or false), and it does not modify them in any other way. (There are experiments that use more compicated techniques, we are describing the "mainstream" here.)

3.3 Terminology

Let us start with introducing the terminology used in the field of CDCL SAT solving.

- Unit clause elimination is usually called **unit propagation**. (Sometimes **BCP**, for Boolean Constraint Propagation).
- We will be speaking about the **level** of decision (prepended by `@` in writing: `@`0, `@`1, `@`2, ...).

  The new level is started by making a new decision for a new variable. However, other variables that get value after that (typically, by unit propagation) are also thought of as assigned at the same level.

  We write: $x_5 = 1$`@`5, meaning that x_5 received value `True` at decision level 5.
- **Implied** variables are those forced by unit propagation.
- The **antecedent** (or the **reason**) of a literal ℓ is the clause of the original formula that turned into the unit clause (ℓ). (Recall that CDCL does not remove or change clauses, so technically this is indeed a clause where all literals but one have been set to `False`.)

- Assigned variables determine the **implication graph** (or **conflict graph**):
  - **Vertices:** assigned variables (or true literals) and `False` (the sink, if present).
  - **Edges:** generated (and marked) by clauses that imply literals through unit propagation:

    for example, if y and z were assigned `False` before x received its value `True`, then the clause $C = (x \vee y \vee z)$ generates $\overline{y} \xrightarrow{C} x$ and $\overline{z} \xrightarrow{C} x$.
- A solver can be called on formula F with "**assumptions**" $\{\ell_1, \ldots, \ell_a\}$; it solves **SAT**$(F \wedge \ell_1 \wedge \ldots \wedge \ell_a)$, but learned clauses do not contain the assumptions and can be further used in calls (of this solver or another one!) with different assumptions.

3.4 The basic CDCL algorithm

We describe the basic algorithm first and then continue to describing its most important procedure of **learning** the clauses, called also the **conflict analysis** procedure, (and determining their **asserting level**).

Algorithm **CDCL** (CNF F):

```
Level := 0
A := empty assignment

Repeat forever:
   Apply unit propagation to F (extending A)
   If F[A] = True ——————————— // found sat. assignment, success!
     Then return A
   Elif F[A] = False ——————————————————————— // conflict!
     Then
       If Level = 0 ——————————————— // unable to backtrack, done!
         Then return "no"
         Else ———————————————————————— // can backtrack
           Learn new clauses {C_i}_i and let F := F ∧ ⋀_i C_i
           Let L_* be the minimal asserting level among C_i's
           Unassign all variables set @levels > L_* ———— // backtrack to L_*!
           Level := L_*
   Else ———————————————————————— // no conflicts yet, new decision
     Pick an unassigned variable x and value v
     Let A := A ∪ {x := v}
     Level := Level + 1
```

3.5 Conflict graph analysis

The goal of the conflict analysis is to learn new clauses and to decide where to go afterwards (that is, how far to backtrack). Nowadays, most solvers typically learn a single clause per conflict and use **1UIP** ("the first unique implication point") strategy. They learn a clause that contains exactly

one literal that obtained its value @current level, and they try to learn this clause as close to the conflict as possible.

The conflict analysis procedure starts at the conflict (falsified) node of the conflict graph and searched for the causes of this conflict by going through the edges in the opposite direction and deriving new clauses by resolution.

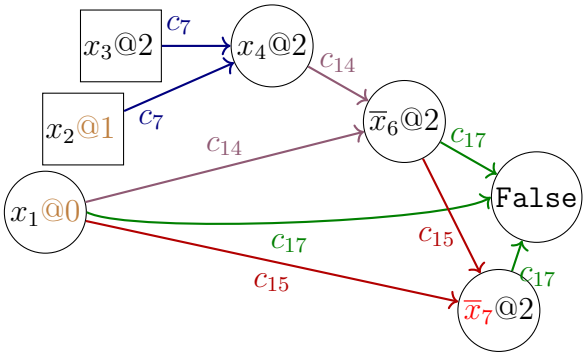
After it stops, if we cut the non-visited nodes out of it, we obtain a graph such that its source nodes themselves imply the conflict. Of course, this is also true for the whole graph, but we are aiming at something more efficient.

On the right of the pseudocode we include the data for the short example provided below the procedure.

Conflict-analysis():
C := antecedent of the `False` node — c_{17}
`Repeat`
  `Let` p `be last assigned literal in` C — $\overline{x}_7$
  D := antecedent of p — c_{15}
  $C := \mathcal{R}(C, D)$ `by` p — $\mathcal{R}(c_{15}, c_{17}) = (x_6 \vee \overline{x}_1)$
`Until` C `contains just one literal` ℓ `@Level`
`Learn` C`, state` **asserting level** `:= highest decision level in` $C \setminus \{\ell\}$ — 0

As the result of the conflict analysis, we will learn a clause that will contain just one literal @current level.

