A gentle introduction to SAT

Rohan Fossé

November, 27th. 2020

Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR-5800

Table of Contents

The Boolean Satisfiability Problem

Propositional formula

Definition

Let $\mathcal{V}$ be a finite set of Boolean valued variables. A *propositional formula* on $\mathcal{V}$ is defined inductively as follows:

- each of the constants false, true is a propositional formula on $\mathcal{V}$;
- if ϕ and ϕ' are propositional formulas on $\mathcal{V}$ then $\neg\phi$, $\phi \wedge \phi'$, $\phi \vee \phi'$,$\phi \Leftrightarrow \phi'$, $\phi \rightarrow \phi'$ are propositional formulas on $\mathcal{V}$ as well.
- An *assignment* on $\mathcal{V}$ is any map from $\mathcal{V}$ to $\{\mathit{false}, \mathit{true}\}$

Satisfiability

Definition

Let $\mathcal{V}$ be a finite set of Boolean variables and let ϕ be a propositional formula on $\mathcal{V}$.

- An assignment $\mathbf{v}$ on $\mathcal{V}$ is a *satisfying assignment* for ϕ if we have $\mathbf{v}(\phi) = \mathit{true}$;
- The propositional formula ϕ is said satisfiable if there exists a satisfying assignment for ϕ.
- Deciding whether or not a propositional formula is satisfiable is called the *Boolean satisfiability problem*, denoted by `SAT`.

Conjunctive normal form

Literals

A literal $(a, b, \ldots)$ is either a boolean variable x or this negation $\neg x$

Clauses

A clause C is a disjunction of literals *i.e*:

$C = a \vee b \vee .. \vee z$

Formula

A formula Φ is a conjunction of clauses *i.e*:

$\Phi = C_1 \wedge C_2 \wedge ... \wedge C_m$

Example

$\Phi = (a \vee \neg b) \wedge a \wedge (\neg a \vee \neg b)$

Resolution of a SAT formula

Let $\Phi_1 = (a \vee \neg b) \wedge a \wedge (\neg a \vee \neg b)$

Resolution

a	¬b
a	
¬a	¬b

Goal: Find an assignment of a and b such that each line is true ✓

a = ?

b = ?

Resolution of a SAT formula

Let $\Phi_1 = (a \vee \neg b) \wedge a \wedge (\neg a \vee \neg b)$

Resolution

a	¬b ✓
a	✓
¬a	¬b

Goal: Find an assignment of a and b such that each line is true ✓

a = True

b = ?

Resolution of a SAT formula

Let $\Phi_1 = (a \vee \neg b) \wedge a \wedge (\neg a \vee \neg b)$

Resolution

$$\begin{array}{ll} \mathbf{a} & \neg\mathbf{b} \; \checkmark \\ \mathbf{a} & \checkmark \\ \neg\mathbf{a} & \neg\mathbf{b} \; \checkmark \end{array}$$

Goal: Find an assignment of a and b such that each line is true ✓

a = `True`

b = `False`

Φ_1 is SAT ☺

Another resolution of a SAT formula

Let $\Phi_2 = (a \vee \neg b) \wedge b \wedge (\neg a \vee \neg b)$

Resolution

a	¬b
b	
¬a	¬b

Goal: Find an assignment of a and b such that each line is true ✓

a = ?

b = ?

Another resolution of a SAT formula

Let $\Phi_2 = (a \vee \neg b) \wedge b \wedge (\neg a \vee \neg b)$

Resolution

a	**¬b**
b	✓
¬a	**¬b**

Goal: Find an assignment of a and b such that each line is True ✓

a = ?

b = True

Another resolution of a SAT formula

Let $\Phi_2 = (a \vee \neg b) \wedge b \wedge (\neg a \vee \neg b)$

Resolution

a	**¬b**	**x**
b	✓	
¬a	**¬b**	**x**

Goal: Find an assignment of a and b such that each line is True ✓

a = ☹

b = True

Φ_2 is UNSAT ☹

Complexity and restricted versions

Complexity

- First known **NP-complete** problem, as proved by *Stephen Cook* in 1971 and *Leonid Levin* in 1973;
- Every decision problem in NP can be reduced to the `SAT` problem;
- Cook's reduction preserves the number of accepting answers.

