Conflict Driven Clause Learning

Alexander T. Schlenga

June 8, 2020

Abstract

I present the CDCL algorithm and its implementation based on existing literature. This algorithm is used to solve SAT problems efficiently. First, the basics of SAT solving are introduced, then the CDCL algorithm's functionality is explained and a modern implementation is presented.

1 The boolean satisfiability problem

In order to reason about the satisfiability problem we first introduce some notions and notations. We use the *boolean notation* to denote the values of variables, i.e. $x_i = 0$ if x_i is assigned 0 or *false* and $x_i = 1$ if x_i is assigned 1 or *true*. We extend this notation to propositional formulae. An *assignment* now assigns to each variable occurring in a formula, and therefore also to the formula itself, a boolean value 0 or 1. The boolean satisfiability problem, often called SAT, is the following: given a formula F on propositional variables, does there exist an assignment $\mathcal{A}$ on these variables, such that $\mathcal{A}(F) = 1$.

It is an NP-complete problem. In recent times, for many other problems in NP, the approach was to reduce them to SAT. This is because nowadays there exist efficient SAT solvers for formulae of even millions of variables (Biere et al., 2009, p. 131)(Knuth, 2015, p. 62). Most of these modern SAT solvers are based on an algorithm called Conflict Driven Clause Learning (CDCL) which is presented here.

Before applying CDCL (or similar algorithms) to a boolean formula, it has to be transformed into Conjunctive Normal Form (CNF). A formula F is in CNF if and only if it is a conjunction of disjunctions of literals, i.e. of the form

$$F = \bigwedge_i \bigvee_j l_{i,j}$$

where $l_{i,j}$ are the literals. A *literal* is either a propositional variable or its negation. Usually, CNF formulae are represented in an alternative form called clauses using the commutativity, associativity and idempotence of the logical operators $\vee$ and $\wedge$. A clause corresponds to a disjunction of literals and a CNF formula is a conjunction of clauses. We write a clause as a set of its literals and a formula as a set of clauses. For example:

$$\text{A propositional formula:} \qquad \neg(((x_0 \wedge \neg x_1) \vee x_2) \leftrightarrow x_1)$$

$$\text{An equivalent formula in CNF:} \qquad (x_0 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

In clause representation: $\{\{x_0, x_1, x_2\}, \{\neg x_1, \neg x_2\}\}$

In reality, formulae are often not being transformed into equivalent CNF formulae but into equisatisfiable CNF formulae (formulae which are satisfiable if and only if the original formula is satisfiable) when preparing them for SAT, because this can be done efficiently.

2 Basics of SAT solving

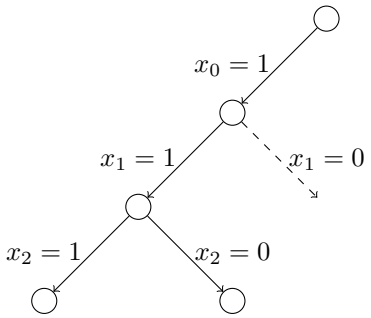
The CDCL algorithm is a mix of two older approaches to SAT solving: DPLL and Resolution. It uses concepts from both and combines them in a new way. So let us have a look at them first.

2.1 Backtracking and unit propagation as in DPLL solvers

When we provide a SAT instance to a DPLL solver, it builds up a search tree of assignments until it either finds a satisfying assignment or reports unsatisfiability because it has gone through the whole search tree.

This is done by iteratively choosing a variable x_i and, depending on some heuristic, setting its value to true or false and appending it to the track. If then, at some point, at least one clause has all of its literals evaluated to false, therefore making the whole formula false, the backtracking procedure begins. It goes backwards through the tree until it finds a variable whose value has not yet been set to both true and false and changes its value. All variables coming after it in the tree are unset again.

Example graph for $G = \{\{\neg x_0, \neg x_1, \neg x_2\}, \{\neg x_0, x_2\}, \{x_0, x_1, x_2\}\}$:



The backtracking algorithm has just discovered that setting $x_2 = 0$ makes the clause $\{\neg x_0, x_2\}$ false. So it goes back to explore the $x_1 = 0$ path and deletes the value of x_2.