Example 1. A conflict graph for the example from [FLHS] demonstrated during the lecture:



c_7 : $(\overline{x}_2 \vee \overline{x}_3 \vee x_4)$
c_{14} : $(\overline{x}_6 \vee \overline{x}_4 \vee \overline{x}_1)$
c_{15} : $(x_6 \vee \overline{x}_7 \vee \overline{x}_1)$
c_{17} : $(x_7 \vee x_6 \vee \overline{x}_1)$

Formal definitions. Consider a conflict graph. A **UIP** (**unique implication point**) is a node present on all paths from the [latest unforced] decision node to the conflict node (`False`) (but distinct from `False`).

The **1UIP** (**the first unique implication point**) is the UIP that is the closest one to the conflict node (but still distinct from it).

Exercise 1 (HW exercise). Prove that 1UIP is indeed absolutely unique, that is, there exists a single UIP in a conflict graph.

3.6 Data structure: Watched literals

A typical CDCL solver spends almost all its time in the unit propagation procedure. Therefore, we need an efficient data structure able to find the unit clauses and to substitute values for the variables.

During the run of the algorithm the formula remains intact, only new clauses are appended to it. So how do we understand that a clause became a unit clause under the current partial assignment?

Let us classify clauses as:

1. Satisfied: there is a literal assigned `True`.
2. Unit: all literals are assigned (`False`, what else?) except one.
3. Passive, or unresolved: at least two literals are unassigned.

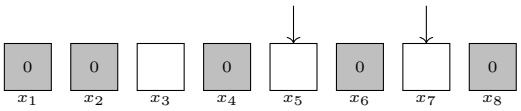
Only the case "2. Unit" is interesting!

We "watch" for (maintain pointers to) two literals in each yet-unsatisfied clause. If one of them is assigned `False`, we may have a unit clause. If both are assigned, there is a conflict. If none are assigned, there is nothing interesting yet.

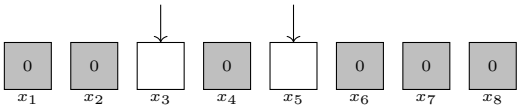
When a new variable gets its value, we look at the yet-unsatisfied clauses where it is watched. In such a clause, we move the pointer from the just assigned `False` literal in a cycle until we find an unassigned variable (not counting the second pointer). If it is found, we move the pointer there; otherwise, we have a unit clause.

When backtracking, we do not need to do anything to this structure: the pointers remain where they were (think why it is correct!).

Here is an example:



Now, the effect of $x_7 := 0$:



4 Components of CDCL solvers

We have already learned that CDCL solvers include

- Conflict analysis: what exactly caused the conflict? Which clauses do we learn from it?
- Non-chronological backtracking: backtrack higher than in DPLL.
- "Lazy" data structures.

Another important feature of CDCL solvers is that they **restart** the whole thing at certain intervals following some restart sequence (strategy). A restart can be hard (just start from the scratch with a different sequence of decisions) or soft (then, for example, the learned clauses are kept). In fact, they are equivalent to the resolution proof system only if they use restarts.

The amount of learned clauses may become excessive (it is proportional to the time spent). Therefore, strategies for deleting excessive learned clauses may be used.

Learned clause can also undergo **minimization** (for example, using resolution that derives a subset of a learned clause).

Also various types of heuristics may be used for the choice of the decision literal.

All these components of a solver are of highly experimental nature, and this is what you needs to work on if you want to design a new potentially ground-breaking solver.

Historical notes and further reading

Further reading: there is an easy expository article [FLHS]; if you want to learn more about CDCL, you can look into Chapters **7** and **4** (I recommend **this particular order!**) in the 2nd edition of Handbook of Satisfiability [HB7, HB4]. See Section 2.2 in another book [KS] for another introduction to the subject.

You can also look at very nice animations in the slides by Emina Torlak at `https://courses.cs.washington.edu/courses/cse507/17wi/lectures/L02.pdf` (See `web.archive.org` if the file is not accessible.)

References

[HB4] Joao Marques-Silva, Ines Lynce, and Sharad Malik.
Conflict-Driven Clause Learning SAT Solvers,
Chapter 4 in *Handbook of Satisfiability*, 2nd Ed., IOS Press, 2021.

[HB7] Samuel R. Buss and Jakob Nordström.
Conflict-Driven Clause Learning SAT Solvers,
Chapter 7 in *Handbook of Satisfiability*, 2nd Ed., IOS Press, 2021.

[FLHS] Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider.
The Silent (R)evolution of SAT,
Communications of the ACM 66(6): 64–72, 2023.
`https://doi.org/10.1145/3560469`

[KS] D. Kroening, O. Strichman. *SAT Solvers.* Section **2.2** in *Decision Procedures: An Algorithmic Point of View.* 2nd Ed., Springer, 2016.