Some restricted versions

- 3-SAT: each clause is limited to at most 3 literals → **NP-complete**
- 2-SAT: each clause is limited to at most 2 literals → **Polynomial**
- MAX-SAT: the problem of determining the maximum number of clauses that can be made true by an assignment. → **APX-Complete**

Some resctricted versions

- 3-SAT: each clause is limited to at most 3 literals → **NP-complete**
- 2-SAT: each clause is limited to at most 2 literals → **Polynomial**
- MAX-SAT: the problem of determining the maximum number of clauses that can be made true by an assignment. → **APX-Complete**

MAX-SAT

$$\phi = (x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1)$$

ϕ is not satisfiable ☹. However, there exists an assignation of ϕ s.t 3 of 4 clauses are true.
Therefore, if this formula is given as an instance of the MAX-SAT problem, the solution to the problem is 3 ☺.

Applications

The Sudoku problem

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 1: A typical Sudoku puzzle

Rules

The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid contain all of the digits from 1 to 9.

How to do this?

Reduce Sudoku to *SAT*

Goal: Reduce an instante of Sudoku to an instance (formula) ϕ_G of SAT

Rules

- Definedness: each cell, each row, each column and each block having *at least* one number from 1 to n;
- Uniqueness: same but with *at most* one number from 1 to n.

How to do this?

Rules

- Definedness: each cell, each row, each column and each sub-grid having *at least* one number from 1 to n;
- Uniqueness: same but with *at most* one number from 1 to n.

Variable s_{xyz} is assigned true *iff* the entry in row x and column y is assigned to number z.

	Definedness	Uniqueness

How to do this?

Rules

- Definedness: each cell, each row, each column and each sub-grid having *at least* one number from 1 to n;
- Uniqueness: same but with *at most* one number from 1 to n.

Variable s_{xyz} is assigned true *iff* the entry in row x and column y is assigned to number z.

	Definedness	Uniqueness
Cell	$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$	$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{8} \bigwedge_{i=z+1}^{9} (\neg s_{xyz} \vee \neg s_{xyi})$

How to do this?

Rules

- Definedness: each cell, each row, each column and each sub-grid having *at least* one number from 1 to n;
- Uniqueness: same but with *at most* one number from 1 to n.

Variable s_{xyz} is assigned true *iff* the entry in row x and column y is assigned to number z.

	Definedness	Uniqueness
Cell	$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$	$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{8} \bigwedge_{i=z+1}^{9} (\neg s_{xyz} \vee \neg s_{xyi})$
Row	$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{x=1}^{9} s_{xyz}$	$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz})$

How to do this?

Rules

- Definedness: each cell, each row, each column and each sub-grid having *at least* one number from 1 to n;
- Uniqueness: same but with *at most* one number from 1 to n.

Variable s_{xyz} is assigned true *iff* the entry in row x and column y is assigned to number z.

	Definedness	Uniqueness
Cell	$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$	$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{8} \bigwedge_{i=z+1}^{9} (\neg s_{xyz} \vee \neg s_{xyi})$
Row	$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{x=1}^{9} s_{xyz}$	$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz})$
Column	$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{y=1}^{9} s_{xyz}$	$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{i=y+1}^{9} (\neg s_{xyz} \vee \neg s_{xiz})$

How to do this?

Rules

- Definedness: each cell, each row, each column and each sub-grid having *at least* one number from 1 to n;
- Uniqueness: same but with *at most* one number from 1 to n.

Variable s_{xyz} is assigned true *iff* the entry in row x and column y is assigned to number z.

	Definedness	Uniqueness
Cell	$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$	$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{8} \bigwedge_{i=z+1}^{9} (\neg s_{xyz} \vee \neg s_{xyi})$
Row	$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{x=1}^{9} s_{xyz}$	$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz})$
Column	$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{y=1}^{9} s_{xyz}$	$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{i=y+1}^{9} (\neg s_{xyz} \vee \neg s_{xiz})$
Sub-grid	$\bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} \bigvee_{z=1}^{9} s_{(3i+x)(3j+y)z}$	Not enough space to write it down but you have the idea ☺

Table 1: Rule table

What do we get ?

The same Soduko problem solved

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 2: A typical Sudoku puzzle

Steps

1. Create the SAT formula using the different rules seen above;
2. Give this formula to a SAT solver;
3. Interpreting the result (i.e s_{134} = *true* means there is a 4 in the 1^{st} line and 3^{rd} column.)

Tseytin tranformation

Problem

How to transform any logical formula into CNF?

