WIKIPEDIA

* Idea: apply this to other NP-C problems ??

Conflict-Driven Clause Learning

In computer science, **Conflict-Driven Clause Learning (CDCL)** is an algorithm for solving the Boolean satisfiability problem (SAT). Given a Boolean formula, the SAT problem asks for an assignment of variables so that the entire formula evaluates to true. The internal workings of CDCL SAT solvers were inspired by DPLL solvers.

Conflict-Driven Clause Learning was proposed by Marques-Silva and Sakallah (1996,[1] 1999 [2]) and Bayardo and Schrag (1997 [3])

Contents

isvalid(f) = !f unsat

not valid = f sat -> cex

Background

Background knowledge about the following issues is needed to have a clear idea about the CDCL algorithm.

Boolean satisfiability problem

The satisfiability problem consists in finding a satisfying assignment for a given formula in conjunctive normal form (CNF).

An example of such a formula is:

((not A) or (not C)) and (B or C),

or, using a common notation:

$$(A' + C')(B + C)$$

Popular because many probs (e.g NP-complete probs) can be reduced to it.

Program semantics can be represented using formulas (VC's) formula

verification / sysnthesis reduced to checks?

where A,B,C are Boolean variables, A', C', B, and C are literals, and $A' + C'$ and $B + C$ are clauses.

A satisfying assignment for this formula is e.g.:

$$A = False, B = False, C = True$$

since it makes the first clause true (since A' is true) as well as the second one (since C is true).

This examples uses three variables (A, B, C), and there are two possible assignments (True and False) for each of them. So one has $2^3 = 8$ possibilities. In this small example, one can use brute-force search to try all possible assignments and check if they satisfy the formula. But in realistic applications with millions of variables and clauses brute force search is impractical. The responsibility of a SAT solver is to find a satisfying assignment efficiently and quickly by applying different heuristics for complex CNF formulas.

3-SAT : NP complete (still decidable)

Unit clause rule (unit propagation)

If an unsatisfied clause has all but one of its literals or variables evaluated at False, then the free literal must be True in order for the clause to be True. For example, if the below unsatisfied clause is evaluated with $A = False$ and $B = False$ we must have $C = True$ in order for the clause $(A \vee B \vee C)$ to be true.

Boolean constraint propagation (BCP)

The iterated application of the unit clause rule is referred to as unit propagation or Boolean constraint propagation (BCP).

→ not used in DP

Resolution

Consider two clauses $(A \vee B \vee C)$ and $(\neg C \vee D \vee \neg E)$. The clause $(A \vee B \vee D \vee \neg E)$, obtained by merging the two clauses and removing both $\neg C$ and C, is called the resolvent of the two clauses.

DP algorithm

The DP algorithm has been introduced by Davis and Putnam in 1960. Informally, the algorithm iteratively performs the following steps until no more variables are left in the formula:

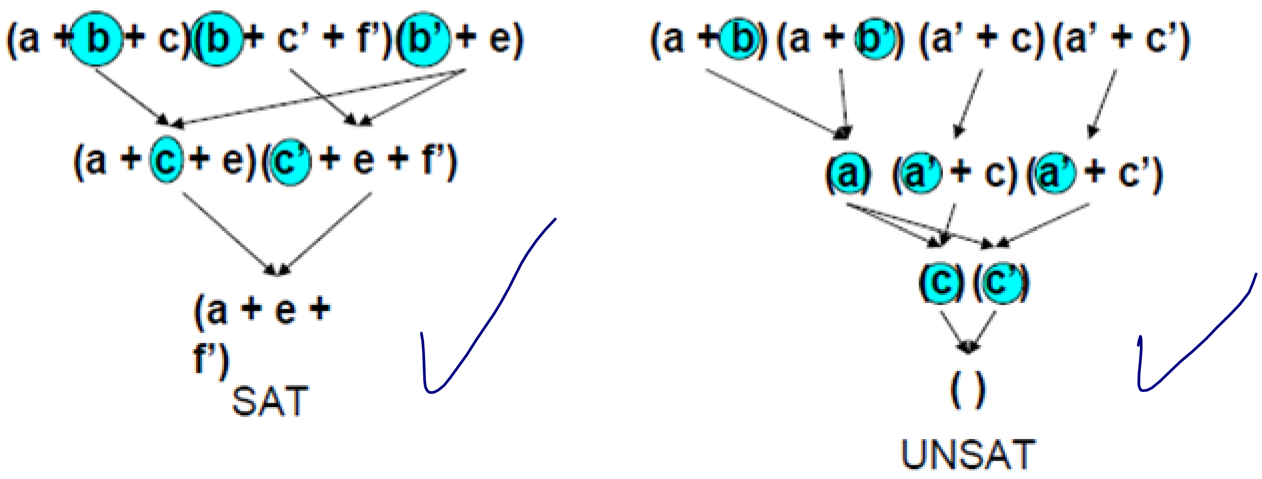
- Select a variable A and add all non-tautological resolvents upon A (a resolvent is non-tautological if it does not contain V and $\neg V$ for some variable V).
- Remove all original clauses that contain A.