Unit propagation There is an important improvement to that procedure that we ignored until now. If we have a closer look at formula G again, we see that after setting $x_0 = 1$, the clause $\{\neg x_0, x_2\}$ can only become true if we set $x_2 = 1$. And then, the clause $\{\neg x_0, \neg x_1, \neg x_2\}$ again, can only become true if we set $x_1 = 0$. We see that actually there was no need to branch on those variables, because their values (given $x_0 = 1$) can be deduced. We call this *unit propagation* and the clause forcing the decision a *unit clause* (Biere et al., 2009, p. 133). It is a very important technique in SAT solving because

unit clauses occur very often when assigning variable values iteratively (Knuth, 2015, p. 31)(Biere et al., 2009, p. 138).

2.2 Resolution

If a formula F contains the clauses $\{\neg x\} \cup A'$ and $\{x\} \cup A''$ (with A' and A'' being disjunctions of other literals), we can derive the new clause $A = \{A'\} \cup \{A''\}$ and append it to F without changing its semantics (Knuth, 2015, p. 54). This works because if a satisfying assignment $\mathcal{A}$ makes x false then a literal of A'' must consequently be set to true, thus also evaluating A to true. If $\mathcal{A}$ makes x true, it is the other way around. This procedure is called *Resolution* and is a proof calculus for propositional formulae. Note that there may be multiple ways in which two clauses can be resolved but throughout this paper it should be clear upon which variables we resolve.

3 Principles of CDCL

Now that we have seen how Backtracking and Resolution work, we are ready to merge these approaches.

From now on let n be the number of variables of the formula.

3.1 The trail

When applying CDCL, rather than exploring a search tree of assignments like in DPLL, we build up a trail of literals which have been made true, respectively, by setting their variables' value appropriately, and which do not falsify any clauses yet. We use an enumerator t starting from 0, to count up the entries in the trail, which are the literals L_t. When we reached $t = n$ (the number of assigned variables equals the total number of variables), we found a satisfying assignment. In parallel, we count up a level d, also starting from 0, which will be explained below. (Knuth, 2015, p. 62)

Here too, we use unit propagation. Whenever we observe that a clause c has all of its literals except one evaluated to false and the remaining literal l has not been assigned a value yet, we can say that $l = 1$ must hold and the variable corresponding to l must receive the appropriate value: $x_i = 1$, if l is of the form $l = x_i$ and $x_i = 0$, if it is of the form $l = \neg x_i$. (Davis & Putnam, 1960, pp. 209–211)

This is realized by incrementing t, setting $L_t = l$, updating the value of l's variable and introducing c as the so-called *reason* of l. This reason enables us to better handle conflicts later. The level d of this new trail entry is the same as of the previous one. (Knuth, 2015, p. 62)

But, as soon as there are no more unit clauses present, we must branch on a new literal l which we pick following a heuristic approach. In this case, we increment t *and* the level d, we set $L_t = l$, update the value of l's variable and put Λ as L_t's reason, which means that it was a decision rather than forced. (Marques-Silva & Sakallah, 1996, p. 221)(Knuth, 2015, p. 62)

Consider, as an example, the clauses

$$\{\{x_1, x_3, x_4\}, \{\neg x_2, \neg x_5\}, \{x_3, \neg x_4, x_5, \neg x_6\}, \{\neg x_1\}, \{x_1, \neg x_2, \neg x_4, x_6\}\}$$

A possible trail would look like this (table design inspired by Knuth (2015, pp. 63–65)):

t	L_t	level	reason
0	$\neg x_1$	0	$\{\neg x_1\}$
1	x_2	1	Λ
2	$\neg x_5$	1	$\{\neg x_2, \neg x_5\}$
3	$\neg x_3$	2	Λ
4	x_4	2	$\{x_1, x_3, x_4\}$
5	x_6	2	$\{x_1, \neg x_2, \neg x_4, x_6\}$
			$\vdots$

Forcing $x_1 = 0$ because of the presence of the original unit clause $\{\neg x_1\}$ was the first thing to do. Then, a decision had to be made. In this case we decided to set $x_2 = 1$ and established a new level. After that, we had a unit clause again, and so it goes on.