The tseytin transformation

Consider the following formula Φ:

$$\Phi = ((p \vee q) \wedge r) \rightarrow (\neg s)$$

Consider all subformulas (without variables):

$$\neg s$$
$$p \vee q$$
$$(p \vee q) \wedge r$$
$$((p \vee q) \wedge r) \rightarrow (\neg s)$$

Tseytin tranformation

$$\Phi = ((p \vee q) \wedge r) \rightarrow (\neg s)$$

Introduce a new variable for each subformula:

$$x_1 \leftrightarrow \neg s$$
$$x_2 \leftrightarrow p \vee q$$
$$x_3 \leftrightarrow x_2 \wedge r$$
$$x_4 \leftrightarrow x_3 \rightarrow x_1$$

Conjunct all substitutions:

$$x_4 \wedge (x_4 \leftrightarrow x_3 \rightarrow x_1) \wedge (x_3 \leftrightarrow x_2 \wedge r) \wedge (x_2 \leftrightarrow p \vee q) \wedge (x_1 \leftrightarrow \neg s)$$

All substitutions can be transformed into CNF, e.g:

$$\begin{aligned} x_2 \leftrightarrow p \vee q &\equiv (x_2 \rightarrow (p \vee q)) \wedge ((p \vee q) \rightarrow x_2) \\ &\equiv (\neg x_2 \vee p \vee q) \wedge ((\neg p \wedge \neg q) \vee x_2) \\ &\equiv (\neg x_2 \vee p \vee q) \wedge (\neg p \vee x_2) \wedge (\neg q \vee x_2) \end{aligned}$$

Polynomial-time Reduction

Definition
Problem Y is polynomial-time reductible to problem X if arbitraty instances of problem Y cans be solved using:

- Polynomial number of standard computational steps;
- Polynomial number of calls to the algorithm that solves problem X.

We note that $Y \leq_p X$

Consequences of $Y \leq_p X$

- if X can be solved in polynomial-time, then Y can alors be solved in polynomiam time;
- If Y cannot be solved in polynomial-time, then X cannot be solved in polynomial time.

3-SAT is reductible to Independent Set

Claim

3-SAT $\leq_p$ INDEPENDENT-SET

Proof

Given an instance Φ of 3-SAT, we construct an instance (G,K) of INDEPENDENT-SET that has an independent set of size k iff Φ is satisfiable.

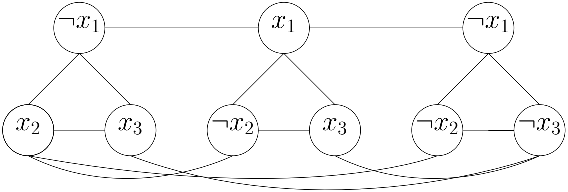
3-SAT is reductible to Independent Set

Claim

3-SAT $\leq_p$ INDEPENDENT-SET

Construction

- G contains 3 vertices for each clause, one for each literal;
- Connect the 3 literals in a clause in a triangle;
- Connect literal to each of its negations.



$$\Phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

3-SAT is reductible to Independent Set

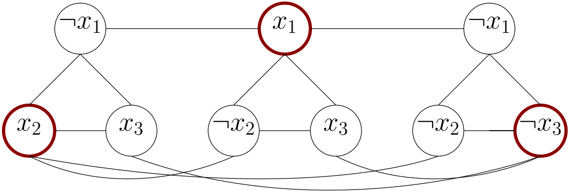
Claim

3-SAT $\leq_p$ INDEPENDENT-SET

Proof of if-part

Let s be independent set of size k.

- S must contrain exactly one vertex in each triangle;
- Set these literals to true;
- Truth assignment is consistent and all clauses are satisfied.



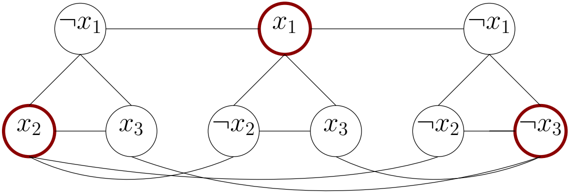
$$\Phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

3-SAT is reductible to Independent Set

Claim
3-SAT $\leq_p$ INDEPENDENT-SET

Proof of only-if part
Given satisfying assignment, select one true literals from each triangle.

This is an independent set of size k.



$$\Phi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3)$$

Many other applications

Example of applications

- Cryptography;
- Planification;
- Resolving software package dependencies

How do SAT solvers work?

The naive method

Using a truth table

The most obvious way to solve a SAT problem is to go through the truth table of the problem.