See more details here DP Algorithm

for all variable A →

add Resolvants from clauses containing A and clauses containing $\bar{A}$

remove orignal clauses



DPLL algorithm

Davis, Putnam, Logemann, Loveland (1962) had developed this algorithm. Some properties of this algorithms are:

- It is based on search.
- It is the basis for almost all modern SAT solvers.
- It uses learning and chronological backtracking (1996).

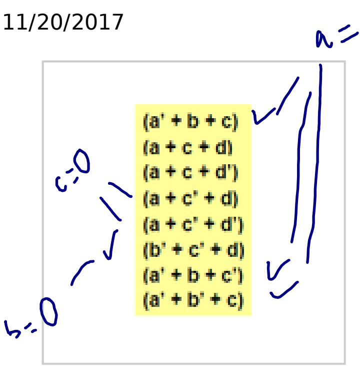
See more details at DPLL algorithm. An example with visualization of DPLL algorithm having chronological backtracking:

- how to choose variables ?

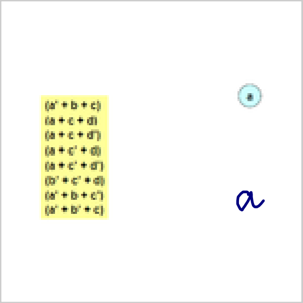
- how to choose values to variables

$(a \vee c \vee e) \wedge (\bar{c} \vee \bar{f} \vee e)$

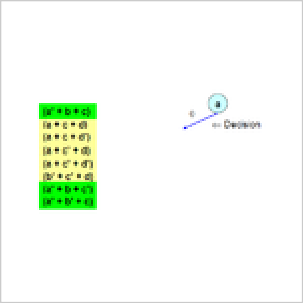
$a \vee e \vee \bar{f}$



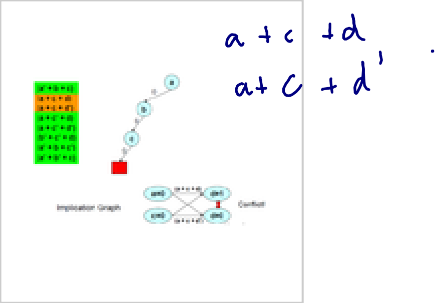
All clauses making a CNF formula



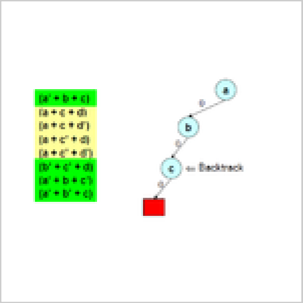
Pick a variable (a) ✓



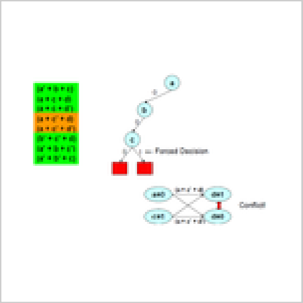
Make a decision, variable a = False (0), thus green clauses becomes True ✓



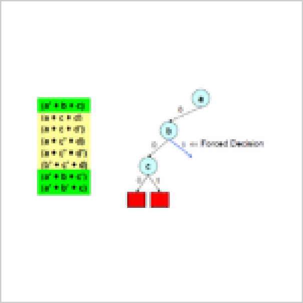
After several decision making, we find an implication graph that leads to a conflict.