Note hereby that this example is for illustration purpose only. A real-world CDCL solver would make better decisions.

Up until now, the process did not differ essentially from normal DPLL.

3.2 Conflict clauses and backjumping

Consider our previous example again. When we want to continue building up our trail, looking at the clause $\{x_3, \neg x_4, x_5, \neg x_6\}$, we can conclude $\neg x_6$ must be true. But now we have a problem, since we already decided for the opposite.

t	L_t	level	reason
0	$\neg x_1$	0	$\{\neg x_1\}$
1	x_2	1	Λ
2	$\neg x_5$	1	$\{\neg x_2, \neg x_5\}$
3	$\neg x_3$	2	Λ
4	x_4	2	$\{x_1, x_3, x_4\}$
5	x_6	2	$\{x_1, \neg x_2, \neg x_4, x_6\}$
6	$\neg x_6$	2	$\{x_3, \neg x_4, x_5, \neg x_6\}$

We call this situation a conflict (Marques-Silva & Sakallah, 1996, p. 221). It brings us to the crucial point of the CDCL algorithm: conflict resolution, learning and backjumping (Biere et al., 2009, p. 137). We introduce new notions: we say a literal l *directly depends* on another literal l' if and only if the reason for l contains l' (Knuth, 2015, p. 63). Also, we say that a literal l *depends* on some other literal l' if and only if either l directly depends on l' or, transitively, a literal in the reason of l depends on l' (Knuth, 2015, p. 63).

In our example we call $c = \{x_3, \neg x_4, x_5, \neg x_6\}$ the conflict clause. It cannot be satisfied because it would need $\neg x_6$ to become true, but x_6 was forced by unit propagation. We take now the reason of x_6 and resolve it with c to obtain $c' = \{x_1, \neg x_2, x_3, \neg x_4, x_5\}$. Unfortunately, this is not enough; we cannot integrate c' in our current level since we still have x_4, on which x_6 depended, forced with the reason $\{x_1, x_3, x_4\}$. The solution is to restart the whole procedure by setting c' as our new c (Knuth, 2015, p. 63) and resolving it with the reason of x_4 to get $c' = \{x_1, \neg x_2, x_3, x_5\}$.

Now, there is only one dependency of $\neg x_6$ left on our current level: $\neg x_3$. This is the point where we can stop resolving. We look for the last level on which another dependency appeared, which is $\neg x_5$ on level 1, and discard all levels after it. We append x_3 with the reason c' to the trail (because of unit propagation) and proceed. This we call backjumping and c' is our newly learned clause. (Knuth, 2015, p. 63)

t	L_t	level	reason
0	$\neg x_1$	0	$\{\neg x_1\}$
1	x_2	1	Λ
2	$\neg x_5$	1	$\{\neg x_2, \neg x_5\}$
3	x_3	1	$\{x_1, \neg x_2, x_3, x_5\}$
		$\vdots$	

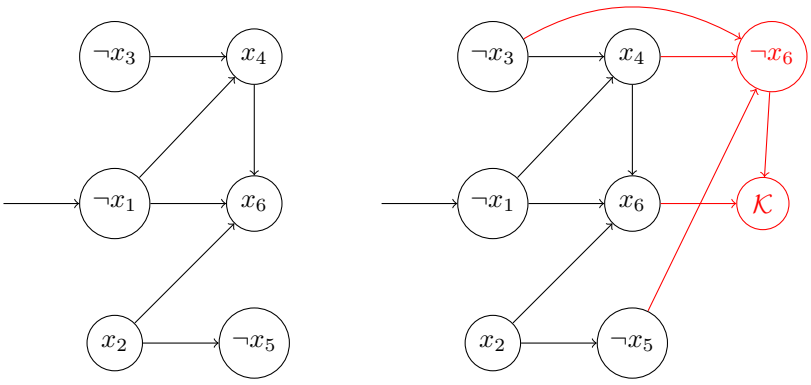
More generally speaking, the conflict resolution procedure goes as follows: If we observe that a clause c has all of its literals falsified by the current assignment, then c is a conflict clause. We do the following until c contains at most one literal occurring in the current trail level: Pick a literal from the current trail level whose negation is in c and resolve c with its reason. Then call the resolvent the new c. (Knuth, 2015, p. 63)