Example

$$\Phi = (a \vee b) \wedge (\neg a \vee \neg b)$$

a	b	$a \vee b$	$\neg a \vee \neg b$	Φ
true	true	true	false	false
true	false	true	true	true
false	true	true	true	true
false	false	false	true	false

$\Longrightarrow$ Complexity : $0(2^n)$

The DPLL algorithm (1962)

The **Davis–Putnam–Logemann–Loveland** (**DPLL**) algorithm is a complete, *backtracking-based* search algorithm for deciding the satisfiability of propositional logic formulae Φ

Major innovations

- Unit propagation
- Pure literal elimination
- Backtracking

Unit propagation

If a clause is a **unit clause**, i.e. it contains only a single literal l, We can apply the following two rules:

- Every clause (other than the unit clause itself) containing l is removed;
- in every clause that contains $\neg l$, this literal is deleted.

Example

$$\Phi = (a \vee b) \wedge (\neg a \vee c) \wedge (\neg c \vee d) \wedge a$$

$$\begin{gathered} a \vee b \\ \neg a \vee c \\ \neg c \vee d \\ a \end{gathered}$$

The following set of clauses can be simplified by unit propagation because it contains the unit clause a.

Unit propagation

Unit propagation

If a clause is a **unit clause**, i.e. it contains only a single literal l, We can apply the following two rules:

- Every clause (other than the unit clause itself) containing l is removed;
- in every clause that contains $\neg l$, this literal is deleted.

Example

$$\cancel{a \vee b}$$
$$\cancel{\neg a} \vee c$$
$$\neg c \vee d$$
$$a$$

The following set of clauses can be simplified by unit propagation because it contains the unit clause a.

Unit propagation

Unit propagation

If a clause is a **unit clause**, i.e. it contains only a single literal l, We can apply the following two rules:

- Every clause (other than the unit clause itself) containing l is removed;
- in every clause that contains $\neg l$, this literal is deleted.

Example

$$c$$
$$\neg c \vee d$$
$$a$$

The following set of clauses can be simplified by unit propagation because it contains the unit clause c.

Unit propagation

Unit propagation

If a clause is a **unit clause**, i.e. it contains only a single literal l, We can apply the following two rules:

- Every clause (other than the unit clause itself) containing l is removed;
- in every clause that contains $\neg l$, this literal is deleted.

Example

$$c$$
$$d$$
$$a$$

The following set of clauses can be simplified by unit propagation because it contains the unit clause c.

Pure literal elimination

Pure literal elimination

If a literal l occurs with only one polarity in the formula, it is called **pure**. Pure literals can always be assigned in a way that makes all clauses containing them true.

Example

$$\phi = (a \vee b) \wedge (\neg a \vee c) \wedge (\neg c \vee d \vee b) \wedge a$$

$$\begin{gathered} a \vee b \\ \neg a \vee c \\ \neg c \vee d \vee b \\ a \end{gathered}$$

We have the following variable assignment:

$$\begin{array}{ll} \mathrm{b} = \mathrm{True} & \mathrm{d} = \mathrm{True} \\ \mathrm{a} = \mathrm{True} & \mathrm{c} = \mathrm{True} \end{array}$$

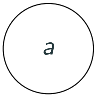
Let's take the following **formula** Φ, represented by a set of clauses :

$$a \vee b \vee c$$
$$a \vee \neg b \vee \neg c$$
$$a \vee b \vee \neg c$$
$$\neg a \vee \neg b \vee c$$

Action taken

First of all, we choose **arbitrarily** a variable

$$a \vee b \vee c$$
$$a \vee \neg b \vee \neg c$$
$$a \vee b \vee \neg c$$
$$\neg a \vee \neg b \vee c$$



Backtracking

Action taken

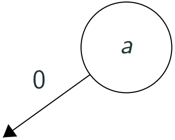
We make a **choice**, the variable takes the value **0**. Some clauses become **true** ✓.

$$a \lor b \lor c$$

$$a \lor \neg b \lor \neg c$$

$$a \lor b \lor \neg c$$

$$\neg a \lor \neg b \lor c \ \checkmark$$



Action taken

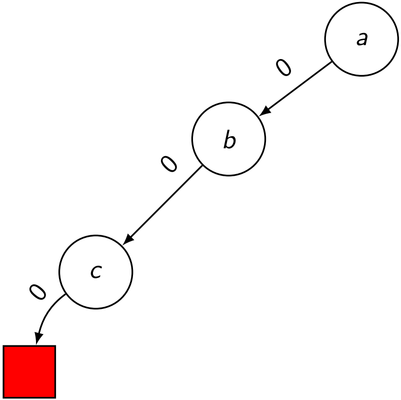
After severals decisions, we have a conflict **conflit x**