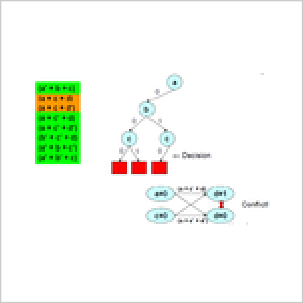
Now backtrack to immediate level and by force assign opposite value to that variable



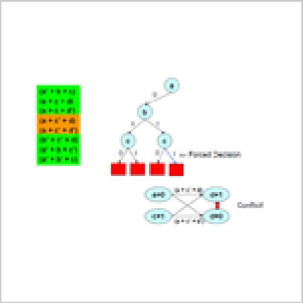
But a forced decision still leads to another conflict



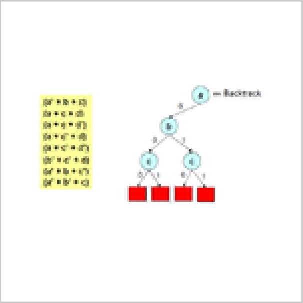
Backtrack to previous level and make a forced decision



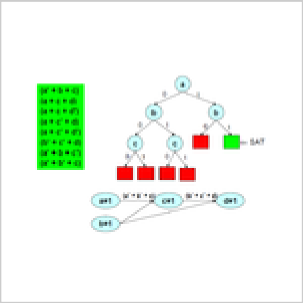
Make a new decision, but it leads to a conflict



Make a forced decision, but again it leads to a conflict



Backtrack to previous level



Continue in this way and the final implication graph

CDCL (Conflict-Driven Clause Learning)

The main difference between CDCL and DPLL is that CDCL's back jumping is non-chronological.

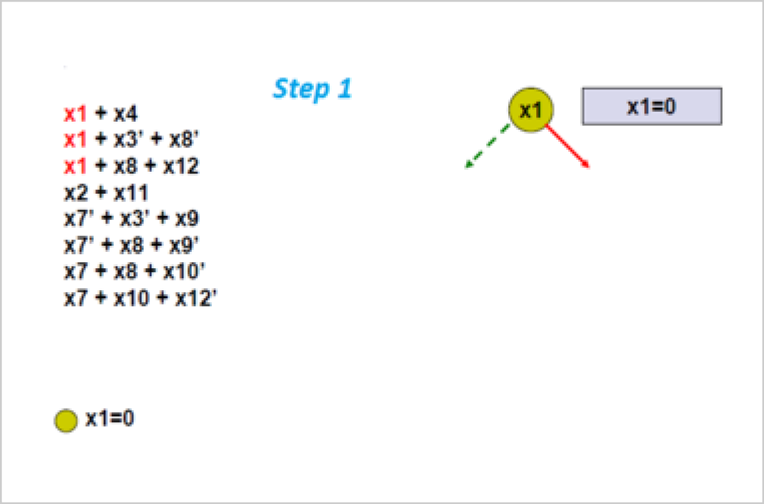
Conflict-Driven Clause Learning works as follows.

1. Select a variable and assign True or False. This is called decision state. Remember the assignment.
2. Apply Boolean constraint propagation (Unit propagation).

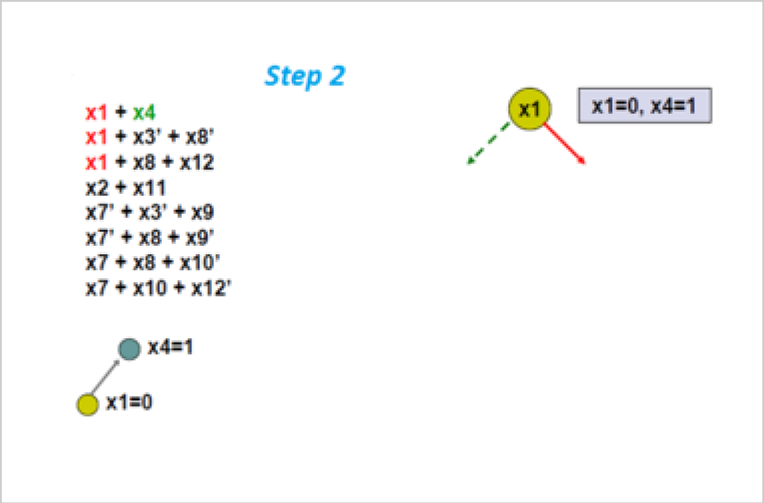
3. Build the implication graph.
4. If there is any conflict then analyze the conflict and non-chronologically backtrack ("back jump") to the appropriate decision level.
5. Otherwise continue from step 1 until all variable values are assigned.