After that, we backjump in the trail (variables become unassigned again) up to the highest decision level where c is a unit clause and then unit propagation is applied. (Knuth, 2015, p. 63)

3.3 The implication graph

A nice way to illustrate the functionality of CDCL are implication graphs. We create a node for each assigned variable and let edges correspond to direct dependencies. This way we create a directed acyclic graph. Nodes without predecessors represent decision assignments and $\mathcal{K}$ is a conflict node (just illustrating the fact that there is a conflict). (Marques-Silva & Sakallah, 1996, p. 222)

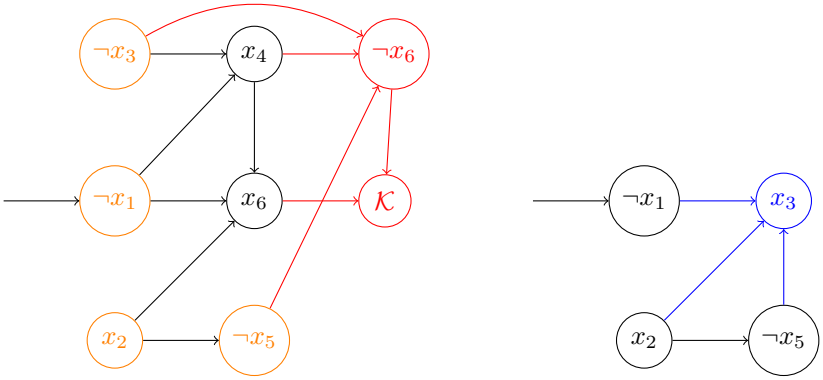
So the implication graph for our example would be:



before conflict with conflict

Upon a conflict, we identify all the nodes from which the conflict originated, i.e. dependencies of x_6 or $\neg x_6$, which are in our case all nodes, and remove from

these the nodes that were created after our most recent decision. We obtain $(\neg x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_5)$. We know that by setting all these literals to true, we get a conflict. So at least one of them must be a false assignment. The new clause we learn is the negation of $(\neg x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_5)$, which is $\{x_1, \neg x_2, x_3, x_5\}$. We can now take this as the reason to purge our last decision and all after it and set $x_3 = 1$. (Marques-Silva & Sakallah, 1996, pp. 222–223)



conflict analysis learning

Note that the analysis procedure based on implication graphs yields the same result as the formal way with resolution and is not meant to be implemented but merely gives an intuition of what is the idea behind conflict analysis and resolution.

3.4 The algorithm

To get a clearer view, Algorithm 4.1 shows the pseudocode for the CDCL algorithm without any further enhancements (taken from Biere et al. (2009, p. 136) and modified). ϕ corresponds to the formula in CNF and ν is the assignment of the variables. The tuple (x, v) returned by PickBranchingVariable(ϕ, ν) is a variable x together with a value $v \in \{0, 1\}$. In ConflictAnalysis we analyze the most recent conflict and thereby learn a new clause. The Backjump procedure jumps back to the level β computed by ConflictAnalysis.

4 Implementation

Let us now look at how the algorithm is implemented in real life by the example of the implementation of Knuth (2015, pp. 66–68).

4.1 Clauses

We use a monolithic array `MEM` to hold the original formula's clauses as well as the newly learned clauses. In this array, a clause $c = \{l_0, l_1, \cdots l_{k-1}\}$ with $k > 1$ is represented by its position `c` in `MEM` as follows: Each literal l_j of c is stored in `MEM[c + j]`. The length of c is stored in `MEM[c - 1]`. Unit clauses are not stored in `MEM` but are treated differently since they appear at level 0 of the trail.