$a \lor b \lor c$ **x**

$a \lor \neg b \lor \neg c$ ✓

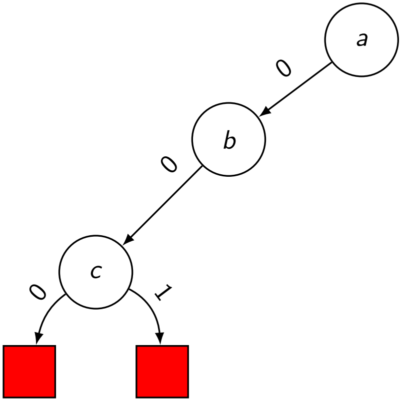
$a \lor b \lor \neg c$ ✓

$\neg a \lor \neg b \lor c$ ✓



Action taken
We make an backtrack to the upper level and try to assign the variable by the **opposite** value.

$a \lor b \lor c$ ✓
$a \lor \neg b \lor \neg c$ ✓
$a \lor b \lor \neg c$ ✗
$\neg a \lor \neg b \lor c$ ✓



Action taken

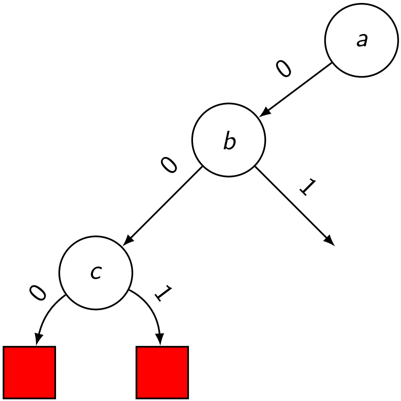
Actions are repeated until all clauses are true or have gone through the tree.

$a \lor b \lor \neg c$ ✓

$a \lor \neg b \lor \neg c$

$a \lor b \lor \neg c$ ✓

$\neg a \lor \neg b \lor c$ ✓



Backtracking

Action taken

Actions are repeated until all clauses are true or have gone through the tree.

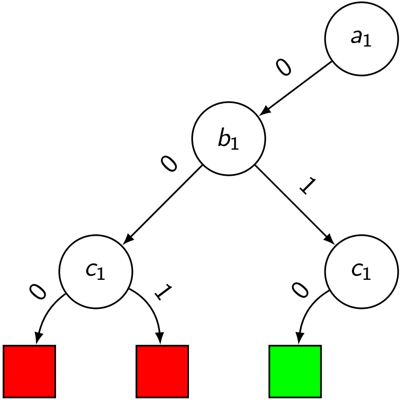
$a \vee b \vee \neg c$ ✓

$a \vee \neg b \vee \neg c$ ✓

$a \vee b \vee \neg c$ ✓

$\neg a \vee \neg b \vee c$ ✓

☺



DPLL algorithm

Satisfiability

- To prove the satisfiability, it is enough to find an assignment of the variables **valid**.
- To prove the non-satisfiability, one must go through the tree **entirely**.

```
function DPLL(a set of clause Φ)
    if Φ is a consistent set of literals then
        return true;
    end if
    if Φ contains an empty clause then
        return false;
    end if
    for every unit clause L in Φ do
        Φ ← unit-propagate(L, Φ);
    end for
    for every literal l that occurs pure in Φ do
        Φ ←pure-literal-assign(l, Φ);
    end for
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ l) or DPLL(Φ ∧ l̄);
end function
```

Algorithme CDCL (conflict-driven clause learning)

Innovations

- Learning phase : Thanks to the resolution rule, we can create new clauses, which are called learned clauses;

Resolution rule [Robinson '65]

Let C_1 and C_2 two clauses such that:

$$C_1 = a \vee b \vee c \vee \mathrm{d}$$

$$C_2 = \neg d \vee e \vee f$$

We apply the resolution rule on d:

$$\overbrace{(a \vee b \vee c \vee d)}^{C_1} \wedge \overbrace{(\neg d \vee e \vee f)}^{C_2}$$

$$\vdash a \vee b \vee c \vee e \vee f$$

Resolution rule [Robinson '65]

More formally,

Let C_1 and C_2 be two clauses, the resolution rule gives us:

$$(C_1 \vee x) \wedge (C_2 \vee \neg x) \vdash C_1 \vee C_2$$

We call $C_1 \vee C_2$ the resolvent of $C_1 \vee x$ and $C_2 \vee \neg x$.