Example

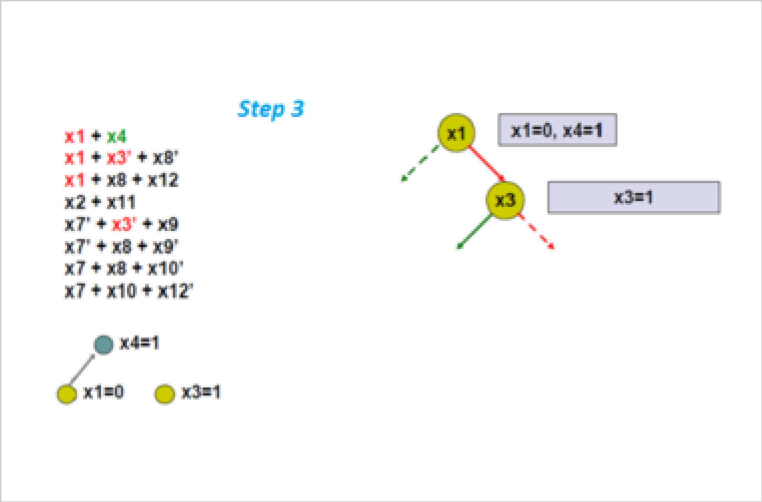
A visual example of CDCL algorithm:



At first pick a branching variable, namely *x1*. A yellow circle means an arbitrary decision.



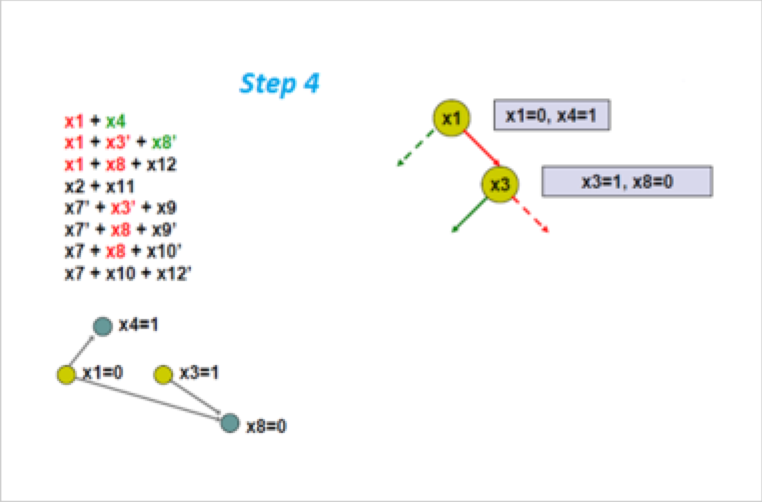
Now apply unit propagation, which yields that *x4* must be 1 (i.e. True). A gray circle means a forced variable assignment during unit propagation. The resulting graph is called an implication graph.



Step 3

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

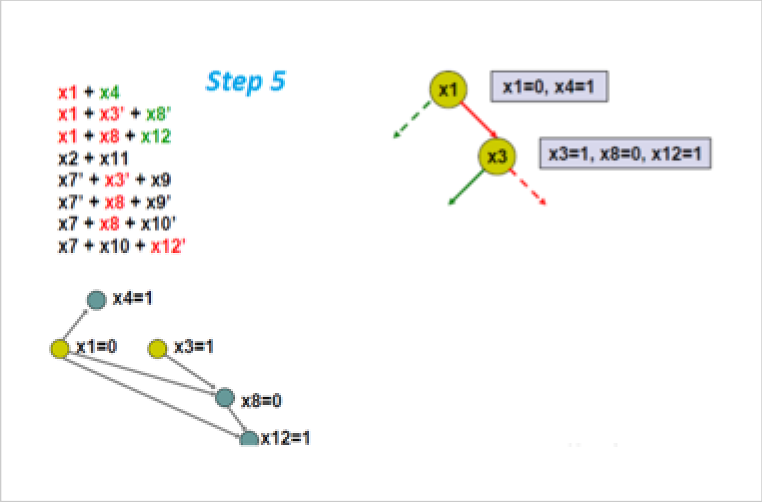
Arbitrarily pick another branching variable, *x3*.



Step 4

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

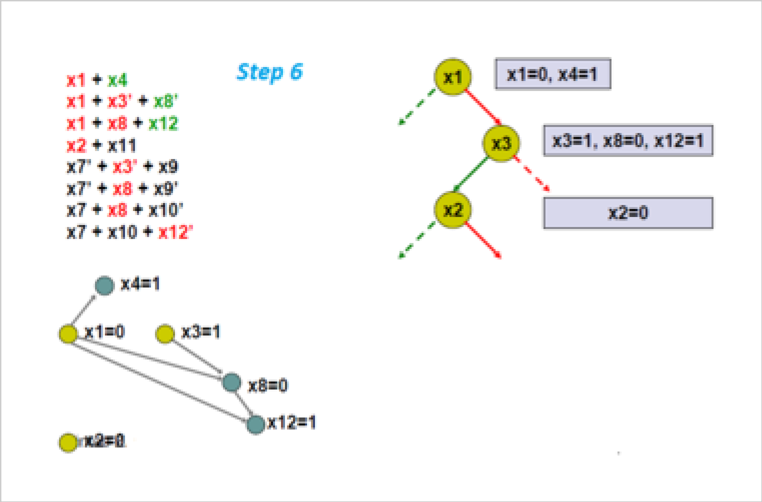
Apply unit propagation and find the new implication graph.