Algorithm 1 Conflict driven clause learning

```
function CDCL(φ, ν)
    if (UnitPropagation(φ, ν) == CONFLICT) then
        return UNSAT
    end if
    d ← 0                                                    ▷ Decision level
    while (not AllVariablesAssigned(φ, ν)) do
        (x, v) = PickBranchingVariable(φ, ν)                 ▷ Decide stage
        d ← d + 1              ▷ Increment decision level due to new decision
        ν ← ν ∪ {(x, v)}
        if (UnitPropagation(φ, ν) == CONFLICT) then          ▷ Deduce stage
            β = ConflictAnalysis(φ, ν)                       ▷ Diagnose stage
            if (β < 0) then
                return UNSAT
            else
                Backjump(φ, ν, β)
                d ← β          ▷ Decrement decision level due to backjumping
            end if
        end if
    end while
    return SAT                                               ▷ Return satisfiability
end function
```

`MEM[c - 5]` to `MEM[c - 2]` contain additional information about a clause (see below). (Knuth, 2015, p. 66)

Two additional values are remembered: `MINL` and `MAXL`. Since the learned clauses appear after the initial ones in `MEM`, we are able to distinguish between them. `MINL` stands for the lowest index in which learned clauses are stored and `MAXL - 1` is the highest index in `MEM` which is occupied by a learned clause (and also by any clause). (Knuth, 2015, p. 66)

4.2 Literals

Assume the variables are $x_1, x_2, \cdots, x_n$. We represent x_k by k. The corresponding literals x_k and $\neg x_k$ are respesented by $2 \cdot k$ and $2 \cdot k + 1$, respectively. In total we have $2 \cdot n$ literals. (Knuth, 2015, p. 66)

One important concept for efficient CDCL implementation is literal watching realized by lazy data structures. Instead of knowing every literals assigned value all the time, we just care whether in every clause there are at least two literals which are not false. If there is only one literal left which is not false and its corresponding variable has not been assigned yet, we apply unit propagation. If all literals in a clause are false, we have a conflict. (Biere, 2008, pp. 78–80)

In order to implement literal watching we keep a watch list W_l for every literal l. It is a linked list and can either be 0 which means l is not watched in any clause or point to a clause. The watched literals in a clause are always in the first and second position. If one of those becomes false, we search in clause for a new literal to watch and swap it with the former watchee. If c is the first clause in which l is watched, W_l can be either `MEM[c - 2]`, if $l = l_0$ in c, or `MEM[c - 3]`, if $l = l_1$. (Biere, 2008, pp. 78–80)

4.3 Variable attributes

As described by Knuth (2015, pp. 66–67), for each variable x_k (represented by k) of our formula we keep track of six attributes:

`VAL(k)` is the value of a variable. We set `VAL(k)` $\leftarrow -1$ as long as variable x_k is unassigned. But when it receives a value we assign to `VAL(k)` $2 \cdot d$ if x_k has been made true and $2 \cdot d + 1$ if $\neg x_k$ has been made true. Remember here that d is our current (decision) level.

`OVAL(k)` Whenever a variable x_k becomes unassigned again due to a backjumping process, we set `OVAL(k)` $\leftarrow$ `VAL(k)` to remember its old value which is often a good guess if we happen to branch on x_k in the future.

`TLOC(k)` is the trail location t of when x_k was assigned.

`ACT(k)` is an activity score which tells us how much x_k is qualified for becoming our next decision variable. When looking for a variable to branch on, we try to keep focus on variables that were involved in recent conflicts. Thus we increase a `ACT(k)` whenever x_k is involved in a conflict, for details see below.

`HLOC(k)` All unassigned variables (and possibly others) are stored in a `HEAP` which always provides the variable with the highest `ACT(k)`. We must update this `HEAP` whenever we resolve a conflict. It shall hold that `ACT(HEAP[j])` $\leq$ `ACT(HEAP[(j - 1) >> 1])`, for $0 < j < h$, where `>>` is the bitshift operator (which we will later use to reduce a literal to its variable) and h the number of elements of the `HEAP`. `HLOC(k)` is the current location of x_k in the `HEAP`.

`S(k)` is a stamp that is used for a more efficient version of conflict resolution and will be explained later.