Resolution rule [Robinson '65]

More formally,

Let C_1 and C_2 be two clauses, the resolution rule gives us:

$$(C_1 \lor x) \land (C_2 \lor \neg x) \vdash C_1 \lor C_2$$

We call $C_1 \lor C_2$ the resolvent of $C_1 \lor x$ and $C_2 \lor \neg x$.

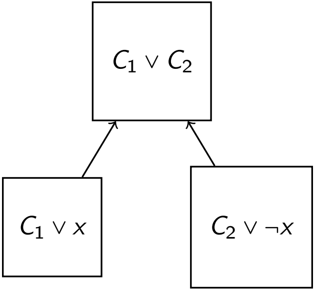


Figure 3: Graphical representation of the resolution

Algorithme CDCL (conflict-driven clause learning)

Innovations

- Learning phase : Thanks to the resolution rule, we can create new clauses, which are called learned clauses;
- non-chronological backtracking : It becomes possible to go back to a decision more **old** than the last decision;
- restarts: It is permitted for the solver to start the search again at any time.

Φ: set of initials clauses

Σ: set of learnts clauses.

Algorithm 1 modern SAT solvers

While $\square \notin \Phi \cup \Sigma$ **do**

  $C \leftarrow learntClause()$

  $\Sigma = \Sigma \cup C$

  If $Full(\Sigma)$ **Then**

    $\Delta = DeleteClause(\Sigma)$

    $\Sigma = \Sigma \backslash \Delta$

  End If

End While

Our work

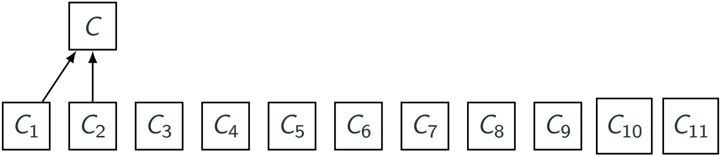
Sequence of resolution with a graph representation

Formula

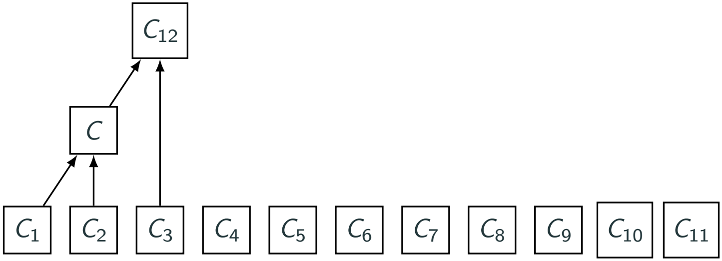
Let $\Phi = C_1 \wedge C_2 \wedge .. \wedge C_{11}$, s.t. $\forall i \in [1..11], C_i$ any clause.

$\boxed{C_1}$ $\boxed{C_2}$ $\boxed{C_3}$ $\boxed{C_4}$ $\boxed{C_5}$ $\boxed{C_6}$ $\boxed{C_7}$ $\boxed{C_8}$ $\boxed{C_9}$ $\boxed{C_{10}}$ $\boxed{C_{11}}$

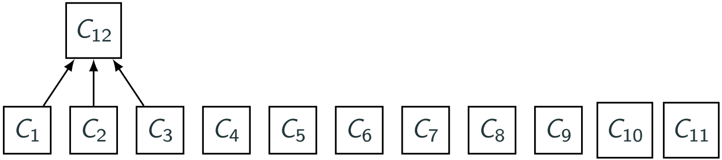
Sequence of resolution with a graph representation



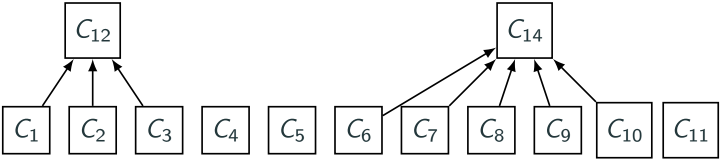
Sequence of resolution with a graph representation