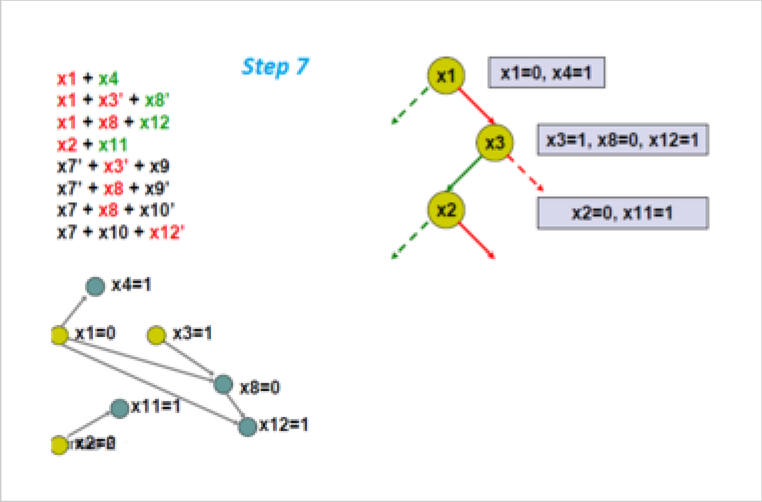
Step 5

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

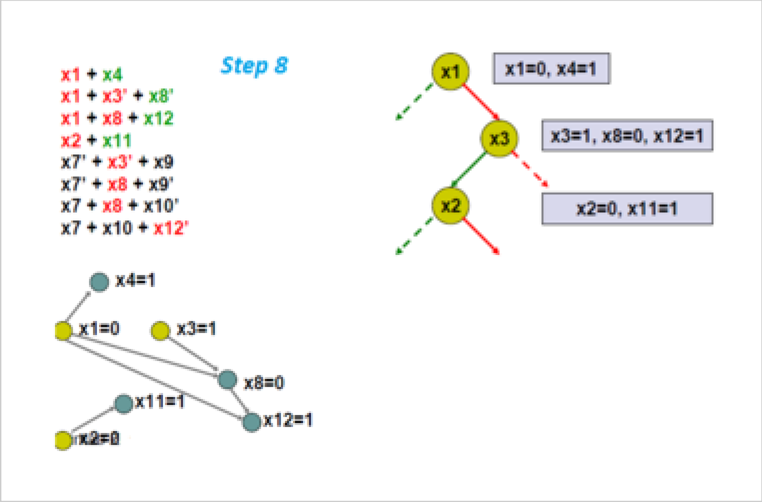
Here the variable *x8* and *x12* are forced to be 0 and 1, respectively.



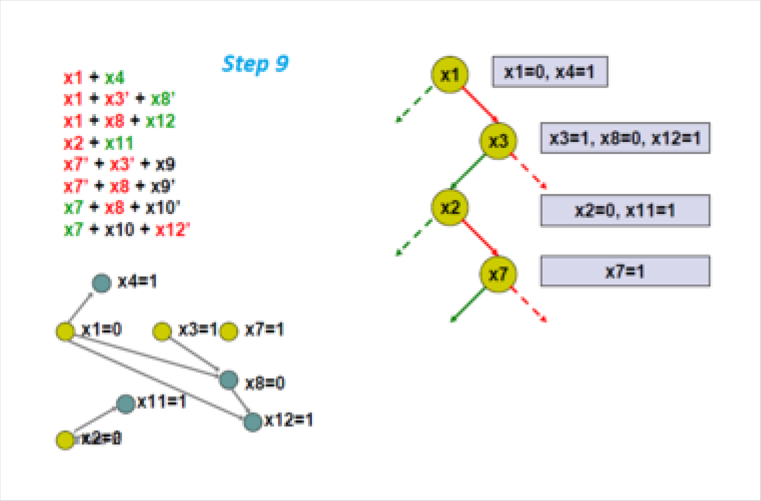
Pick another branching variable, *x2*.



Find implication graph.