Let us have a look at the details of our activity scores. Everytime we analyse a conflict, we identify each variable x_k involved in it and add to its `ACT(k)` a summand ρ^{-i} when we have the ith conflict. Here $0 < \rho < 1$, e.g. $\rho = 0.95$. This way, every new conflict increases the activity scores a bit more and older conflicts become more and more irrelevant. At some point the activity scores will become too huge for our program to handle. To cope with this, we divide all of them by a value of e.g. 10^{100} as soon as any `ACT(k)` exceeds this value. A variable called `DEL` is used to store the current scaling factor ρ^{-M} that we obtained after having resolved M conflicts. (Knuth, 2015, p. 67)

4.4 Flushing clauses

Many of the clauses that a CDCL run learns are very long and/or almost never used. They slow down our algorithm in several ways. (Knuth, 2015, p. 71)

To get rid of those clauses we purge them out of `MEM` whenever the number M of learned clauses exceeds some threshold M_p, which could be e.g. 10000. Of course, we cannot purge clauses which are currently used as reasons in our trail, but all others we can purge and we will try to do this for at least half of them. (Knuth, 2015, p. 72)

A good metric to do so is the literal block distance. It is defined, for a clause c, as the number of different levels of the trail in which literals of c appear. We purge those clauses with the smallest literal block distance. (Audemard & Simon, 2009, pp. 401–402)

4.5 Flushing literals

Apart from flushing clauses from time to time, we want to do this also with literals which might have become out of date in the sense that they are assignments that no longer suit our current direction of exploring the search space. (Knuth, 2015, p. 75)

In order to do so, we keep an eye on the activity scores of the literals and when it seems that there are enough literals which are no longer involved in recent conflicts, we flush our trail up to the point to which there are no more such literals in it. (van der Tak et al., 2011, pp. 134–136)

Then, we will see most of the other literals again soon because they had high activity scores and will therefore probably be branched on, receiving their `OVAL` (see below).

4.6 Efficient conflict resolution

We will see now how we can form the new clause we are learning during a conflict in a faster way than simple iterative resolution. If we have a conflict on the variable of l and our conflict clause is $\{\neg l, \neg a_1, \cdots, \neg a_k\}$, we apply the following procedure: Set up an array which will in the end contain the literals of our newly learned clause. Stamp every literal a_i with a unique number which is then the `S` attribute. Insert every $\neg a_i$, whose complement a_i was set before the current level but after level 0, into the array. Then, stamp l and set up a counter for the number of stamped literals of level d (the current decision level). Look for a literal L_t in the trail with our currently used stamp `S`. Say, it has the reason $\{L_t, \neg a'_1, \cdots, \neg a'_k\}$. If our counter is bigger than 1, stamp each a'_i and put it into the array if it has not been stamped before with the current stamp. Then decrease the counter and start over, looking for a new L_t. When the counter becomes 1, put $\neg L_t$ into the array and return the disjunction of the literals in it as the new clause. (Knuth, 2015, p. 64)

This technique reduces the redundancy of the resolution procedure regarding both storing intermediate results and computing the actual resolvents.

4.7 Bringing it all together

The following description of the CDCL algorithm is taken from Knuth (2015, p. 68), who based it on MiniSAT by Eén & Sörensson (2003) and updated it according to newer techniques.

Step 1 [Initialize] First we set n to the number of variables occurring in our formula. Then we do the following for every variable k (remember, we represent x_k by k): Set `VAL(k)`, `OVAL(k)` and `TLOC(k)` to -1 (unassigned). Set `ACT(k)` and `S(k)` to 0. Set R_{2k} and R_{2k+1} to Λ. Here, R_l denotes the reason of l. Create a random permutation $p_1 \dots p_n$ of $\{1, \dots, n\}$ and set `HLOC(k)` to $p_k - 1$ and `HEAP(`$p_k - 1$`)` to k. After that, initialize `MEM`, `MINL`,

`MAXL` and the watch lists. Set i_0, d, s, M and G to 0 (G always points to the last literal for which we already did watch list updating and therefore also possibly unit propagations). Then start filling the trail with the literals from the unit clauses. The last of them is L_{F-1}. This way we implicitly set F. Finally, set h to n and `DEL` to 1.

Step 2 [Level complete?] Jump to Step 5 if $G == F$. Otherwise proceed as normal with Step 3.

Step 3 [Advance G] Let l be L_G and then increment G. Execute Step 4 for all clauses c in which $\neg l$ is watched. If no conflict occurred, go back to Step 2.