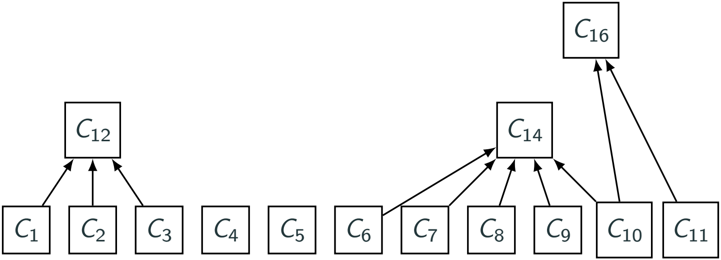
Sequence of resolution with a graph representation



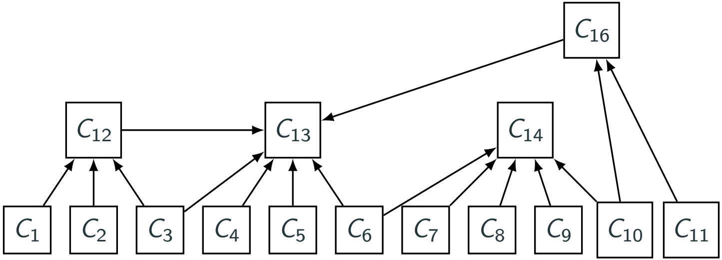
Sequence of resolution with a graph representation



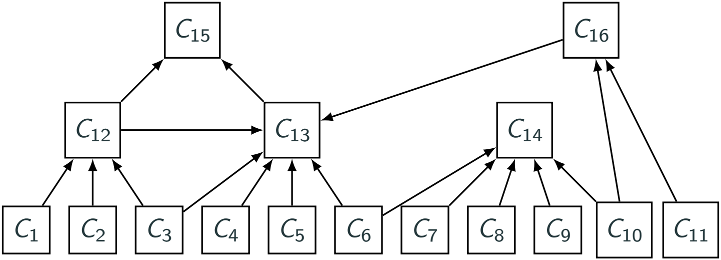
Sequence of resolution with a graph representation



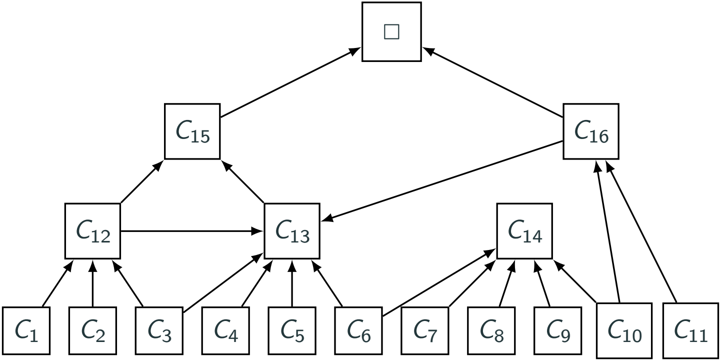
Sequence of resolution with a graph representation



Sequence of resolution with a graph representation



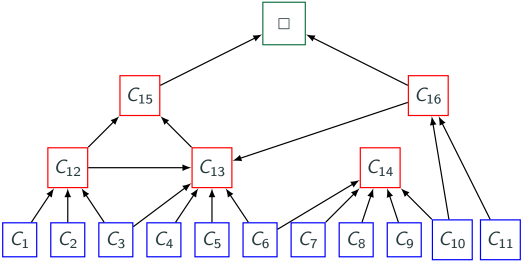
Sequence of resolution with a graph representation



Resolution graph

Definition
The resolution graph is a directed acyclic graph (or **DAG**) such that:

- Leaves are **initials** clauses;
- Internal nodes are **learnts** clauses;
- The root is the **empty** clause.



We call it the proof produced by the SAT solver.

Representation of a real proof

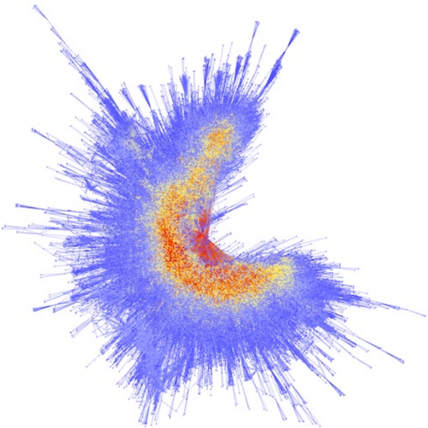


Figure 4: Force-Directed layout of the Dependency Graph for the benchmark `een-pico-prop-05`. The color shows the **degree** of each node.