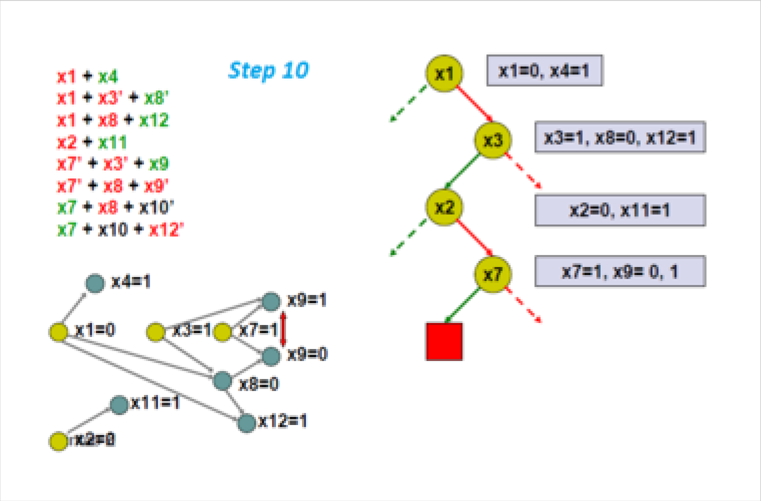
Pick another branching variable, *x7*.



Step 9

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

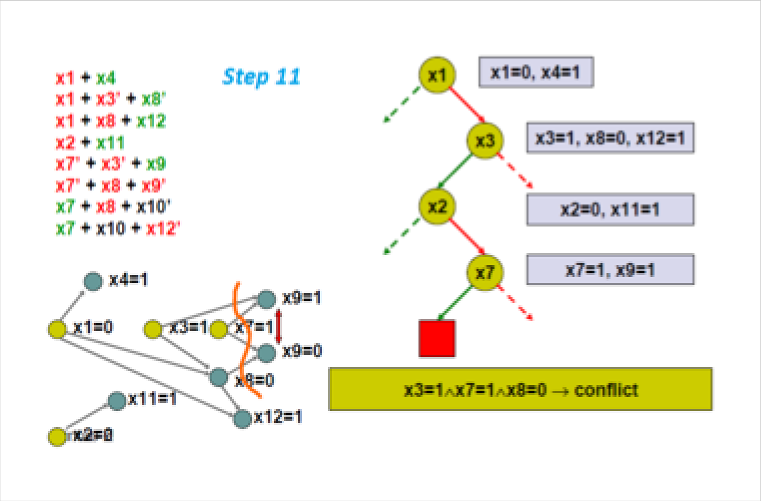
Find implication graph.



Step 10

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Found a conflict!

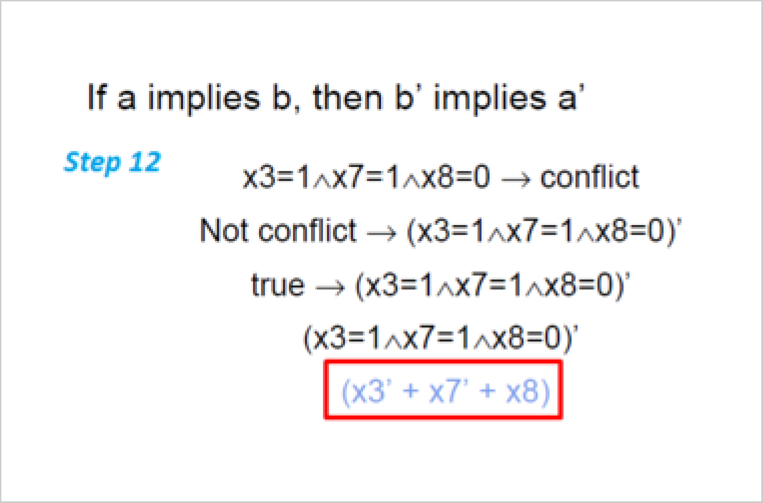


Step 11

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict

Find the cut that led to this conflict. From the cut, find a conflicting condition.