Step 4 [Does c force a unit?] Assume $c = \{l_0, l_1, \cdots, l_{k-1}\}$ and that the watched $\neg l$ is l_1. If it is l_0 instead, swap l_0 and l_1. If l_0 is in the trail, everything is fine. If not, swap l_1 with a non-false literal of c not currently watched and make it a new watchee. Update the respective watch lists. If there is no such literal, look at l_0. If l_0 is unassigned, we have a unit propagation: Set L_F to l_0, `TLOC(`l_0 `>> 1)` to F, set the `VAL` of l_0's variable appropriately. Set R_{l_0} to c and increment F. If l_0 is false, we have a conflict, jump to Step 7.

Step 5 [New level?] If $F == n$, return SAT and the assignment of the variables. If not, check for the necessity of flushing literals or clauses. If literals were flushed, go back to Step 2. If not, increment d and set i_d to F.

Step 6 [Make a decision] Pop the variable k with the highest activity score from the `HEAP`. If it was already assigned (by unit propagation), repeat this step. Otherwise, check whether an `OVAL` has been set for k. If so, assign the same truth value to k again and set `VAL(k)` to `OVAL(k)`. If not, make it true. Then let l be the chosen literal (represented by $2k(+1)$) and set L_F to l, `TLOC(`l `>> 1)` to F, R_l to Λ and increment F. Here we have $F = G + 1$. Then go back to Step 3.

Step 7 [Resolve a conflict] Return UNSAT if $d == 0$. If not, resolve the conflict with the efficient method described above with c being the conflict clause. Increase `ACT(`l `>> 1)` by `DEL` for every literal l stamped by the procedure. Update the `HEAP` correspondingly. Then set d' to the backjumping level.

This step corresponds to ConflictAnalysis in Algorithm 4.1.

Step 8 [Backjump] While $F > i_{d'+1}$ do clean up: decrement F, set l to L_F, k to $l >> 1$, `OVAL(k)` to `VAL(k)`, `VAL(k)` to -1, R_l to Λ and, if k is currently not in the `HEAP`, put it back in. Finally, set G to F and d to d'.

This step corresponds to Backjump in Algorithm 4.1.

Step 9 [Learn] If $d > 0$, set c to `MAXL`, add c to `MEM` and set the new `MAXL`. Set L_F to l' (the literal forced by the newly learned clause), `TLOC(`l' `>> 1)` to F, $R_{l'}$ to c, `DEL` to `DEL`/ρ and increment M and F. Then go back to Step 3.

5 Results

It is important to say that CDCL is a sound and complete algorithm for the propositional satisfiability problem (Marques-Silva, 1995, pp. 227–228).

To certify the result of CDCL solver's run we have to look at two cases:

1. We return SAT. In this case our algorithm also returns a satisfying assignment for the formula and we can easily check whether it is really a satisfying assignment.

2. We return UNSAT. This is a bit more difficult, but also here our CDCL algorithm comes in handy as the sequence of learned clauses is a certificate of unsatisfiability. Every set of learned clauses implies by unit propagation that the negation of the next learned clause would lead to a contradiction and therefore the next clause must also hold. At the end we have the empty clause.

References

Audemard, G., & Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, 399-404.

Biere, A. (2008). PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, *4*(2-4), 75-97. doi: 10.3233/SAT190039

Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.). (2009). *Handbook of satisfiability* (Vol. 185 of Frontiers in Artificial Intelligence and Applications). IOS Press. doi: 10.3233/978-1-58603-929-5-131

Davis, M., & Putnam, H. (1960, 07). A computing procedure for quantification theory. *Journal of the ACM*, *7*, 201-215. doi: 10.1145/321033.321034

Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. *Lecture Notes in Computer Science*, *2919*, 502-518.

Knuth, D. E. (2015). *The art of computer programming: Satisfiability, volume 4, fascicle 6.* Addison-Wesley Professional.

Marques-Silva, J. P. (1995). *Search algorithms for satisfiability problems in combinational switching circuits* (phdthesis). University of Michigan.

Marques-Silva, J. P., & Sakallah, K. A. (1996, November). GRASP—a new search algorithm for satisfiability. *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, 220-227.

van der Tak, P., Ramos, A., & Heule, M. (2011). Reusing the assignment trail in cdcl solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, *7*(4), 133-138. doi: 10.3233/SAT190082