Information

Formula

clauses: 55585

variables: 50076

Conflicts

conflicts: 59792

CPU time: **6s**

Graph

vertices: 51274

edges: **960620**

Examples of SAT solvers

Pysat: A toolkit for SAT-based prototyping in Python

PySAT

PySAT is a Python (2.7, 3.4+) toolkit, which aims at providing a simple and unified interface to a number of state-of-art SAT solvers.

Pros:

- In Python;
- Great documentation;
- Easy to install ($pipinstallpython - SAT$)

(Major) Cons:

- Less powerful than other solvers

Pysat: A toolkit for SAT-based prototyping in Python

```
>>> from pysat.solvers import Glucose3
>>>
>>> g = Glucose3()
>>> g.add_clause([-1, 2])
>>> g.add_clause([-2, 3])
>>> print g.solve()
>>> print g.get_model()
...
True
[-1, -2, -3]
```

Figure 5: Trivial example using PySAT

Glucose

Glucose

`Glucose` is a winning award `SAT` solvers developped in `LaBRI` and `CRIL` by Laurent Simon and Gilles Audemard.

Pros:

- Developped in LaBRI ☺
- Powerful;
- Relatively easy to implement.

Glucose

```
% ./glucose ~/Desktop/These/Benchs-POS14/2008-satrace/satelited/een-pico-prop05-75.satelited.cnf.gz
c
c This is glucose 4.0 --  based on MiniSAT (Many thanks to MiniSAT team)
c
c ========================================[ Problem Statistics ]===========================================
c |                                                                                                       |
c |  Number of variables:        18188                                                                    |
c |  Number of clauses:          87504                                                                    |
c |  Parse time:                  0.05 s                                                                  |
c |                                                                                                       |
c | Preprocesing is fully done
c |  Eliminated clauses:          0.01 Mb                                                                 |
c |  Simplification time:         0.05 s                                                                  |
c |                                                                                                       |
c ========================================[ MAGIC CONSTANTS ]==============================================
c | Constants are supposed to work well together :-)                                                      |
c | however, if you find better choices, please let us known...                                           |
c |-------------------------------------------------------------------------------------------------------|
c | Adapt dynamically the solver after 100000 conflicts (restarts, reduction strategies...)               |
c |-------------------------------------------------------------------------------------------------------|
c |                                 |                                |                                    |
c | - Restarts:                     | - Reduce Clause DB:            | - Minimize Asserting:              |
c |   * LBD Queue    :     50       |   * First     :   2000         |    * size <  30                    |
c |   * Trail  Queue :   5000       |   * Inc       :    300         |    * lbd  <   6                    |
c |   * K            :   0.80       |   * Special   :   1000         |                                    |
c |   * R            :   1.40       |   * Protected :  (lbd)< 30     |                                    |
c |                                 |                                |                                    |
c ==================================[ Search Statistics (every  10000 conflicts) ]=========================
c |                                                                                                       |
c |          RESTARTS           |          ORIGINAL         |              LEARNT              | Progress |
c |       NB   Blocked  Avg Cfc |    Vars  Clauses Literals |   Red   Learnts    LBD2  Removed |          |
c =========================================================================================================
c |       38         0      263 |   18131    87321   328060 |     2      5160    2076     4716 | 0.236 % |
c |       89        85      224 |   18119    87230   327836 |     3      8927    3811    10918 | 0.302 % |
c |      129       161      232 |   18113    87198   327772 |     3     18921    5193    10918 | 0.335 % |
c |      200       227      200 |   18111    87186   327748 |     4     19552    5975    20285 | 0.346 % |
c |      243       312      205 |   18108    87165   327694 |     4     29479    6770    20285 | 0.363 % |
c |      324       376      185 |   18108    87165   327694 |     5     26139    7200    33625 | 0.363 % |
c |      418       406      167 |   18105    87140   327621 |     5     35430    7494    33625 | 0.379 % |
c =========================================================================================================
c restarts              : 451 (167 conflicts in avg)
c blocked restarts      : 419 (multiple: 149)
c last block at restart : 451
c nb ReduceDB           : 5
c nb removed Clauses    : 33625
c nb learnts DL2        : 7613
c nb learnts size 2     : 2311
c nb learnts size 1     : 72
c conflicts             : 75590          (28861 /sec)
c decisions             : 338134         (0.00 % random) (129101 /sec)
c propagations          : 22747444       (8685068 /sec)
c nb reduced Clauses    : 4211
c CPU time              : 2.61914 s

s UNSATISFIABLE
```

Conclusion

Thank you !

Questions ? ☺