If a implies b, then b' implies a'

Step 12

$$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow \text{conflict}$$

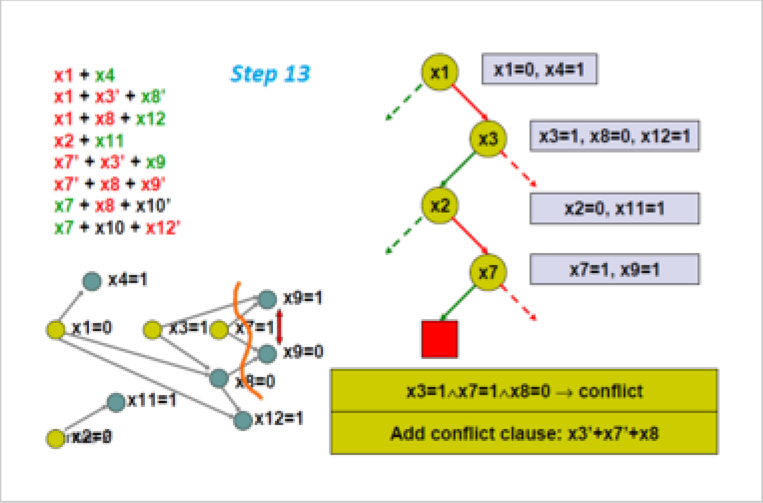
$$\text{Not conflict} \rightarrow (x3=1 \wedge x7=1 \wedge x8=0)'$$

$$\text{true} \rightarrow (x3=1 \wedge x7=1 \wedge x8=0)'$$

$$(x3=1 \wedge x7=1 \wedge x8=0)'$$

$$\boxed{(x3' + x7' + x8)}$$

Take the negation of this condition and make it a clause.



Step 13

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

x1=0, x4=1

x3=1, x8=0, x12=1

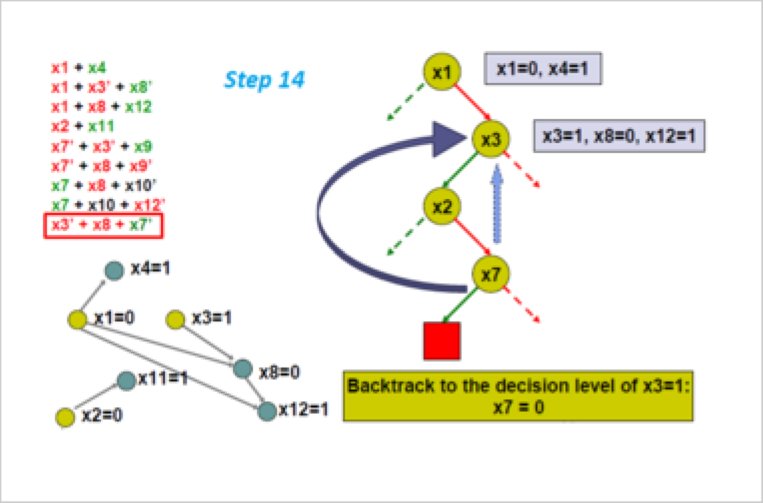
x2=0, x11=1

x7=1, x9=1

$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict

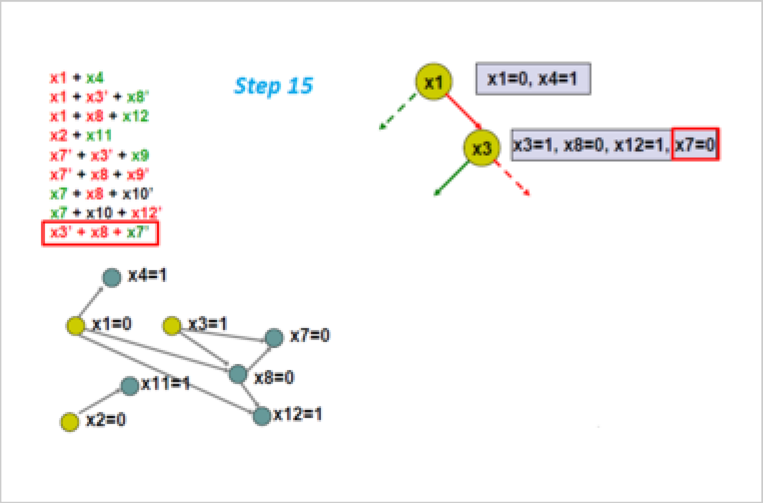
Add conflict clause: x3'+x7'+x8

Add the conflict clause to the problem.



Non-chronological back jump to appropriate decision level, which is the second highest decision level of the literals in the learned clause.

x3, x8, x7



Back jump and set variable values accordingly.

Completeness

DPLL is a sound and complete algorithm for SAT. CDCL SAT solvers implement DPLL, but can learn new clauses and backtrack non-chronologically. Clause learning with conflict analysis does not affect soundness or completeness. Conflict analysis identifies new clauses using the resolution operation. Therefore, each learnt clause can be inferred from the original clauses and other learnt clauses by a sequence of resolution steps. If cN is the new learnt clause, then ϕ is satisfiable if and only if ϕ ∪ {cN} is also satisfiable. Moreover, the modified backtracking step also does not affect soundness or completeness, since backtracking information is obtained from each new learnt clause.[4]

Applications

The main application of CDCL algorithm is in different SAT solvers including:

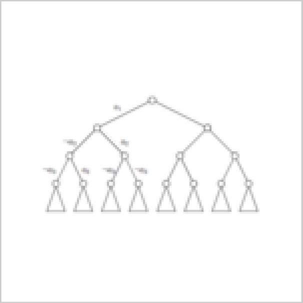
- MiniSAT
- Zchaff SAT

- Z3
- ManySAT etc.

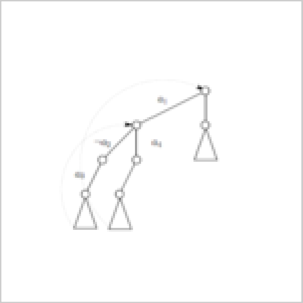
The CDCL algorithm has made SAT solvers so powerful that they are being used effectively in several real world application areas like AI planning, bioinformatics, software test pattern generation, software package dependencies, hardware and software model checking, and cryptography.

Related algorithms

Related algorithms to CDCL are the DP and DPLL algorithm described before. The DP algorithm uses resolution refutation and it has potential memory access problem. Whereas the DPLL algorithm is OK for randomly generated instances, it is bad for instances generated in practical applications. CDCL is a more powerful approach to solve such problems in that applying CDCL provides less state space search in comparison to DPLL.



DPLL: no learning and chronological backtracking.



CDCL: conflict-driven clause learning and non – chronological backtracking.

References

- Martin Davis; Hilary Putnam (1960). "A Computing Procedure for Quantification Theory". *J. ACM*. **7** (3): 201–215. doi:10.1145/321033.321034 (https://doi.org/10.1145%2F321033.321034).
- Martin Davis; George Logemann; Donald Loveland (Jul 1962). "A machine program for theorem-proving". *CACM*. **5** (7): 394–397. doi:10.1145/368273.368557 (https://doi.org/10.1145%2F368273.368557).
- Matthew W. Moskewicz; Conor F. Madigan; Ying Zhao; Lintao Zhang; Sharad Malik (2001). "Chaff: engineering an efficient SAT solver" (https://www.princeton.edu/~chaff/publication/DAC2001v56.pdf) (PDF). *Proc. 38th Ann. Design Automation Conference (DAC)*. pp. 530–535.
- Lintao Zhang; Conor F. Madigan; Matthew H. Moskewicz; Sharad Malik (2001). "Efficient conflict driven learning in a boolean satisfiability solver" (http://www.mimuw.edu.pl/~tsznuk/tmp/dpll.pdf) (PDF). *Proc. IEEE/ACM Int. Conf. on Computer-aided design (ICCAD)*. pp. 279–285.
- Presentation – "SAT-Solving: From Davis-Putnam to Zchaff and Beyond" by Lintao Zhang. (Several pictures are taken from his presentation)

1. J.P. Marques-Silva; Karem A. Sakallah (November 1996). "GRASP-A New Search Algorithm for Satisfiability". *Digest of IEEE International Conference on Computer-Aided Design (ICCAD)* (http://ieeexplore.ieee.org/document/569607/). pp. 220–227. doi:10.1109/ICCAD.1996.569607 (https://doi.org/10.1109%2FICCAD.1996.569607).
2. J.P. Marques-Silva; Karem A. Sakallah (May 1999). "GRASP: A Search Algorithm for Propositional Satisfiability" (http://www.broadinstitute.org/~ilya/area/grasp.pdf) (PDF). *IEEE Transactions on Computers*. **48** (5): 506–521. doi:10.1109/12.769433 (https://doi.org/10.1109%2F12.769433).
3. Roberto J. Bayardo Jr.; Robert C. Schrag (1997). "Using CSP look-back techniques to solve real world SAT instances" (http://cse-wiki.unl.edu/wiki/images/0/06/Using_CSP_Look-Back_Techniques_to_Solve_Real-World_SAT_Instances.pd

f) (PDF). *Proc. 14th Nat. Conf. on Artificial Intelligence (AAAI)*. pp. 203–208.

4. Biere, Heule, Van Maaren, Walsh (February 2009). *Handbook of Satisfiability* (http://satassociation.org/articles/FAIA185-0131.pdf) (PDF). IOS Press. p. 138. ISBN 978-1-60750-376-7.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Conflict-Driven_Clause_Learning&oldid=805555149"

This page was last edited on 16 October 2017, at 04:02.